O'REILLY®

серия Head First

Учимся программировать

с примерами на Python

Книга, которая учит думать как программист

Как избегать неверных решений и подводных камней

Начните карьеру программиста прямо сейчас

Как не допускать глупых синтаксических ошибок

Все, что вы хотели знать о программировании, но боялись спросить

Потренируйте мозг с помощью задач и упражнений

Эрик Фримен

ДИАЛЕКТИКА

серия Head First

Учимся программировать

с примерами на Python

Head First Learn to Code

Eric Freeman

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

серия Head First

Учимся программировать

с примерами на Python

Ах, если бы существовала такая книга по основам программирования, которая была бы увлекательнее похода к стоматологу и информативнее налоговой декларации! Знаю, это лишь мои мечты...

Эрик Фримен

ДИАЛЕКТИКА

Москва • Санкт-Петербург
2020

ББК 32.973.26-018.2.75
Ф88
УДК 004.43(075.8)

Компьютерное издательство "Диалектика"

Перевод с английского *И.В. Василенко, В.Р. Гинзбурга*

Под редакцией *В.Р. Гинзбурга*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Фримен, Эрик

Ф88 Учимся программировать с примерами на Python (серия Head First). : Пер. с англ. – СПб. : ООО "Диалектика", 2020. – 640 с. : ил. – Парал. тит. англ.

ISBN 978-5-907144-98-9 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства O'Reilly Media, Inc.

Authorized Russian translation of the English edition of *Head First Learn to Code* (ISBN 978-1-491-95886-5) © 2018 Eric Freeman.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the Publisher.

Научно-популярное издание

Эрик Фримен

Учимся программировать с примерами на Python (серия Head First)

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-907144-98-9 (рус.)
ISBN 978-1-491-95886-5 (англ.)

© 2020 ООО "Диалектика"
© 2018 Eric Freeman

До группы KISS у меня не было опыта выступлений в рок-группе, участники которой носят макияж.

Джин Симмонс

Эрик Фримен

Эрик Фримен – "из той редкой породы людей, которые одинаково хорошо умеют писать книги, программировать, вести бизнес и заниматься аналитикой", как описывает его Кэти Сиерра, один из создателей серии книг "Head First". И его резюме в полной мере соответствует данному описанию. По образованию Эрик – компьютерный инженер, защитивший диплом у самого Дейвида Гелернтера в Йельском университете. По окончании учебы занимал должность главного инженера сайта Disney.com, принадлежавшего медиагиганту The Walt Disney Company. Впоследствии работал в издательстве O'Reilly Media, нескольких стартапах и даже в НАСА. Одно из его изобретений было лицензировано для использования во всех компьютерах Mac. Последние 15 лет Эрик остается одним из самых продаваемых авторов в сегменте компьютерной литературы. Тематика его книг варьируется от основ веб-программирования до принципов проектирования программного обеспечения.

В настоящее время Эрик Фримен руководит компанией WickedlySmart, LLC. Проживает вместе с женой и дочерью в Остине, штат Техас.

Можете написать Эрику электронное письмо по адресу `eric@wickedlysmart.com` или посетить его сайт `http://wickedlysmart.com`.

Оглавление (кратко)

	Введение	23
1	Приступим	37
2	Сначала было значение	69
3	Принятие решений	109
4	Структуры данных	161
5	Функциональный код	215
4а	Наведение порядка в данных	261
6	Сведем все воедино	281
7	Модульное программирование	327
8	И снова об индексах и циклах	377
9	Длительное хранение данных	429
10	Так хочется большего	471
11	Интерактивные возможности	503
12	Путешествие в страну объектов	559
	Приложение: Десять ключевых тем (которые не были рассмотрены)	611

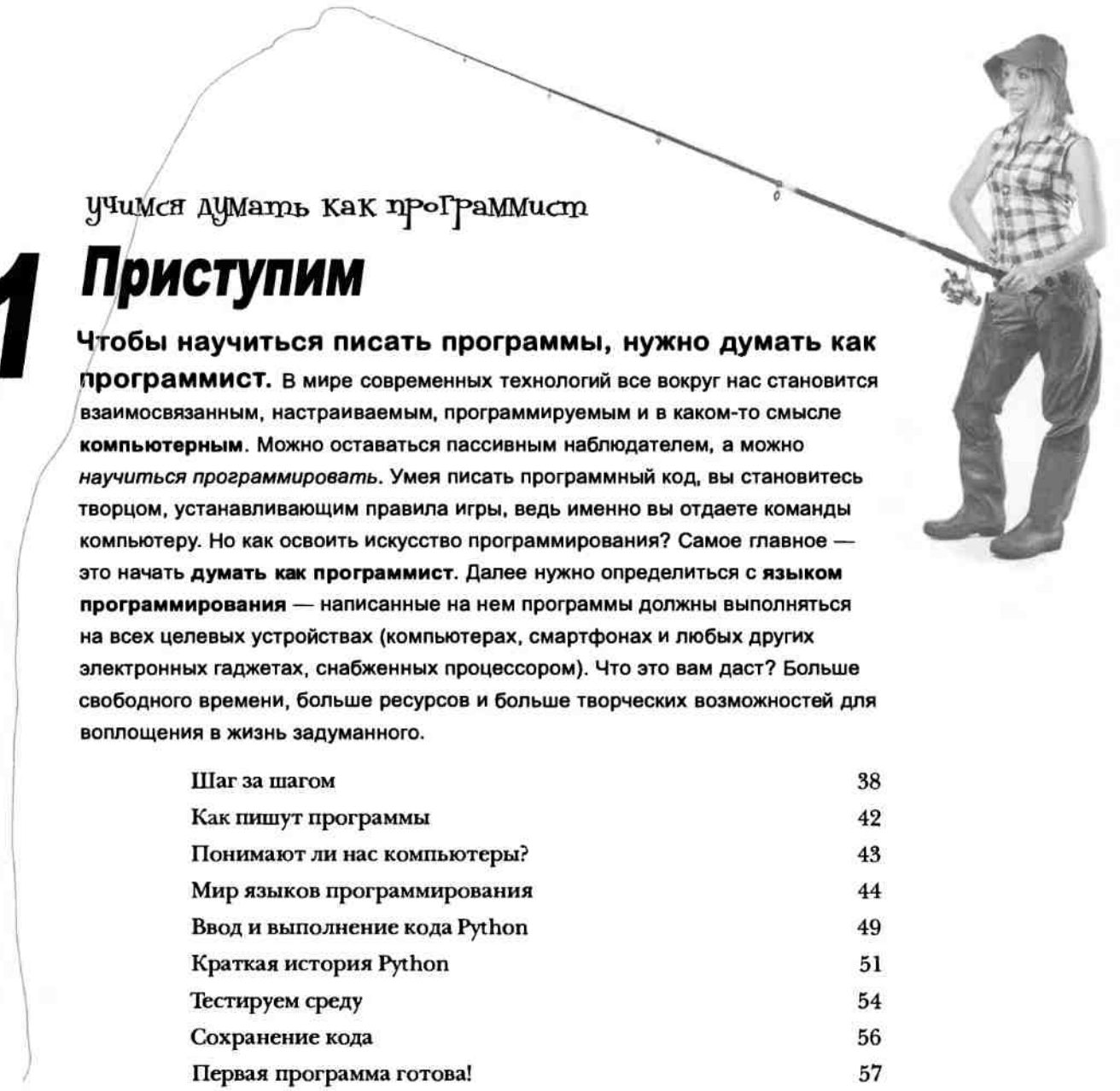
Содержание (подробно)

Введение

Настройте свой мозг на изучение программирования.

Конечно, вам хочется чему-то научиться, но у мозга могут быть свои планы на этот счет. Он контролирует, чтобы вы не забивали себе голову ненужной информацией. Есть ведь вещи и поважнее, например какой фильм пойти посмотреть на выходных или что надеть на вечеринку. Как же убедить мозг, что книга по программированию содержит нужную информацию?

Для кого эта книга	24
Мы знаем, о чем вы думаете	25
Мы считаем читателей книги учениками	26
Метапознание: осознание знаний	27
Вот что сделали мы	28
Вот что можете сделать вы, чтобы настроить свой мозг на обучение	29
Это важно	30
Необходимо установить Python	32
Как работать с исходными кодами	34
Благодарности	35
Команда рецензентов	36



учимся думать как программист

1 Приступим

Чтобы научиться писать программы, нужно думать как программист. В мире современных технологий все вокруг нас становится взаимосвязанным, настраиваемым, программируемым и в каком-то смысле **компьютерным**. Можно оставаться пассивным наблюдателем, а можно *научиться программировать*. Умея писать программный код, вы становитесь творцом, устанавливающим правила игры, ведь именно вы отдаете команды компьютеру. Но как освоить искусство программирования? Самое главное — это начать **думать как программист**. Далее нужно определиться с **языком программирования** — написанные на нем программы должны выполняться на всех целевых устройствах (компьютерах, смартфонах и любых других электронных гаджетах, снабженных процессором). Что это вам даст? Больше свободного времени, больше ресурсов и больше творческих возможностей для воплощения в жизнь задуманного.

Шаг за шагом	38
Как пишут программы	42
Понимают ли нас компьютеры?	43
Мир языков программирования	44
Ввод и выполнение кода Python	49
Краткая история Python	51
Тестируем среду	54
Сохранение кода	56
Первая программа готова!	57
Phrase-O-Matic	61
Запуск кода на выполнение	62

значения, переменные и типы данных

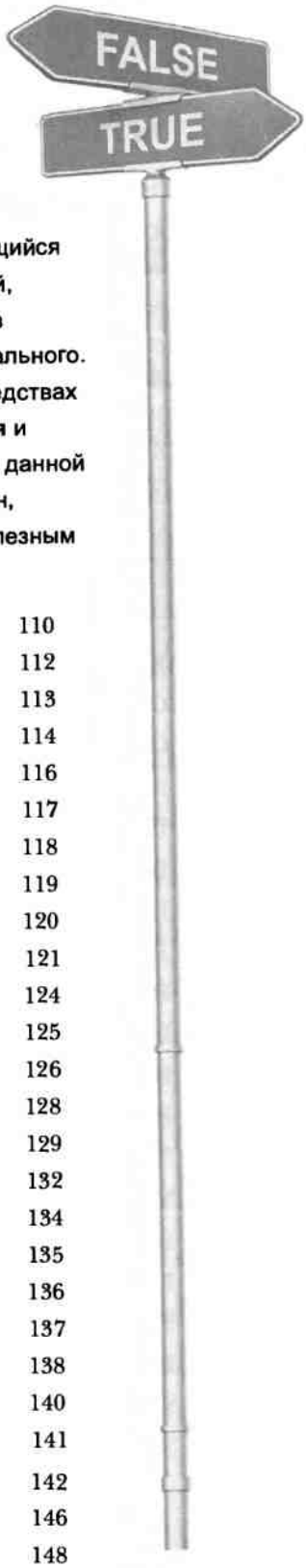
2 Сначала было значение

В реальности компьютеры умеют делать всего две вещи: сохранять значения и выполнять над ними операции. А как же редактирование текста, ретуширование фотографий и онлайн-покупки, спросите вы? Удивительно, но решение любых подобных задач в конечном счете сводится к выполнению **простых операций** над не менее **простыми значениями**. Человек, обладающий вычислительным мышлением, понимает, что это за операции и значения и как с их помощью создать что-то более сложное. В этой главе мы познакомимся с типами значений, узнаем, какие операции можно выполнять над ними и какую роль во всем этом играют переменные.

Калькулятор возраста собаки 70
От псевдокода к программному коду 72
Этап 1: запрос входных данных 73
Как работает функция `input()` 74
Сохранение значений в переменных 74
Сохранение вводимых данных в переменной 75
Этап 2: еще один запрос 75
Пора выполнить код 76
Ввод кода в редакторе 79
Подробнее о переменных 80
Добавляем выражения 81
Изменение переменных 82
Приоритет операторов 83
Вычисление выражений согласно приоритету операторов 84
Руки прочь от клавиатуры! 87
Этап 3: вычисление возраста собаки 88
У нас проблема! 89
Неизбежность ошибок 90
Еще немного отладки 92
Типы данных Python 94
Исправление ошибки 95
Ура, заработало! 96
Этап 4: вывод результата 97

Булевы значения, условия и циклы

3 Принятие решений



Думаю, вы заметили, что наши предыдущие программы были не особо интересными. Содержащийся в них код представлял собой простую последовательность инструкций, выполняемых интерпретатором **сверху вниз**. Никаких тебе поворотов сюжета, внезапных изменений, сюрпризов — в общем, ничего оригинального. Чтобы программа могла стать более интересной, она нуждается в средствах **принятия решений**, позволяющих **управлять ходом ее выполнения** и многократно повторять одни и те же действия. Этим мы и займемся в данной главе. По ходу дела вы научитесь играть в загадочную игру шоушилин, познакомитесь с персонажем по имени Буль и увидите, насколько полезным может быть тип данных, содержащий всего два возможных значения.

Сыграем в игру? 110
Общий алгоритм игры 112
Компьютер делает свой выбор 113
Применение случайных чисел 114
Истина или ложь? 116
Булевы значения 117
Принимаем решения 118
Дополнительные условия 119
Вернемся к игре 120
Пользователь делает свой выбор 121
Проверяем, что выбрал пользователь 124
Код определения ничьей 125
Кто выиграл? 126
Реализация логики игры 128
Подробнее о булевых операторах 129
Вывод имени победителя 132
А где документация? 134
Добавление комментариев в код 135
Завершение игры 136
Проверка правильности введенных данных 137
Проверка и уточнение выражения 138
Повторный вывод запроса 140
Многократное повторение действий 141
Как работает цикл `while` 142
Как использовать цикл `while` для повторного вывода запроса 146
Итак, наша первая игра готова! 148

списки и циклы

4 Структуры данных

Числа, строки и булевы значения — не единственные типы данных в программировании. Пока что мы использовали только **примитивные типы**: числа с плавающей точкой, целые числа, строки и булевы значения (например, `3.14`, `42`, `"имя собаки"` и `True`). И хотя их возможности достаточно велики, время от времени в программах приходится иметь дело с большими наборами данных, такими как все добавленные в корзину покупки, названия всех известных звезд или содержимое всего каталога товаров. Для эффективной работы с подобными наборами нужны **структуры данных**. В этой главе вы познакомитесь с новым типом данных — **списком**, хранящим коллекцию значений. Списки позволяют структурировать данные, вместо того чтобы хранить каждый элемент в отдельной переменной. Вы узнаете о том, как работать со списком в целом и как поочередно перебирать его элементы в цикле `for`. Все это расширит ваши возможности по работе с данными.

Поможете разобраться?	162
Сохранение нескольких значений в одной переменной	163
Работа со списками	164
Размер списка	167
Извлечение последнего элемента списка	168
В Python все намного проще	168
Отрицательные индексы	169
Продолжим подсчет мыльных пузырьков	171
Просмотр элементов списка	174
Устранение ошибки вывода	175
Правильный способ вывода результата	176
Цикл `for` предпочтителен при работе со списками	178
Обработка диапазона чисел в цикле `for`	181
Другие операции с диапазонами	182
Составление отчета	184
Построение списка с нуля	192
Другие операции со списками	193
Тест-драйв готового отчета	197
И победителями становятся...	197
Тестирование самого выгодного раствора	201

функции и абстракции

5 Функциональный код

Вы уже многому научились. Переменные и типы данных, условные конструкции и циклы — этих инструментов вполне достаточно, чтобы написать почти любую программу. Но зачем останавливаться на достигнутом? Следующий шаг — научиться **создавать абстракции** в коде. Звучит пугающе, хотя ничего страшного в этом нет. Абстракции — ваш спасательный круг. Они упрощают процесс разработки, давая возможность создавать более эффективные и функциональные программы. Абстракции позволяют упаковать код в небольшие блоки, удобные для повторного использования. С ними вы перестанете концентрироваться на низкоуровневых деталях и сможете программировать на высоком уровне.

Что не так с кодом? 217
Превращение блока кода в функцию 219
Функция создана – можно использовать 220
Как работает функция 220
Функции могут возвращать значения 228
Вызов функции, возвращающей значение 229
Улучшение имеющегося кода 231
Выполнение кода 232
Создание абстракций в коде выбора аватарки 233
Тело функции `get_attribute()` 234
Вызов функции `get_attribute()` 235
И снова о переменных 237
Область видимости переменной 238
Передача переменных в функцию 239
Вызов функции `drink_me()` 240
Использование глобальных переменных в функциях 243
Размышления о параметрах: значения по умолчанию и ключевые слова 246
Параметры по умолчанию 246
Первыми всегда указываются обязательные параметры 247
Аргументы с ключевыми словами 248
Как правильно использовать все доступные возможности 248

сортировка и вложенные списки

4 (продолжение) Наведение порядка в данных

Иногда стандартный порядок сортировки данных нас не устраивает. Например, у вас есть список лучших результатов в аркадных играх, и вы хотите упорядочить его по названиям игр. Или же это может быть список сотрудников, регулярно опаздывающих на работу, и вы хотите узнать, кто из них проштрафился чаще остальных. Для решения подобных задач нужно знать, как сортировать данные и как самостоятельно настраивать порядок сортировки. В этой главе мы также изучим вложенные циклы и выясним, насколько эффективен написанный нами код.

Пузырьковый метод сортировки	264
Псевдокод алгоритма пузырьковой сортировки	267
Реализация пузырьковой сортировки на Python	270
Правильное определение номеров растворов	272

текст, строки и эвристические алгоритмы

6 Сведем все воедино

Вы уже многое знаете. Пора задействовать знания по максимуму. В этой главе мы сведем все воедино и начнем писать **сложный код**. В то же время мы продолжим пополнять арсенал знаний и навыков программирования. В частности, будет показано, как написать код, который **загружает текст**, обрабатывает его и выполняет **анализ текстовых данных**. Вы также узнаете, что такое **эвристический алгоритм** и как его реализовать. Приготовьтесь — вас ждет по-настоящему сложная и насыщенная глава.

Добро пожаловать в науку о данных	282
Как вычисляется индекс удобочитаемости	283
План действий	284
Общий псевдокод	285
Нужен текст для анализа	286
Создание основной функции	288
Подсчет числа слов в тексте	289
Подсчет общего числа предложений	293
Функция `count_sentences()`	294
Подсчет слогов: знакомство с эвристическими алгоритмами	300
Создание эвристического алгоритма	303
Начинаем кодировать эвристический алгоритм	304
Подсчет гласных	305
Игнорирование последовательных гласных	305
Код игнорирования последовательных гласных	306
Учет конечной 'e', а также знаков препинания	308
Создание срезов (подстрок)	310
Завершение эвристического алгоритма	312
Код вычисления индекса удобочитаемости	314
Дальнейшие улучшения	319

классы, объекты, методы и модули

7 Модульное программирование

Написанные вами программы становятся все более сложными. Это заставляет тщательнее продумывать структуру и организацию программ. Вы уже знаете, что функции позволяют группировать строки кода в компактные блоки, удобные для повторного использования. Вы также знаете, что наборы функций и переменных можно объединять в модули, подключаемые к программам. В этой главе мы вернемся к теме модулей и рассмотрим, как работать с ними эффективнее (и даже делиться ими с другими пользователями). Кроме того, вас ждет знакомство с ключевыми строительными блоками программ — *объектами*. Вы увидите, что в Python они встречаются повсеместно, даже там, где вы не ожидаете их встретить.

Знакомство с модулями	330
Глобальная переменная `__name__`	332
Обновление файла *analyze.py*	333
Использование файла *analyze.py* в качестве модуля	335
Добавление документирующих строк в файл *analyze.py*	337
Знакомство с другими модулями Python	341
Что еще за черепашки?	342
Создание собственной черепашки	344
Изучение черепашек	345
Еще одна черепашка	347
Кто вы, черепашки?	350
Что такое объект?	351
А что тогда класс?	352
Использование объектов и классов	354
Что такое атрибуты и методы?	355
Объекты и классы повсюду	356
Черепашьи бега	358
Планирование игры	359
Приступаем к написанию кода игры	360
Код настройки игры	361
Спешка ни к чему!	362
Начало забега	364

рекурсия и словари

8 И снова об индексах и циклах

Настало время перейти на новый уровень. Пока что мы придерживались итеративного стиля программирования, создавая структуры данных наподобие списков, строк и диапазонов чисел и обрабатывая их в циклах поэлементно. В этой главе мы пересмотрим подход к программированию и выбору структур данных. Наш новый стиль программирования будет требовать написания *рекурсивного*, т.е. вызывающего самого себя, кода. Кроме того, вы познакомитесь с новым типом данных — *словарем*, который больше напоминает ассоциативный массив, а не список. Воспользовавшись новыми знаниями, мы научимся эффективно решать множество задач. Заранее предупредим: это достаточно сложные темы. Пройдет какое-то время, прежде чем вы освоите все нюансы, но поверьте, оно того стоит.

Иной стиль программирования 378
Мы пойдем другим путем 379
Код для обоих случаев 380
Давайте попрактикуемся 383
Нахождение палиндромов рекурсивным способом 384
Код рекурсивной функции распознавания палиндромов 385
Антисоциальная сеть 396
Знакомство со словарем 398
Просмотр словаря 400
Словари в антисоциальной сети 402
Добавление новых атрибутов 404
Ключевая особенность антисоциальной сети 406
Определение самого антисоциального пользователя 407
Слово за вами! 408
Можно ли запомнить результаты работы функции? 412
Мемоизация 413
Анализ функции `koch()` 416
Фрактал Коха 418

Чтение и запись файлов

9 Длительное хранение данных

Вы умеете хранить данные в переменных, но как только программа завершается — бах! — они исчезают навсегда. Вот для чего нужно *постоянное* хранилище данных. Большинство компьютерных устройств оснащено такими хранилищами, в частности, жесткими дисками и флеш-накопителями. Кроме того, данные можно хранить в облачной службе. В этой главе вы узнаете, как писать программы, умеющие работать с файлами. Зачем это нужно? Вариантов множество: сохранение конфигурационных настроек, создание файлов отчетов, обработка графических файлов, поиск сообщений электронной почты — список можно продолжать еще долго.

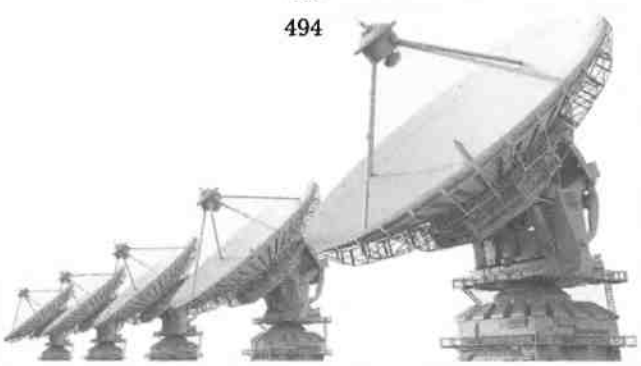
Игра в слова	430
Логика игры	432
Считывание истории из файла	435
Путь к файлу	436
Не забудьте все закрыть в конце	438
Чтение файла в коде	439
Пора прекратить...	442
Вернемся к игре	443
Как узнать, что мы достигли конца?	445
Чтение шаблона игры	446
Обработка текста шаблона	447
Исправление ошибки с помощью метода `strip()`	449
Окончательное исправление ошибки	450
Остались проблемы посерьезнее	451
Обработка исключений	453
Явная обработка исключений	454
Обработка исключений в игре в слова	456
Последний этап: сохранение результата	457
Обновление остального кода	457

Веб-службы

10 Так хочется большего

Вы научились писать неплохие программы, но всегда хочется большего. В Интернете хранится просто невероятное количество **данных**, которые так и ждут, чтобы с ними поработали. Нужен прогноз погоды? Как насчет базы данных рецептов? Или вас интересуют спортивные результаты? А может, ваш интерес — музыкальная база данных с информацией о музыкантах, альбомах и композициях? Всю эту информацию можно получить с помощью веб-служб. Для работы с ними нужно знать, как организуется передача данных в Интернете, а также познакомиться с парочкой модулей Python: `requests` и `json`. В этой главе вы изучите возможности **веб-служб** и повысите свой уровень знаний Python до заоблачных высот. Так что приготовьтесь к путешествию в космос.

Расширение возможностей благодаря веб-службам	472
Как работает веб-служба	473
Адрес веб-службы	474
Небольшое обновление Python	477
Установка обновления	478
Нам нужна интересная веб-служба	479
Возможности Open Notify	480
Передача данных в формате JSON	481
Вернемся к модулю `requests`	483
Собираем все вместе: создание запроса к службе Open Notify	485
Работа с данными JSON в Python	486
Применение модуля `json`	487
Представим результат в графическом виде	488
Знакомство с объектом `screen`	489
Черепашка-космонавт	491
Черепашка может выглядеть как космическая станция	492
Забудьте об МКС. Где находимся мы?	493
Завершение программы	494



Виджеты, события и непредсказуемое поведение

11 Интерактивные возможности

Вам уже приходилось писать простые графические приложения, но у них не было полноценного графического интерфейса. Другими словами, приложения не позволяли пользователям взаимодействовать с экранными элементами. Чтобы это стало возможным, необходимо применить иную модель выполнения программы, в рамках которой приложение **реагирует** на действия пользователя. Пользователь щелкнул на кнопке? Программа должна знать, как реагировать на такое событие, и быть готовым к нему. Процесс создания графического интерфейса сильно отличается от привычного процедурного стиля программирования, который мы применяли до сих пор. Нам придется совершенно по-иному подойти к решению задач. В этой главе вы создадите свой первый графический интерфейс, причем не какое-то примитивное экранное приложение, а нечто намного более интересное. Мы напишем игровой симулятор искусственной жизни с непредсказуемым поведением.

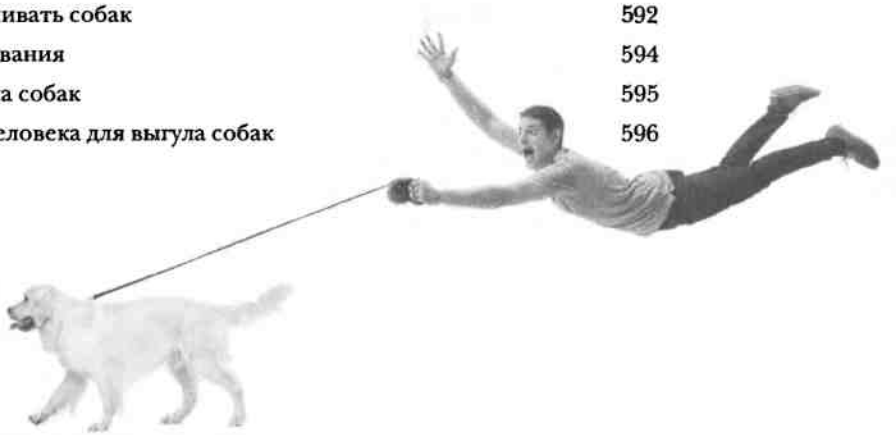
Откройте для себя удивительный мир игры "Искусственная жизнь" 504
Правила игры "Жизнь" 505
Что нам предстоит создать 508
Дизайнерские решения 509
Создание симулятора игры "Жизнь" 512
Построение модели данных 513
Вычисление поколений клеток 514
Завершение кода модели 518
Экранное представление 521
Создание первого виджета 522
Добавление остальных виджетов 523
Исправление макета 524
Размещение виджетов по сетке 525
Переходим к написанию контроллера 527
Новый способ вычислений 530
Обработка щелчков мыши 531
Обработка событий, связанных с кнопкой Start/Pause 533
Реализация кнопки Start/Pause 534
Другой тип событий 535
А как же метод `after()`? 537
Непосредственное редактирование клеток 540
Обработчик `grid_handler()` 541
Добавление шаблонов 542
Обработчик событий для объекта `OptionMenu` 543
Создание собственных фигур 545
Загрузчик шаблонов 546
Улучшаем симулятор игры "Жизнь" 553

объектно-ориентированное программирование

12 Путешествие в страну объектов

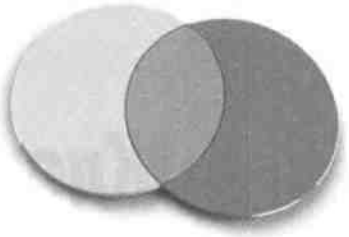
В предыдущих главах для создания абстракций в коде применялись функции. Мы всячески придерживались **процедурного** подхода, используя в функциях простые выражения, условные конструкции и циклы `for/while` — ничего такого, что относилось бы к **объектно-ориентированному** программированию. Конечно, вы познакомились с объектами и научились применять их, но вам еще ни разу не приходилось создавать классы и писать полностью объектно-ориентированный код. Пришло время попрощаться со скучным процедурным прошлым. Нас ожидает новый мир, полный объектов и надежд на лучшую жизнь.

Другой подход к структурированию программ 560
Преимущества объектно-ориентированного программирования 561
Разработка своего первого класса 563
Написание кода класса 564
Как работает конструктор 564
Метод `bark()` 567
Как работает метод 568
Немного о наследовании 570
Реализация класса `ServiceDog` 571
Подклассы 572
`ServiceDog` — это потомок класса `Dog` 573
Проверка принадлежности к классу 574
Переопределение и расширение поведения 578
Учим термины 580
Объекты внутри объектов 582
Создание отеля для собак 585
Реализация отеля для собак 586
Усовершенствование отеля для собак 589
Расширение функций отеля 590
Мы с тобой одной крови: полиморфизм 591
Учимся выгуливать собак 592
Сила наследования 594
Служба выгула собак 595
Как нанять человека для выгула собак 596



приложение: не попавшее в избранное

Десять ключевых тем (которые не были рассмотрены)



Вы прошли длинный путь и находитесь у финишной черты. Нам искренне жаль расставаться с вами, и, прежде чем отправить вас в самостоятельное плавание, нам бы хотелось дать последние наставления. Конечно, в одном маленьком приложении невозможно охватить все необходимое, хотя мы попытались это сделать, уменьшив размер шрифта до 0,00004 пункта. К сожалению, никому в издательстве так и не удалось прочитать столь микроскопический текст, поэтому нам пришлось выкинуть все лишнее, оставив только самое важное.

1. Списковые включения 612
2. Работа с датой и временем 613
3. Регулярные выражения 614
4. Другие типы данных: кортежи 615
5. Другие типы данных: множества 616
6. Написание серверного кода 617
7. Отложенные вычисления 618
8. Декораторы 619
9. Объекты первого класса и функции высшего порядка 620
10. Важные библиотеки 621

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам электронное письмо либо просто посетить наш сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

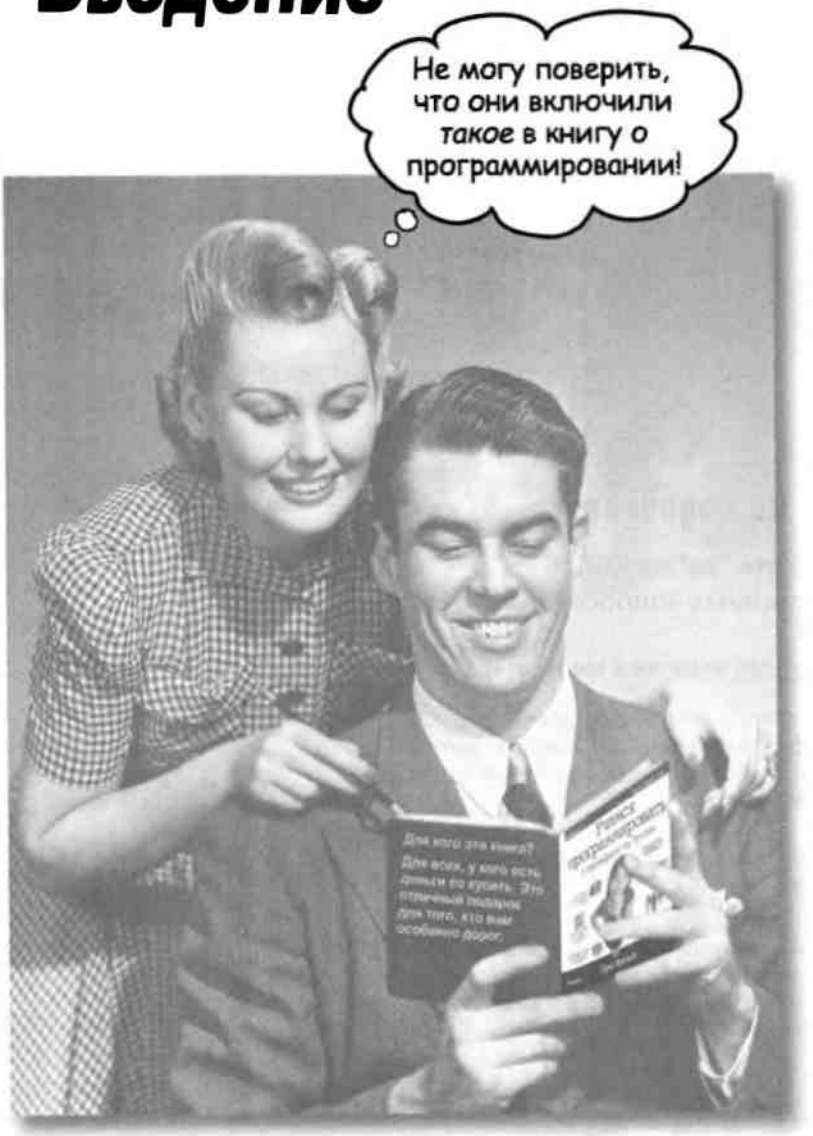
Наши электронные адреса:

E-mail: `info@dialektika.com`

WWW: `http://www.dialektika.com`

Как пользоваться книгой

Введение



Сейчас мы ответим на самый главный вопрос: "Зачем они включили ТАКОЕ в книгу о программировании?"

Для кого эта книга

Если вы ответите "да" на все нижеперечисленные вопросы...

1. Вы хотите научиться **писать, понимать** и **редактировать** программы?

2. Вы предпочитаете **неформальное общение**, а не **скучные лекции**?

...то эта книга для вас.

(Замечание от сбытовиков: это книга для всех, у кого есть деньги ее купить.)

Это НЕ справочник и не руководство по синтаксису Python. (Для этого есть Google, правильно?) Это книга для тех, кто хочет научиться программировать.

Кому эта книга не подойдет

Если вы ответите "да" на любой из нижеперечисленных вопросов...

1. **Вы абсолютный новичок в компьютерах?**

   Если вы не знаете, как работает компьютер, как управлять файлами и папками, как устанавливать приложения и набирать тексты, то лучше сначала освоить компьютер.

2. Вы опытный программист, который ищет **справочник**?

3. Вы **боитесь нестандартных решений**? Вам проще остаться дома, чем пойти в ресторан в кедах? Вы считаете, что юмору не место в книге по программированию?

...то эта книга не для вас.

Мы знаем, о чем вы думаете

"Это точно не комикс?"

"Что это за странные рисунки?"

"А что, так можно чему-то научиться?"

И мы знаем, о чем думает ваш мозг

Мозг всегда ищет новизну. Он напряжен, насторожен и *ждет* чего-то необычного. Таким его создала природа, и это помогает нам выживать.

В современном мире у нас не так много шансов попасть в лапы тигру, но мозг все равно контролирует ситуацию. Мало ли что...

А как мозг реагирует на обыденные, повседневные, рутинные ситуации? Всячески *игнорирует* их, чтобы они не мешали ему решать *основную* задачу — фиксировать только то, что *важно*. Он не утруждает себя запоминанием скучной информации, создавая для нее фильтр "это явно не важно".

Но как мозг определяет, что важно? Представьте, что вы вышли погулять в парк, и тут — бац! — из кустов выпрыгивает тигр. Что происходит в этот момент в вашей голове и вашем теле?

Вся жизнь проносится перед глазами... Всплеск адреналина... Запускается цепь *химических реакций*.

Именно они подсказывают мозгу...

Вот что важно! Думай быстрее!

А теперь представим, что вы сидите дома или в библиотеке. Тепло, комфортно, тигры не тревожат. Вы занимаетесь учебой, готовитесь к экзамену или пытаетесь справиться с заданием, на которое начальник отвел вам неделю, максимум полторы.

Тут-то и возникает проблема: ваш мозг исправно пытается вам помочь. Разве можно тратить ценные ресурсы памяти на запоминание столь очевидно несущественной информации? Лучше ведь использовать память для хранения чего-то действительно важного. Вот тигры — это важно. Лесные пожары — тоже важно. Идти или не идти в ресторан в кедах — ну куда уж важнее?!

Нельзя сказать мозгу: "Эй, приятель, спасибо, конечно, за заботу, но я тут книгу хочу почитать! Ты не смотри, что она скучная и не вызывает всплеска адреналина. Просто потрудись все запомнить".

Мы считаем читателей книги учениками

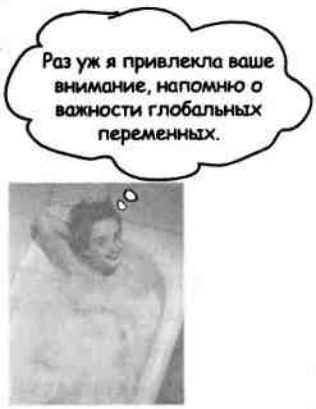
Как выучить что-либо? Сначала нужно понять, а затем постараться не забыть. Дело вовсе не в зубрежке, когда мы забиваем голову фактами. Последние исследования в области когнитивной науки, нейробиологии и педагогической психологии показали, что обучение не сводится к запоминанию текста на страницах учебника. Мозг включается по-другому.

Ключевые принципы обучения в книгах серии Head First

Визуальный подход. Изображения запоминаются лучше, чем текст, существенно повышая эффективность обучения (до 89% согласно современным исследованиям). Они также делают книгу более понятной.

Выноски и подписи к рисункам. Релевантный текст лучше располагать как можно ближе к иллюстрациям, а не где-то внизу или на следующей странице. Это почти вдвое улучшает способность ученика решать соответствующие задачи.

Разговорный стиль и импровизированные диалоги. Согласно последним исследованиям ученики на 40% лучше усваивают информацию, когда автор обращается напрямую к читателю от первого лица, используя разговорный стиль вместо формальных описаний и рассказывая истории вместо того, чтобы читать лекции. Язык должен быть простым, не стоит все воспринимать слишком серьезно. К кому вы больше прислушаетесь: к другу, популярно объясняющему, как все работает, или лектору, читающему конспект перед большой аудиторией?



Развитие навыков мышления. Если вы не научитесь включать мозги, то знаний в голове не прибавится. Читатель должен быть мотивирован, пытлив и вовлечен в процесс обучения, ему должно быть интересно решать задачи, находить ответы и приобретать новые знания. Для этого нужны головоломки, упражнения и каверзные вопросы — все то, что заставляет работать оба полушария мозга.

Привлечение (и удержание) внимания читателя. Все мы сталкивались с ситуацией, когда книгу хочется прочитать, но после первой же страницы мы начинаем засыпать от скуки. Мозг реагирует на все, что нестандартно, непривычно, странно, притягательно, неожиданно. Изучение сложных технических тем не обязательно должно быть скучным. Мозг будет быстрее учиться в нетипичной обстановке.

Эмоциональная вовлеченность. Мы знаем, что способность к запоминанию сильно зависит от эмоционального состояния. Вы помните то, что для вас важно, то, что вызывает *переживания*. Нет, речь не идет о душераздирающих историях про собак и их преданность хозяевам. Мы говорим о таких эмоциях, как удивление, любопытство и смех, неожиданных реакциях ("Что за ерунда?") и возгласах "Ух ты!", когда удается справиться с головоломкой или найти решение задачи, которую все вокруг считали слишком сложной.

Метапознание: осознание знаний

Если вы хотите учиться, причем учиться быстро, получая фундаментальные знания, то следует уделять внимание тому, как мы концентрируем внимание на главном. Необходимо думать о том, как мы думаем, и изучать то, как мы учимся.

Никто из нас не изучал метакогнитивные процессы и теорию обучения, когда учился в школе. Учителя *ожидали*, что мы будем учиться, но не объясняли нам, как правильно учить предметы.

Раз вы держите в руках эту книгу, значит, хотите научиться программировать, но наверняка не заинтересованы в том, чтобы тратить на это все свободное время. Вы стремитесь *запомнить* прочитанное и применить полученные знания на практике. Для этого вы должны *понять* то, что прочитали. Чтобы извлечь максимум из книги и вообще из любого обучающего процесса, необходимо настроить свой мозг на обучение.

Трудность состоит в том, чтобы убедить мозг рассматривать изучаемый материал как *особо важный*. Жизненно важный. Такой же важный, как тигр, выпрыгивающий из кустов. В противном случае вам придется постоянно сражаться за то, чтобы не дать мозгу выкинуть из головы новую информацию как нестоящую запоминания.

Как же убедить мозг в том, что программирование не менее важно, чем тигр в кустах?

Есть медленный, утомительный путь и быстрый, более эффективный. Медленный заключается в регулярном повторении. Нам по силам изучить и запомнить даже самую скучную тему, если постоянно ее повторять. В какой-то момент мозг подумает: "Не вижу тут ничего важного, но он повторяет это *снова и снова*, наверное, для него это все-таки важно".

Быстрый способ заключается в ***повышении активности мозга*** и сочетании разных ее видов. Доказано, что все принципы, перечисленные на предыдущей странице, помогают мозгу запоминать информацию, повышая его активность. Согласно исследованиям включение поясняющего текста в иллюстрации (вместо того чтобы располагать его в пределах страницы) заставляет мозг искать связь между словами и рисунками, вовлекая в процесс больше нейронов. А чем больше активных нейронов, тем выше шансы, что мозг посчитает данную информацию стоящей внимания и запоминания.

Разговорный стиль изложения полезен, поскольку люди становятся более внимательными, когда чувствуют, что вовлечены в диалог и должны следить за ходом мысли. И что самое поразительное, мозг совершенно не заботит тот факт, что "разговор" происходит между вами и книгой. С другой стороны, когда стиль изложения сухой и формальный, мозг воспринимает это так, будто вы на лекции в аудитории. Значит, можно пересесть на заднюю парту и поспать.

Впрочем, наглядные иллюстрации и разговорный стиль — это далеко не все.

Вот что сделали мы

Мы задействовали множество ***иллюстраций***, поскольку мозгу легче воспринимать образы, чем текст. Не зря говорят, что одна картинка стоит тысячи слов. Точнее, не 1000, а 1024. Мы старались сочетать текст с изображениями, так как известно, что мозг работает эффективнее, когда текст включен в иллюстрацию, к которой относится, а не расположен отдельно от нее.

Мы нередко применяли ***избыточность***, высказывая одну и ту же мысль по-разному и разными способами. Это повышает шансы на то, что информация в конечном итоге будет запечатлена в разных участках вашего мозга.

Мы старались подавать концепции и рисунки в ***неожиданной*** манере, ведь мозг восприимчив ко всему новому. Кроме того, мы стремились создавать определенный ***эмоциональный фон***, поскольку мозг чутко реагирует на биохимию эмоций. Нам легче запоминать то, что вызывает *сопереживание*, даже если это всего лишь ***юмор***, ***удивление*** или ***заинтересованность***.

Учимся понимать Python

Мы использовали ***разговорный стиль*** изложения от первого лица, так как мозг более сосредоточен во время беседы, чем когда вы просто слушаете лекцию. Это происходит даже в процессе *чтения* книги.

Мы включили в книгу более 120 ***упражнений***, поскольку мозгу легче обучаться и запоминать информацию, когда вы что-то ***делаете***, а не просто *читаете* текст. Все задачи непростые, но решаемые — такой формат наиболее привычен.

САМОЕ ГЛАВНОЕ

Мы применяли ***разные методики обучения***: кому-то нравятся пошаговые инструкции, кто-то предпочитает сначала понять картину в целом, а кому-то достаточно увидеть пример кода. Но независимо от предпочтений читатели только выиграют, если научатся анализировать одни и те же задачи под разными углами.

Кроссворды

Мы постарались задействовать ***оба полушария*** вашего мозга, ведь чем большая часть мозга вовлечена в процесс обучения, тем выше вероятность усвоения и запоминания информации и тем дольше сохраняется концентрация внимания. Зачастую, когда одно из полушарий работает, другое отдыхает. Это позволяет повысить продуктивность обучения в течение более длительного времени.

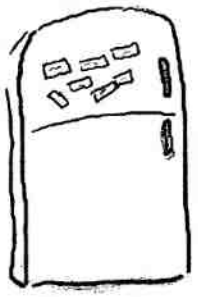
Мы добавили в книгу ***диалоги*** и упражнения, позволяющие взглянуть на проблему с ***разных точек зрения***. Наш мозг начинает глубже вникать в проблему, когда его заставляют давать оценки и составлять суждения.

Они будут учиться с нами

Мы предложили читателям ряд ***открытых задач*** в виде упражнений и ***вопросов***, на которые не всегда можно дать однозначный ответ. Наш мозг настроен на обучение и запоминание, когда ему приходится *работать* над решением. Сами посудите: вы не сможете стать атлетом, наблюдая за тем, как тренируются другие. Но мы постарались сделать так, чтобы, упорно работая, вы продвигались к намеченной цели и чтобы ***ни один дендрит вашего мозга не был задействован зря*** в процессе рассмотрения сложных примеров или анализа насыщенного техническими терминами текста.

Мы создали ***персонажей***. Они встретятся вам в диалогах, примерах, иллюстрациях — в общем, по всей книге. Мозг уделяет больше внимания живым существам, чем неодушевленным предметам или отвлеченным понятиям.

Мы придерживались принципа ***80/20*** (известного как *принцип Парето*). Мы полагаем, что, раз уж вы решили стать программистом, то наверняка захотите прочитать и другие книги. Так что мы не пытались рассказать вам *все*. Только то, что вам *действительно* нужно.



Вот что можете сделать вы, чтобы настроить свой мозг на обучение

Итак, мы свое дело сделали. Остальное зависит от вас. Приведенные ниже советы — это лишь начальные ориентиры. Прислушайтесь к своему внутреннему голосу и постарайтесь понять, что вам подходит, а что — нет. Не бойтесь искать новые пути!

Можете вырезать и приклеить на дверцу холодильника

1. **Не торопитесь. Чем больше вы поймете, тем меньше придется зубрить.**

   Не нужно просто *читать*. Делайте паузы и анализируйте прочитанное. Если вам задают вопрос, не спешите подсматривать ответ. Представьте, будто кто-то из друзей спрашивает вашего совета. Чем глубже вы пытаетесь вникнуть в суть вопроса, тем выше шансы понять и запомнить ответ.

2. **Выполняйте упражнения и делайте заметки.**

   Упражнения придумали мы, но выполнять их вам. И помните: они в книге не для того, чтобы вы просто прочитали ответ. ***Возьмите карандаш*** и запишите свое решение. Доказано, что физическая активность в процессе обучения улучшает усвоение информации.

3. **Читайте врезки "Не бойтесь задавать вопросы".**

   Все они важны. В данной книге это ***основной материал***! Их нельзя пропускать.

4. **Не читайте на ночь другие книги. Не нагружайте мозг.**

   Основная часть обучения (та, что связана с запоминанием) происходит *после* того, как вы закрываете книгу. Мозгу требуется время на то, чтобы все переварить. Если в этот промежуток времени вы нагрузите его еще чем-то, часть прочитанного будет успешно забыта.

5. **Пейте воду. Много воды.**

   Мозг на 90% состоит из воды. Обезвоживание (которое может случиться еще до того, как вы почувствуете жажду) снижает когнитивные способности человека.

6. **Обсуждайте прочитанное. Думайте вслух.**

   Речь стимулирует активность различных участков мозга. Если вы пытаетесь понять и запомнить прочитанное, проговаривайте материал вслух. А еще лучше, попробуйте объяснить его кому-то другому. Это ускорит обучение, и к тому же так можно открыть для себя малозаметные детали, на которые вы не обратили внимания в процессе чтения книги.

7. **Прислушивайтесь к внутреннему голосу.**

   Следите за тем, чтобы не перегружать мозг. Если чувствуете, что теряете нить изложения или начинаете забывать прочитанное, значит, пора сделать перерыв. После определенного момента в голову уже ничего не лезет, сколько ни запихивай. Только хуже сделаете.

8. ***Дайте волю эмоциям!***

   Мозг должен понимать, что подаваемая ему информация *важна для вас*. Придумывайте истории персонажей. Делайте шутливые подписи к иллюстрациям. Лучше поморщиться от плохой шутки, чем не испытать никаких эмоций.

9. ***Пишите программы.***

   Примените полученные знания в новом проекте или вернитесь к старому проекту и переработайте его. Постарайтесь получить какой-то опыт помимо упражнений и примеров из книги. Все, что вам нужно, — карандаш и задача, которую можно решить средствами программирования.

10. ***Старайтесь высыпаться.***

    Чтобы научиться программировать, необходимо создать множество новых связей между нейронами мозга. Это происходит во время сна. Так что утро вечера мудренее!

Это важно

Перед вами обучающая книга, а не справочник. Мы намеренно выкинули все, что может помешать изучению тех или иных тем. И если вы впервые взяли книгу в руки, то рекомендуем читать с самого начала, поскольку в книге предполагается, что вы знакомы с предыдущим материалом.

Мы хотим, чтобы вы научились думать как программист.

Кто-то считает программирование компьютерной наукой, но откроем маленький секрет: это вообще не наука, да и связь с компьютерами условна (они связаны не сильнее, чем астрономия с телескопами). Это в большей степени способ мышления, называемый в наши дни *вычислительным мышлением*. Научившись мыслить подобным образом, вы сможете применять имеющиеся знания к любым задачам, системам и языкам программирования.

В книге мы используем язык Python.

Невозможно научиться водить автомобиль, не сев за руль. Точно так же невозможно обучиться вычислительному мышлению, не освоив язык программирования. Без практики такая теория бесполезна. Поэтому в книге мы остановили свой выбор на чрезвычайно популярном языке программирования Python. Его достоинства будут подробно описаны в главе 1, а пока вам достаточно знать, что он прекрасно подойдет как начинающим, так и профессиональным программистам.

Мы не собираемся рассматривать все аспекты языка Python.

Мы даже не пытаемся этого делать, т.к. Python — слишком обширная тема. Это было бы уместно для справочника, но наша книга — не справочник, и мы не ставим перед собой цель рассказать все что только можно о Python. Наша настоящая цель — обучить вас основам программирования и вычислительного мышления, чтобы в будущем, когда вам захочется прочитать еще какую-то книгу по *любому из языков программирования*, вы понимали, о чем идет речь.

Не важно, какая у вас система: Mac, Windows или Linux.

В книге рассматривается Python — кросс-платформенный язык программирования, поддерживаемый в любой операционной системе. Большинство снимков экрана в книге было сделано в Mac, но в Windows и Linux они будут выглядеть почти так же.

Мы стараемся придерживаться практики написания хорошо структурированного и понятного кода.

Вам хочется писать программы, которые будут понятны и вам, и другим людям; программы, которые продолжат работать даже с выходом в следующем году очередной версии Python. В книге мы научим вас, как с самого начала писать понятный, четко структурированный код, которым можно гордиться и который не стыдно показать друзьям. Единственное, что отличает его от профессионального кода, — *аннотации* (а-ля рукописные), поясняющие работу программ. Мы считаем, что они намного лучше подходят для обучающей книги, чем традиционные комментарии в коде (если вы не понимаете, о чем идет речь, то скоро все поймете, буквально через пару глав). Но не переживайте: мы заодно научим вас правильно документировать код и покажем примеры составления документации к программам. Как бы там ни было, важно уметь писать программы самым простым способом, чтобы можно было как можно быстрее решить поставленную задачу и заняться чем-то более приятным.

Аннотации выглядят так

Программирование — серьезная профессия. Придется много и упорно работать.

Программисты мыслят иначе, смотрят на мир по-другому. Иногда подход к программированию будет казаться вам вполне логичным, а иногда — чересчур абстрактным или слишком запутанным. Некоторые концепции программирования требуют длительного обдумывания. Понадобится время, прежде чем вы все поймете. Главное — помните: мы стараемся все подавать в максимально дружественной манере. Просто уделите этому достаточно времени, чтобы все улеглось в голове, и не стесняйтесь перечитывать непонятные разделы, если почувствуете такую необходимость.

Упражнения обязательны.

Все приведенные в книге упражнения и задания *обязательны* для выполнения. Они являются неотъемлемой частью процесса обучения. Одни упражнения тренируют память, другие закрепляют понимание материала, третьи помогают применить полученные знания на практике. Если пропустить их, то часть материала останется непонятой (и рано или поздно вы будете сбиты с толку). Единственное, что можно смело пропустить, — это кроссворды, хотя и они помогают мозгу усвоить новую терминологию.

Повторение — мать учения.

Характерной особенностью книги является то, что мы пытаемся научить вас по-настоящему понимать программирование. Мы хотим, чтобы, прочитав книгу, вы помнили, чему научились. В большинстве справочных руководств не ставится цель добиться запоминания и усвоения материала, но наша книга — обучающая, поэтому время от времени некоторые концепции будут упоминаться повторно.

Краткость — сестра таланта.

Читатели часто пишут нам о том, как это утомительно — продираться сквозь 200 строк кода примера ради тех двух строчек, которые нужно изучить. Большинство примеров в книге подано максимально компактно, чтобы все было просто и понятно. Не ждите от примеров чудес — они составлены в целях обучения и не всегда полнофункциональны. Но мы постарались сделать их забавными и необычными, чтобы их было интересно показать друзьям или родственникам.

Не ко всем упражнениям и заданиям есть ответы.

Некоторые из заданий, приведенных во врезках "Сила мысли", не имеют однозначного решения, а некоторые составлены таким образом, что вы сами должны решить, является ли ваш ответ правильным. В некоторых упражнениях есть подсказки, которые направят вас в нужную сторону.

Скачайте файлы примеров.

Все рассматриваемые в книге примеры доступны по следующему адресу:

`http://wickedlysmart.com/hflearntocode`

Кроме того, файлы примеров доступны также на сайте издательства "Диалектика":

`http://www.williamspublishing.com/Books/978-5-907144-98-9.html`

У нас нет горячей линии, но все примеры и рабочие файлы можно скачать на сайте `http://wickedlysmart.com/hflearntocode`

Необходимо установить Python

Скорее всего, на вашем компьютере либо не установлен Python, либо установлен, но более старой версии. В книге мы используем Python 3, который на момент написания книги имел версию 3.6. Соответственно, вам понадобится установить версию 3.6 или более позднюю. Вот как это сделать.

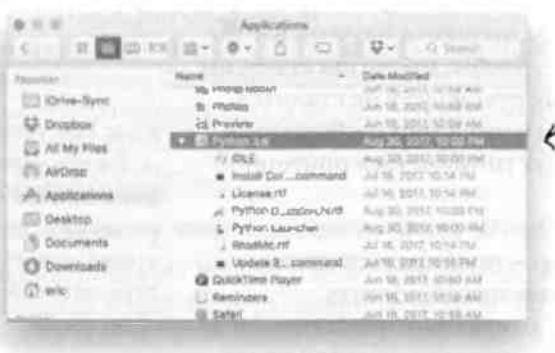
- **В macOS** откройте браузер и введите в адресной строке следующее:

  `https://www.python.org/downloads`

Появится страница, где содержится ссылка для загрузки Python в macOS. Если не видите ее, откройте меню Downloads.

1. Выберите релиз Python 3.x (где x – номер новейшей версии языка). Не загружайте Python 2.7.
2. После скачивания инсталлятора откройте установочный пакет в папке загрузки и следуйте появляющимся на экране инструкциям.
3. По окончании установки перейдите в папку Applications, в которой вы найдете папку Python 3.x. Чтобы протестировать дистрибутив, выполните двойной щелчок на приложении IDLE в этой папке.

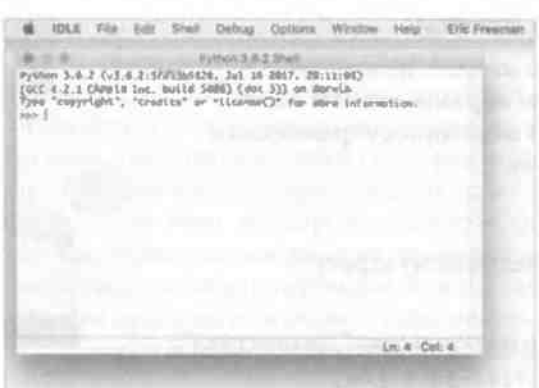
Учтите, что для установки Python нужно обладать правами администратора. Если вы регулярно устанавливаете программы, то беспокоиться не о чем. В противном случае попросите администратора помочь вам



Приложение IDLE находится в папке Python 3.x, которая сама находится в папке Applications. О том, что такое IDLE, мы поговорим в главе 1

Желательно закрепить значок IDLE на панели задач, поскольку мы будем много работать с данным приложением в книге

4. После запуска IDLE на экране появится окно, подобное показанному ниже. Если окна нет, проверьте, не было ли сообщений об ошибках в ходе установки.



Чтобы выйти из приложения, воспользуйтесь командой IDLE > Закрыть IDLE

- **В Windows** откройте браузер и введите в адресной строке следующее:

  `https://www.python.org/downloads`

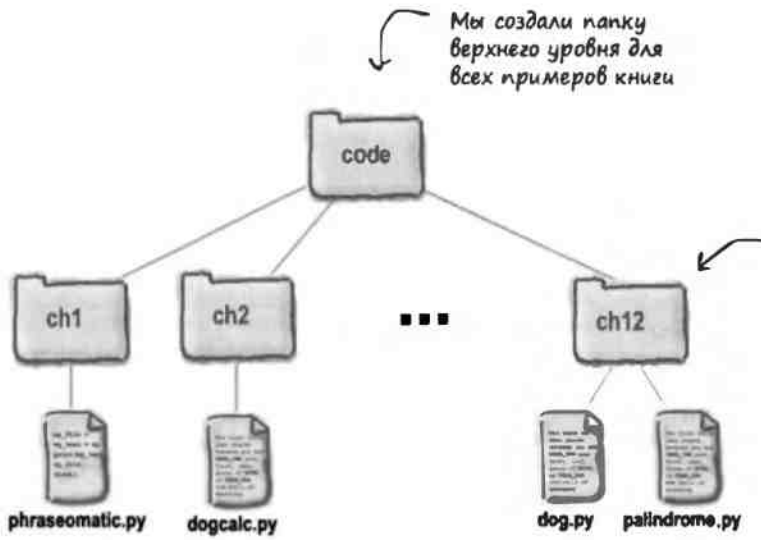
1. Выберите релиз Python 3.x (где x – номер новейшей версии языка). Не загружайте Python 2.7.
2. Запустите инсталлятор либо сохраните его на диске, а затем дважды щелкните на нем для запуска.
3. Когда на экране появится окно инсталлятора, обязательно установите флажок Add Python 3.x to PATH внизу, а затем выберите вариант Install Now.
4. По окончании установки в меню Пуск > Все программы должно появиться подменю Python 3.x (где x – номер установленной версии языка). В нем содержатся команды запуска Python, справочной службы и IDLE – редактора кода, с которым мы будем работать в книге.
5. Чтобы протестировать дистрибутив, выберите пункт IDLE. На экране должно появиться окно, подобное показанному ниже. Если окна нет, проверьте, не было ли сообщений об ошибках в ходе установки.

Чтобы выйти из приложения, воспользуйтесь командой IDLE > Закрыть IDLE

Примечание для пользователей Linux: за вас мы совершенно спокойны, ведь вы всегда знаете, что делать. Просто скачайте соответствующий дистрибутив с сайта python.org.

Как работать с исходными кодами

Исходный код — это любой файл, создаваемый вами в процессе изучения книги. Мы рекомендуем сохранять файлы в отдельных папках, каждая из которых соответствует конкретной главе. В книге предполагается, что для каждой главы создана своя папка с исходными кодами.



Учтите, что примеры в книге были адаптированы для русскоязычного издания. В архиве находятся исходные англоязычные версии программ

Файлы всех рассматриваемых в книге примеров можно скачать по следующему адресу:

`http://wickedlysmart.com/hflearntocode`

На этой веб-странице вы найдете ссылку для скачивания архива примеров. В архиве содержатся авторские версии программ, которые вам предстоит написать, а также вспомогательные файлы данных и применяемые изображения.

Мы рассчитываем, что вы будете набирать все создаваемые программы самостоятельно, шаг за шагом, по мере чтения книги. Это помогает развивать мышечную память, формируя навыки программирования, и способствует лучшему усвоению материала. Но если вы вдруг столкнетесь с проблемами, причина которых не ясна, вы всегда сможете сравнить свой код с нашим, чтобы понять, где была допущена ошибка.

Кроме того, файлы примеров доступны также на сайте издательства "Диалектика":

`http://www.williamspublishing.com/Books/978-5-907144-98-9.html`

Несколько примеров в этом архиве (главы 6 и 7) подверглись незначительным модификациям, чтобы программы могли работать так, как описано в книге.

Благодарности*

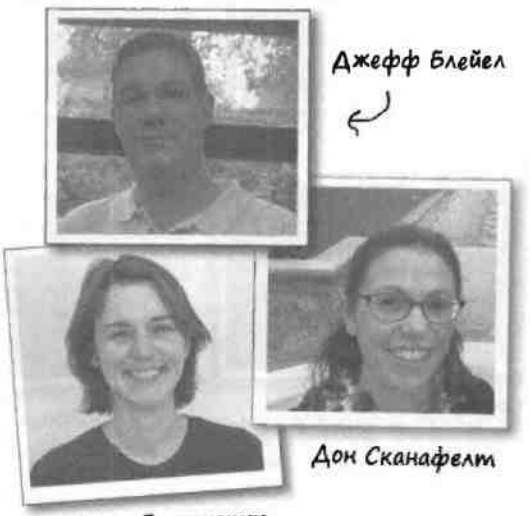
Элизабет Робсон Джош Шарфман Дейвид Пауэрс Пол Барри

В первую очередь хочу сердечно поблагодарить моих технических рецензентов. **Элизабет Робсон** внимательно просмотрела черновик книги, и ни одна смысловая ошибка не укрылась от ее зоркого глаза. **Джош Шарфман** был нашим главным рецензентом, давшим массу полезных советов и рекомендаций по каждой главе. **Дейвид Пауэрс** в своем привычном стиле тщательно проверил все главы на предмет технических ошибок. Ветеран серии Head First **Пол Барри** взял на себя важную миссию по проверке листингов Python. Кроме того, неоценимый вклад в проект внесла команда рецензентов (все они перечислены на следующей странице), которые проверили каждую страницу книги.

Огромная благодарность моим редакторам **Джеффу Блейелу, Дон Сканафелт** и **Меган Бланшетт**. Меган серьезно помогла мне на начальном этапе, Дон контролировала ход работы, а Джефф довел проект до отправки в типографию.

Отдельное спасибо всему коллективу издательства O'Reilly, в особенности **Сьюзан Конант, Рейчел Румелиотис** и **Мелани Ярбро**. Благодарю **Джейми Бертон** из компании WickedlySmart за помощь в подготовке книги, включая сбор отзывов от читателей и управление форумом рецензентов. И как всегда, спасибо **Берту Бейтсу** и **Кэти Сиерра** за вдохновение, увлекательные дискуссии и помощь в решении возникающих проблем. Кроме того, хочется поблагодарить **Кори Доктороу** за поддержку и за то, что предоставил свою книгу в качестве материала для главы 7.

В завершение хочу упомянуть тех людей, которые, сами того не ведая, послужили для меня источником вдохновения: это профессор **Даниэль П. Фридман, Нейтан Берджи**, а также разработчики из **Raspberry Pi Foundation** и **Socratica**.



Джефф Блейел

Дон Сканафелт

Меган Бланшетт

Джейми Бертон

Берт Бейтс

Кэти Сиерра

* Столь большой список благодарностей объясняется тем, что мы решили проверить теорию, согласно которой каждый, кто упомянут на странице благодарностей, купит хотя бы один экземпляр книги, а возможно и больше, ведь у него есть родственники, члены семьи, друзья, которым захочется иметь свой экземпляр. Так что если хотите быть упомянутым в нашей следующей книге и у вас большая семья, пишите, не стесняйтесь.

Команда рецензентов

Рецензированием книги занималась целая команда потрясающих людей самой разной квалификации — от **новичков** до **экспертов** — и самых разных профессий, включая **архитектора, стоматолога, учителя младших классов, агента по недвижимости** и **преподавателя информатики в школе**. Причем все они были из разных уголков земного шара: от **Албании** до **Австралии**, от **Кении** до **Косово**, от **Нидерландов** и **Нигерии** до **Новой Зеландии**.

Рецензенты вычитали каждую из более чем 600 страниц книги, выполнили все упражнения и протестировали каждую строку кода, высказав свои замечания. Они сформировали настоящую команду, помогая друг другу в прояснении непонятных моментов и перепроверяя найденные ошибки.

Каждый из рецензентов внес важный вклад в книгу и помог сделать ее лучше.

Спасибо вам всем!

Кристал Гилмор Родригес

Ридван Бунджаку

Трой Уэлч

Хадсон Рид

Марк ван дер Линден

Митч Джонсон

Крис Талент

Андреа Тостон

Крис Григгз

Дейвид Опаранти

Джонни Ривера

Дейвид Киноти

Майкл Пек

Мауро Кейзер

Бенджамин И. Холл

Альфред Дж. Спеллер

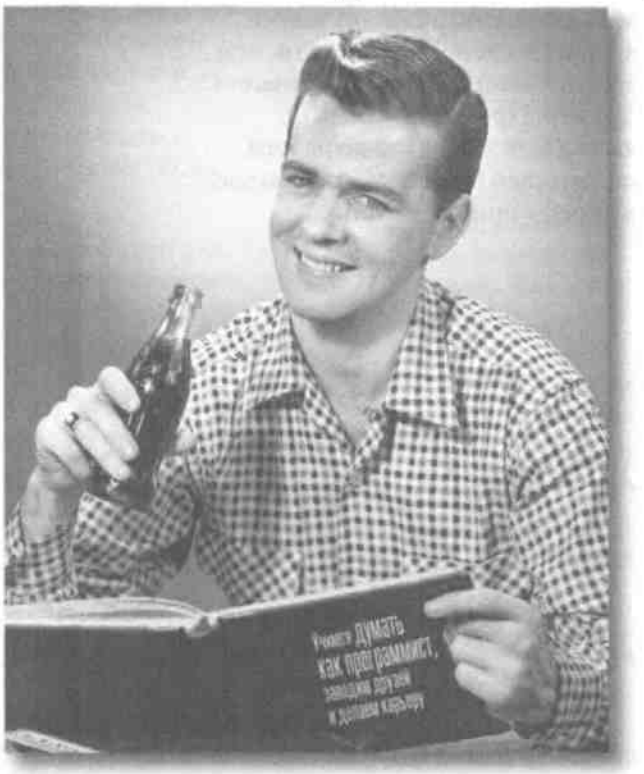
Тайрон Андрич

Деннис Фицджеральд

Абдул Рахман Шаик

1 учимся думать как программист

Приступим



Чтобы научиться писать программы, нужно думать как программист. В мире современных технологий все вокруг нас становится взаимосвязанным, настраиваемым, программируемым и в каком-то смысле **компьютерным**. Можно оставаться пассивным наблюдателем, а можно *научиться программировать*. Умея писать программный код, вы становитесь творцом, устанавливающим правила игры, ведь именно вы отдаете команды компьютеру. Но как освоить искусство программирования? Самое главное — это начать **думать как программист**. Далее нужно определиться с **языком программирования** — написанные на нем программы должны выполняться на всех целевых устройствах (компьютерах, смартфонах и любых других электронных гаджетах, снабженных процессором). Что это вам даст? Больше свободного времени, больше ресурсов и больше творческих возможностей для воплощения в жизнь задуманного. Итак, давайте приступим...

Шаг за шагом

Первое, что необходимо сделать, чтобы начать писать собственный код, — научиться разбивать стоящие перед вами задачи на небольшие действия, которые компьютер сможет *выполнить за вас*. Для этого вы и компьютер должны использовать понятный друг другу язык, но об этом мы вскоре поговорим.

Несмотря на кажущуюся сложность, разбивка задачи на последовательность простых действий — это то, чем вы занимаетесь каждый день, иногда даже неосознанно. Рассмотрим простой пример. Предположим, вы хотите описать процесс рыбной ловли в виде набора инструкций для робота, который будет рыбачить за вас. Вот что мы получаем в первом приближении.

Давайте разобьем процесс рыбной ловли на несколько простых и понятных этапов

Мы проходим этапы по очереди

1. Наживить червя на крючок.
2. Закинуть приманку в пруд.
3. Следить за поплавком, пока он не дернется.
4. Выудить рыбку.
5. Если рыбалка окончена, отправиться домой; иначе вернуться к п. 1.

Некоторые этапы представляют собой простые инструкции

Некоторые инструкции предполагают ожидание, пока не наступит определенное условие

Эта инструкция будет выполнена только после того, как поплавок дернется на предыдущем этапе

В инструкциях могут приниматься решения, например о том, не пора ли домой или можно еще порыбачить

Инструкции часто повторяются, как в данном случае: если мы не уходим домой, то возвращаемся в начало и повторяем инструкции, чтобы поймать очередную рыбку

Поплавок

Крючок

Червяк

Приведенные выше инструкции представляют собой **рецепт** рыбной ловли. Как и в любом другом рецепте, последовательное выполнение приведенных в нем инструкций приводит к получению желаемого результата (предположительно будет поймана рыба).

Обратите внимание на то, что на каждом этапе выполняется всего одна простая инструкция, например "наживить червя на крючок" или "выудить рыбку". Также заметьте, что некоторые операции становятся возможными только при соблюдении определенных условий, например "следить за поплавком, пока он не дернется". В инструкциях также может описываться порядок действий: "Если рыбалка окончена, отправиться домой".

Подобные простые инструкции служат основой для написания программного кода. Любое приложение и любая программа содержат именно такой (разве что немного больший по размеру) набор инструкций для компьютера.

УПРАЖНЕНИЕ

В рецептах блюд содержатся не только инструкции по их приготовлению, но и перечисляются все необходимые ингредиенты (в программировании мы называем их *объектами*). Какие объекты используются в рецепте рыбной ловли? Обведите все объекты в списке инструкций на предыдущей странице и сверьтесь с ответом, приведенным в конце главы.

Считайте книгу рабочей тетрадью и не стесняйтесь писать прямо в ней; мы только за

Возьмите карандаш

Прежде всего вы должны понять, что компьютеры выполняют **в точности** те команды, которые вы им передаете, — ни больше ни меньше. Рассмотрим рецепт рыбной ловли, приведенный на предыдущей странице. Если робот будет следовать ему буквально, то с какими трудностями он может столкнуться? Насколько успешной будет такая рыбалка?

- ☐ А. Если в пруду нет рыбы, то вы будете ее ловить очень долго (т.е. вечно).
- ☐ Б. Если червь сорвется с крючка, то вы не узнаете об этом и не сможете его заменить.
- ☐ В. Что, если у нас закончатся черви?
- ☐ Г. Что делать с рыбой после того, как мы ее поймали?
- ☐ Д. А что делать с удочкой?
- ☐ Е. _____

Знаете другие проблемы?

Это поплавок; когда рыба клюет, он уходит под воду

Ответы на подобные упражнения приводятся в конце главы

Я, вообще-то, заплатила кучу денег за книгу по программированию, а вы мне даете какие-то рецепты! Я ничего не перепутала?

По правде говоря, рецепт — отличный способ описать набор инструкций для компьютера. Этот термин можно встретить и в более сложных книгах по программированию, некоторые из них даже так и называются: "Сборник рецептов". Но если вам по душе технический жаргон, то не проблема: компьютерные ученые и профессиональные разработчики используют более привычный термин **"алгоритм"**. Что такое алгоритм? Да, собственно, то же самое, что и рецепт: список инструкций для решения той или иной задачи. Зачастую алгоритмы записывают в виде текстовых команд, называемых **псевдокодом**.

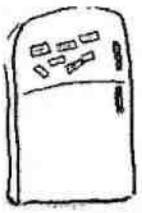
Об этом мы вскоре поговорим

Помните о том, что, говоря о рецепте, псевдокоде или алгоритме, мы будем подразумевать общее описание процесса решения той или иной задачи, которое предшествует собственно программированию на языке, понятном компьютеру.

Подобный подход упрощает процесс кодирования, снижая число ошибок

В книге вам встретятся все три термина, которые используются взаимозаменяемо. Когда в следующий раз пойдете на собеседование, попробуйте удивить потенциального работодателя знанием термина "алгоритм" или "псевдокод". Глядишь, вам предложат оклад повыше (хотя и термина "рецепт" не стоит бояться).

Подобно тому как существует множество рецептов приготовления одного и того же блюда, любая задача по программированию имеет множество алгоритмов решения



Код на магнитиках

Давайте поупражняемся в составлении алгоритмов. У нас был отличный рецепт приготовления омлета из трех яиц, записанный с помощью набора магнитиков на дверце холодильника. Вот только незадача: кто-то пришел и все перемешал. Сможете расположить магнитики в правильном порядке, чтобы восстановить алгоритм? Учтите, что наш рецепт описывает два вида омлета: обычный и с сыром. **Не забудьте свериться с ответом, приведенным в конце главы.**

Расположите магнитики в нужном порядке, чтобы получить алгоритм

Если клиент заказал сыр:

Добавить сверху сыр

Пока все не перемешается:

Выложить яичницу на тарелку

Пока яичница не поджарится:

Снять сковороду с плиты

Перемешивать

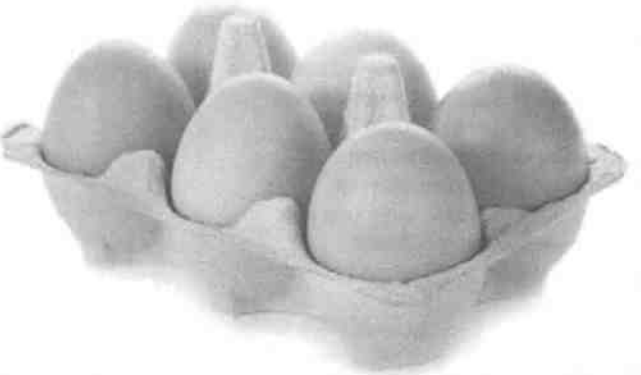
Подать

Нагреть сковороду

Разбить яйца

Разбить три яйца в миску

Вылить яйца на сковороду



Как пишут программы

Предположим, перед нами стоит задача, которую мы хотим решить с помощью компьютера. Мы знаем, что ее нужно разбить на последовательность инструкций, понятных компьютеру, но как их *передать компьютеру*? Здесь нам и пригодится язык программирования, команды которого будут понятны *как вам, так и компьютеру*! Но прежде чем приступать к знакомству с языком программирования, опишем общую последовательность действий.

1 Составьте алгоритм.

Проанализируйте стоящую перед вами задачу и составьте высокоуровневый рецепт, или алгоритм, ее решения, перечислив все действия, которые должен выполнить компьютер.

1. Наживить червя на крючок.
2. Закинуть приманку в пруд.
3. Следить за поплавком, пока он не дернется.
4. Выудить рыбку.
5. Если рыбалка окончена, отправиться домой; иначе вернуться к п. 1.

На этом этапе мы продумываем порядок действий, прежде чем писать алгоритм на языке программирования

2 Напишите программу.

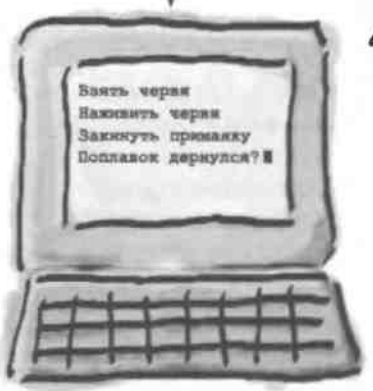
Переведите составленный на предыдущем этапе рецепт в набор инструкций, записав их на выбранном языке программирования. Это этап *кодирования*, результатом которого будет *программа*, или просто *код* (формально — *исходный код*).

```
def hook_fish():
    print('Поймал рыбку!')

def wait():
    print('Ждем...')

print('Взять червя')
print('Наживить червя')
print('Закинуть приманку')

while True:
    response = input('Поплавок дернулся? ')
    if response == 'да':
        is_moving = True
        print('Ккккуго!')
        hook_fish()
    else:
        wait()
```

Это этап кодирования, на котором мы преобразуем алгоритм в программный код, выполняемый на следующем этапе

3 Выполните программу.

Наконец, исходный код программы передается компьютеру, который начинает выполнять заданные инструкции. В зависимости от выбранного языка программирования этот этап может называться *интерпретацией*, *запуском* или *выполнением* программы.



Когда программный код написан, его можно выполнить, и если все прошло успешно и код правильный, то вы получите от компьютера тот результат, который ожидаете

В дальнейшем будем считать эти термины взаимозаменяемыми

Понимают ли нас компьютеры?

Любой **язык программирования** специально разрабатывается для решения на компьютере определенного круга задач. Языки программирования позволяют преобразовать набор команд в понятные компьютеру инструкции.

Чтобы изучить язык программирования, необходимо понять две вещи: какие конструкции в нем поддерживаются и что они означают. Компьютерные ученые называют это **синтаксисом** и **семантикой языка**. Пока что просто запомните оба термина; их смысл станет вам ясен позднее.

Подобно языкам человеческого общения, существует *множество* языков программирования. Как вы уже поняли, в книге мы будем практиковаться в написании программ на языке Python. Нам предстоит понять, чем он отличается от других языков.

Выдохни

Не волнуйтесь, от вас пока не требуется ни писать, ни анализировать программный код. Впереди еще вся книга, а сейчас мы лишь знакомимся с процессом программирования. В конце концов, это вводная глава.

Методики, с которыми вам предстоит познакомиться в книге, применимы к любым языкам программирования, не только к Python.

Фразы одного сорта

Слева записан ряд инструкций на понятном для вас языке, а справа — они же на языке программирования. Соедините линией каждое из предложений с соответствующей инструкцией программного кода. Первую линию мы нарисовали за вас. Сверьтесь с ответом в конце главы.

Напечатай "Привет!" на экране.

Если температура выше, чем 27 °C, напечатай "Надень шорты" на экране.

Создай список продуктов, включающий хлеб, молоко и яйца.

Налей пять напитков.

Спроси пользователя: "Как тебя зовут?"

```
for num in range(0, 5):
        pour_drink()

name = input('Как тебя зовут? ')

if temperature > 27:
        print('Надень шорты')

grocery_list = ['хлеб', 'молоко', 'яйца']

print('Привет!')
```

Мир языков программирования

Вы наверняка знаете о том, что существует огромное количество всевозможных языков программирования. Чтобы убедиться в этом, достаточно заглянуть в отдел компьютерной литературы любого книжного магазина, — там будут книги по Java, C, C++, C#, Objective-C, Perl, PHP, Swift, Ruby, JavaScript и, конечно же, Python. Вы спросите: откуда взялись такие странные названия? Это как с названиями рок-групп: первоначально они что-то значили для их создателей. В частности, язык Java был назван в честь популярного сорта кофе, когда выяснилось, что название Oak ("Дуб") уже занято. Название языка программирования C получено как следующее в алфавитном порядке после A и B (языки программирования, разработанные в Bell Labs). Но зачем нужно столько языков программирования? Давайте спросим у программистов.

Я использую язык **Objective-C**, поскольку пишу приложения для iPhone. Он основан на C, но при этом более динамический и объектно-ориентированный. Я также изучаю новый язык программирования, разработанный компанией Apple: Swift.

Java позволяет мне думать на уровне объектов, а не системного кода. Здесь многие низкоуровневые задачи, такие как управление памятью и программирование потоков, решаются за меня.

Я работаю в среде **WordPress**, которая написана на **PHP**, поэтому PHP — мой основной язык. Некоторые называют его языком сценариев, хотя он позволяет мне делать все необходимое.

Я в основном применяю язык **C**, поскольку разрабатываю компоненты операционных систем, которые должны функционировать максимально эффективно. Мне приходится учитывать каждый байт памяти и каждый цикл центрального процессора.

Можете считать меня ретроградом, но я люблю **Scheme** и другие языки в стиле LISP. Для меня в первую очередь важна возможность работы с функциями высшего порядка и абстракциями. Не может не радовать возрастающая популярность таких функциональных языков, как **Clojure**.

Я веб-разработчик, и мой основной язык — **JavaScript**. Он поддерживается всеми современными браузерами и часто применяется для создания серверных решений.

Как системный администратор я часто пишу скрипты на **Perl**. Они компактные, но невероятно эффективные. Всего несколько строк кода позволяют делать удивительные вещи.

Как все сложно...

Как видите, сколько языков, столько и мнений, причем мы перечислили далеко не все современные языки. С ними связано огромное количество терминов, которые со временем станут для вас более понятными. А пока что достаточно знать, что в мире современных технологий применяется множество языков программирования, и не проходит и дня, чтобы их количество не увеличивалось.

На чем же мы остановим свой выбор? Самое главное, что нам предстоит, — научиться думать *как программист*. Это позволит в будущем освоить любой язык программирования. Но с чего-то ведь нужно начать, и, как вы уже поняли, это будет Python. Почему? Как сказали только что наши друзья-программисты, он считается одним из лучших языков для новичков, поскольку позволяет писать простой и понятный код. Кроме того, это довольно мощный язык, предлагающий готовые решения (их называют *модулями* или *библиотеками*) для широкого круга задач. Его поддерживает огромное сообщество разработчиков, которые всегда готовы протянуть вам руку помощи. Наконец, мы найдем немало забавного в программировании на Python. В общем, беспроигрышный вариант!

Возьмите карандаш

Насколько легко писать код на Python

Вы еще толком не знаете Python, но наверняка сможете понять назначение многих конструкций. Проанализируйте приведенный ниже код построчно и попытайтесь угадать, что делает каждая команда. Запишите свои идеи в таблицу справа. Первый ответ мы вписали за вас. Остальные ответы даны на следующей странице.

`customers = ['Джонни', 'Ким', 'Джон', 'Стэйси']`	*Создать список клиентов.*
`winner = random.choice(customers)`	
`flavor = 'ванильный'`	
`print('Поздравляем, ' + winner +` `'! Вы выиграли десерт!')`	
`prompt = 'Хотите с вишенкой сверху? '`	
`wants_cherry = input(prompt)`	
`order = flavor + ' десерт '`	
`if (wants_cherry == 'да'):` `order = order + 'с вишенкой сверху'`	
`print('Один ' + order + ' для ' + winner +` `'. Заказ принят...')`	

Оболочка Python

```
Поздравляем, Стэйси! Вы выиграли десерт!
Хотите с вишенкой сверху? да
Один ванильный десерт с вишенкой сверху для Стэйси.
Заказ принят...
```

Это образец работы листинга вам в помощь. Будут ли результаты одними и теми же при каждом запуске программы?

Возьмите карандаш

Решение

Насколько легко писать код на Python

Вы еще толком не знаете Python, но наверняка сможете понять назначение многих конструкций. Проанализируйте приведенный ниже код построчно и попытайтесь угадать, что делает каждая команда. Запишите свои идеи в таблицу справа. Первый ответ мы вписали за вас.

`customers = ['Джонни', 'Ким', 'Джон', 'Стэйси']`	*Создать список клиентов.*
`winner = random.choice(customers)`	*Произвольным образом выбрать одного из клиентов.*
`flavor = 'ванильный'`	*Записать в переменную* `flavor` *текст "ванильный".*
`print('Поздравляем, ' + winner +` `'! Вы выиграли десерт!')`	*Вывести на экран приветственное сообщение с именем победителя. Например, если победила Ким, то программа выведет "Поздравляем, Ким! Вы выиграли десерт!"*
`prompt = 'Хотите с вишенкой сверху? '`	*Записать в переменную* `prompt` *текст "Хотите с вишенкой на сверху? "*
`wants_cherry = input(prompt)`	*Попросить пользователя ввести текст и записать этот текст в переменную* `wants_cherry`*. Обратите внимание на то, что сначала выводится текст самого приглашения.*
`order = flavor + ' десерт '`	*Объединить в заказе текст "ванильный" и " десерт ".*
`if (wants_cherry == 'да'):` `order = order + 'с вишенкой сверху'`	*Если пользователь ответил "да" на вопрос "Хотите с вишенкой сверху?", то добавить к заказу текст "с вишенкой сверху".*
`print('Один ' + order + ' для ' + winner +` `'. Заказ принят...')`	*Сообщить о том, что заказ победителя принят.*

Оболочка Python

```
Поздравляем, Стэйси! Вы выиграли десерт!
Хотите с вишенкой сверху? да
Один ванильный десерт с вишенкой сверху для Стэйси.
Заказ принят...
```

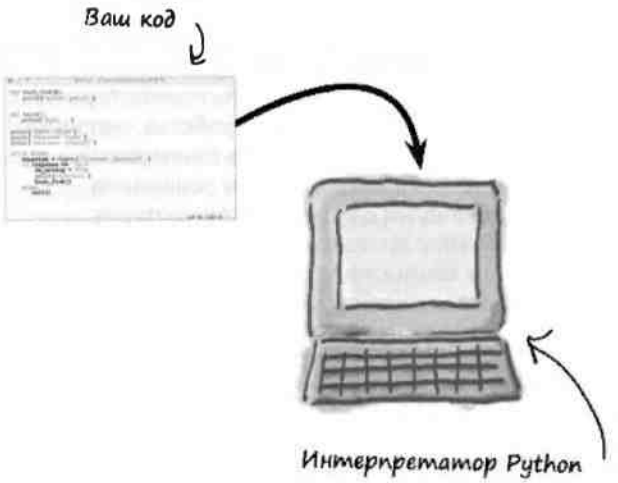
P.S. Если захотите набрать этот листинг, чтобы проверить его работу, то добавьте в самом начале строку `import random`*. О том, зачем это нужно, мы поговорим позже. Кроме того, подчеркнем, что на данном этапе от вас не требуется проверять листинги. Но любопытство никогда не бывает лишним!*

Ввод и выполнение кода Python

Раз уж дело дошло до анализа готового кода Python, пора узнать, как вводится и выполняется реальный код. Как уже говорилось, в разных языках программирования и средах разработки это делается по-разному. В случае Python ввод и запуск программного кода выполняется следующим образом.

1 Ввод программного кода.

Сначала необходимо ввести в редакторе код и сохранить его в виде отдельного файла. Для этого подойдет обычный текстовый редактор, например Блокнот в Windows или TextEdit в Mac. Тем не менее большинство разработчиков применяют специальные редакторы кода, называемые IDE (Integrated Development Environment — интегрированная среда разработки). Эти редакторы обладают широким набором возможностей, таких как автозаполнение ключевых слов, цветовое выделение синтаксиса, индикация ошибок, встроенные инструменты тестирования и т.п. Для Python тоже имеется удобная среда разработки, называемая IDLE, с которой мы вскоре познакомимся.



2 Выполнение введенного кода.

Запуск кода Python заключается в его передаче интерпретатору — программе, которая берет на себя выполнение инструкций. Вскоре мы детальнее рассмотрим этот этап, а пока что вам достаточно знать, что передать код Python интерпретатору можно из IDLE или из командной строки.

3 Интерпретация кода.

Python — язык программирования, понятный и человеку, и компьютеру. А интерпретатор — это программа, которая считывает и выполняет программные инструкции. Он преобразует инструкции в низкоуровневые команды, выполняемые аппаратным обеспечением компьютера. Вам не нужно быть в курсе того, как все это происходит, достаточно лишь знать, что интерпретатор обрабатывает каждую инструкцию Python.

Не бойтесь задавать вопросы

В: Почему нельзя программировать непосредственно на английском языке? Тогда нам не пришлось бы учить никакие дополнительные языки.

О: К сожалению, все не так просто. Английский, как и любой естественный язык, полон двусмысленностей, что делает построение интерпретатора практически невозможным. Мы все еще очень далеки от того, чтобы превратить разговорный язык в язык программирования. Кроме того, в процессе исследований выяснилось, что программисты предпочитают использовать не языки, близкие к разговорному английскому, а языки командного типа.

В: Почему нельзя довольствоваться одним языком программирования?

О: С технической точки зрения все языки программирования схожи между собой в том смысле, что позволяют решать одни и те же задачи, так что чисто теоретически мы могли бы обойтись одним языком программирования. Но, как и разговорные языки, все языки программирования обладают разными выразительными способностями, и в результате для некоторых задач (например, построение веб-сайтов) одни языки подходят лучше, чем другие. Иногда выбор языка программирования определяется личными предпочтениями программиста, методологией решения задачи или требованиями заказчика проекта.

В: Можно ли считать Python учебным языком, предназначенным только для начинающих? Если я хочу стать профессиональным разработчиком, стоит ли мне изучать его?

О: Python — серьезный язык, на нем написано большое количество популярных приложений. И в то же время это один из нескольких языков, считающихся удобными для новичков. Почему? Да потому что, в отличие от многих других языков, в нем применяются простые и однозначно трактуемые программные конструкции (со временем вы в этом убедитесь).

В: Научиться программировать и научиться думать как программист — это разные вещи?

О: Вычислительное мышление — термин из компьютерных наук, означающий умение решать задачи с помощью компьютеров. Мы должны понимать, как разбивать решение задачи на отдельные этапы, как создавать алгоритмы и обобщать их для решения более сложных задач. В тех случаях, когда мы хотим, чтобы компьютер выполнил предложенный алгоритм, мы прибегаем к программированию, которое представляет собой способ описания алгоритма для компьютера (или любого другого вычислительного устройства, например смартфона). Как видите, оба понятия взаимосвязаны. Вычислительное мышление дает нам понимание того, как решать задачи с помощью компьютеров, а программирование позволяет описать решение на языке, понятном компьютеру. Вообще говоря, навык вычислительного мышления может быть полезен не только в программировании.

Сила мысли

Как думаете, откуда появилось название Python? Выберите, на ваш взгляд, наиболее вероятный ответ.

- ☐ **А.** Автор языка Python очень любил змей и ранее создал другой, менее успешный язык Cobra.
- ☐ **Б.** Автор языка Python любил число "пи" и потому назвал свое детище "Пи-тон".
- ☐ **В.** Язык Python назван в честь британской комик-группы.
- ☐ **Г.** Python — это акроним от "Programming Your Things, Hosted On the Network".
- ☐ **Д.** Язык Python назван в честь среды Anaconda, поверх которой он работает.

В. Монту Пайтон — это британская комик-группа, создавшая комедийный скетч-сериал "Летающий цирк Монти Пайтона". Можете посмотреть их скетчи в Интернете. На них часто ссылаются в сообществе Python.

Краткая история Python

Python 1.0

Однажды в Нидерландах — в Центре математики и информатики (CWI) — возникла весьма неожиданная проблема: научные сотрудники считали языки программирования сложными для изучения. Представляете, даже маститым ученым существующие языки программирования казались запутанными и непоследовательными!
В качестве решения проблемы был разработан новый язык, который назвали ABC. (А вы думали, вот так сразу появился Python, да?) ABC достиг определенной популярности, но вскоре молодой разработчик Гвидо ван Россум, большой поклонник комик-группы "Монти Пайтон", решил, что способен на большее. Взяв за основу ABC, он создал язык Python. Все остальное — история.

Которая описана в следующих двух абзацах

Python 2.0

Со временем Python эволюционировал до версии 2.0, пополнившись новыми средствами и получив поддержку у большого количества разработчиков по всему миру. В частности, в Python 2.0 была включена возможность работы с текстами в кодировках, сильно отличающихся от английской. Также были усовершенствованы многие технические аспекты языка, в первую очередь касающиеся управления памятью и обработки стандартных типов данных (списков и строк).

Команда создателей Python постаралась сделать язык максимально открытым для сообщества разработчиков, которые были готовы вносить свои улучшения в язык.

Python 3.0

Ничто не совершенно, и спустя какое-то время создатели Python осознали, что определенные компоненты языка требуют дальнейшего усовершенствования. Python был известен своим стремлением к простоте, однако опыт программирования показал, что некоторые элементы дизайна языка могут быть улучшены, тогда как некоторые инструменты не прошли проверку временем и должны быть удалены.

Внесенные изменения означали, что некоторые средства Python 2 больше не поддерживаются. В то же время создатели Python сделали так, чтобы код, написанный на Python 2, мог продолжать выполняться. Так что, если у вас имеется старый код — не переживайте: он по-прежнему работоспособен. Просто нужно понимать, что будущее — за Python 3.

Мы ожидаем, что наш летающий автомобиль будет поддерживать Python

1994 — 2000 — 2008 — **Будущее!**

Насколько я понимаю, сейчас есть две версии Python: 2 и 3. Какая между ними разница, и какую из них мы будем использовать?

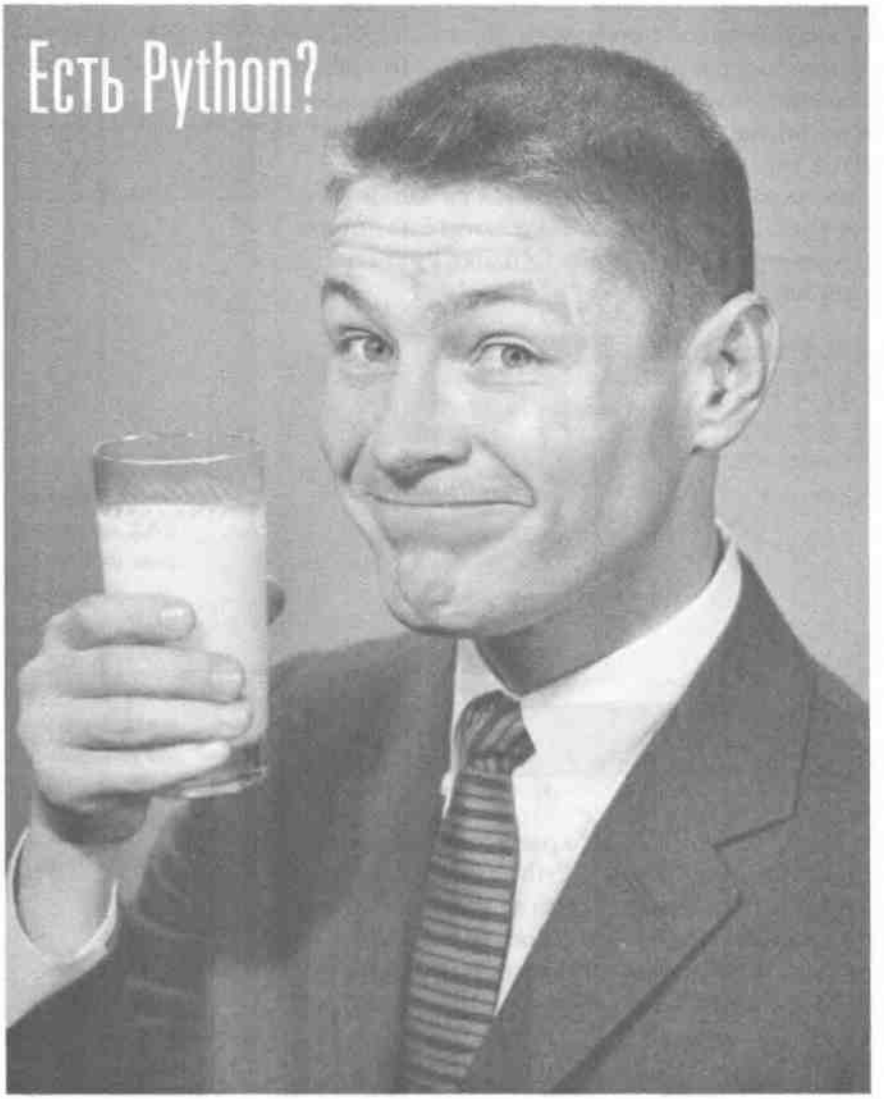
Хороший вопрос. Вы правы, в настоящий момент существуют две версии языка Python. На момент написания книги текущими были версии 3.6 и 2.7.

На момент выпуска русского издания книги основной была версия 3.8, а версия 2.7 уже не рекомендовалась к применению. Это означает, что уже никто не должен разрабатывать новый код для версии 2.7! Нужно изучать исключительно версию 3.x.

Что касается различий, то они становятся понятными только при детальном изучении языка. Для новичков обе версии выглядят почти одинаковыми. Тем не менее разница между версиями *существенна* — настолько, что программа, написанная для одной версии, иногда может не работать в другой. В книге мы будем использовать более новую версию, т.е. Python 3. Выбор вполне очевиден: обучение нужно начинать с той версии, которая будет продолжать развиваться в будущем.

Впрочем, не стоит забывать о том, что на Python 2 написано огромное количество программ, и вы вполне можете столкнуться со старым кодом, скачав какой-нибудь модуль из Интернета. Или же, если вы станете разработчиком, вам могут поручить сопровождение старого проекта, написанного на предыдущей версии Python. Прочитав книгу, вы получите достаточно знаний, чтобы разобраться в различиях между Python 2 и 3.

Говоря о Python 2 или 3, мы подразумеваем последние версии, доступные на момент написания книги (2.7 и 3.6)

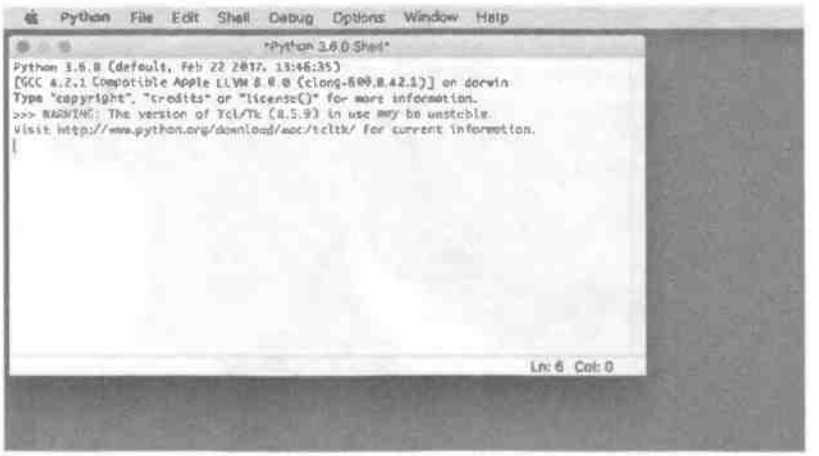


Для дальнейшего обучения вам нужно установить среду разработки Python. Если вы не сделали этого ранее, то обязательно сделайте сейчас. Детальные инструкции были приведены во введении. Если вы пользователь Mac или Linux, то вполне вероятно, что Python у вас уже установлен. С другой стороны, это может оказаться версия 2, а не 3. Следовательно, вам придется установить Python 3 самостоятельно.

После установки и запуска среды разработки Python можно наконец-то приступать к написанию программного кода.

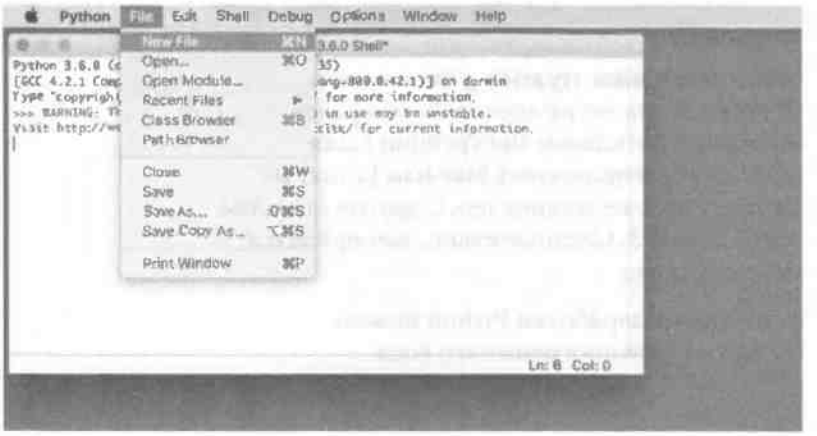
Тестируем среду

Итак, Python установлен, и можно приступать к работе. Мы начнем с элементарной тестовой программы, просто чтобы убедиться, что все настроено и работает правильно. Но прежде чем запустить программу, ее необходимо набрать в редакторе. Именно здесь нам и пригодится встроенный редактор Python – IDLE. Откройте IDLE так, как было описано во введении. Напомним, что в Mac редактор кода запускается из папки **Applications > Python 3.x**. В Windows щелкните на кнопке **Пуск**, в разделе **Все программы** перейдите к папке **Python 3.x**, а затем выберите опцию **IDLE**.



При первом запуске редактора IDLE на экране отображается окно интерактивного интерпретатора — Python Shell (оболочка Python). Если интересно, введите 1 + 1 и нажмите <Enter>. Подробнее о работе с интерпретатором мы поговорим в следующей главе

При первом запуске редактора IDLE на экране отображается окно интерактивного интерпретатора – Python Shell (оболочка Python), в котором можно непосредственно вводить инструкции. Это окно называется *консолью*. Здесь же можно набирать программный код. Для этого выполните команду **File > New File**. На экране появится новое, пустое окно встроенного редактора кода.



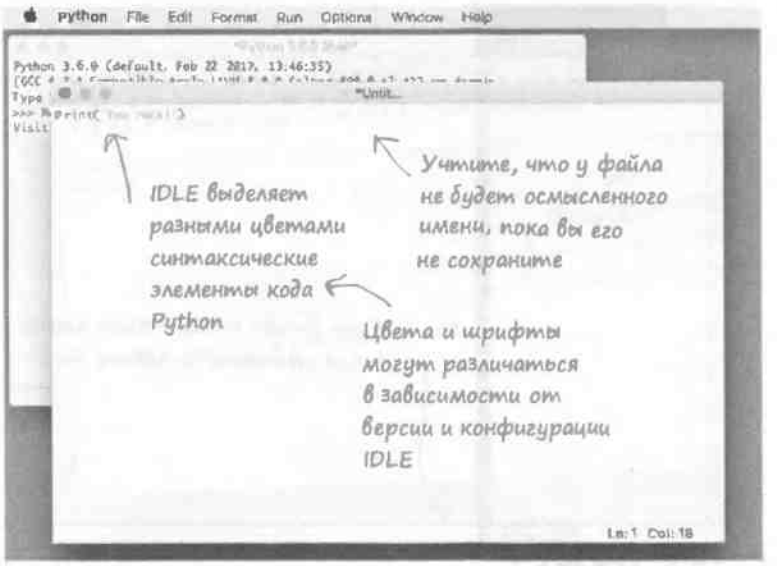
Воспользуйтесь командой File > New File, чтобы открыть новое окно для ввода кода Python

Редактор IDLE во многом напоминает текстовый процессор, за тем лишь исключением, что ему понятен синтаксис языка Python. Он автоматически выделяет разными цветами ключевые слова, помогает правильно форматировать код и по возможности предлагает свои услуги по автозаполнению программных конструкций.

После выбора команды New File появится новое пустое окно поверх окна оболочки Python

После появления на экране нового пустого окна можно начинать вводить программный код. Введите следующую инструкцию:

```
print('You rock!')
```



IDLE выделяет разными цветами синтаксические элементы кода Python

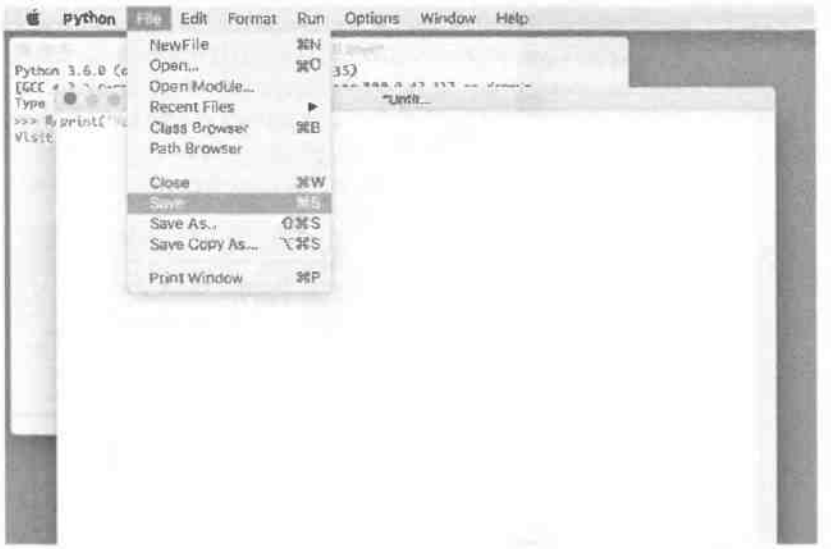
Учтите, что у файла не будет осмысленного имени, пока вы его не сохраните

Цвета и шрифты могут различаться в зависимости от версии и конфигурации IDLE

Тщательно следите за орфографией и пунктуацией в коде, поскольку Python, как и все остальные языки программирования, не терпит никаких ошибок

Сохранение кода

Завершив ввод первой строки программного кода, давайте сохраним ее. Для этого выполните команду **File > Save** в окне редактора IDLE.

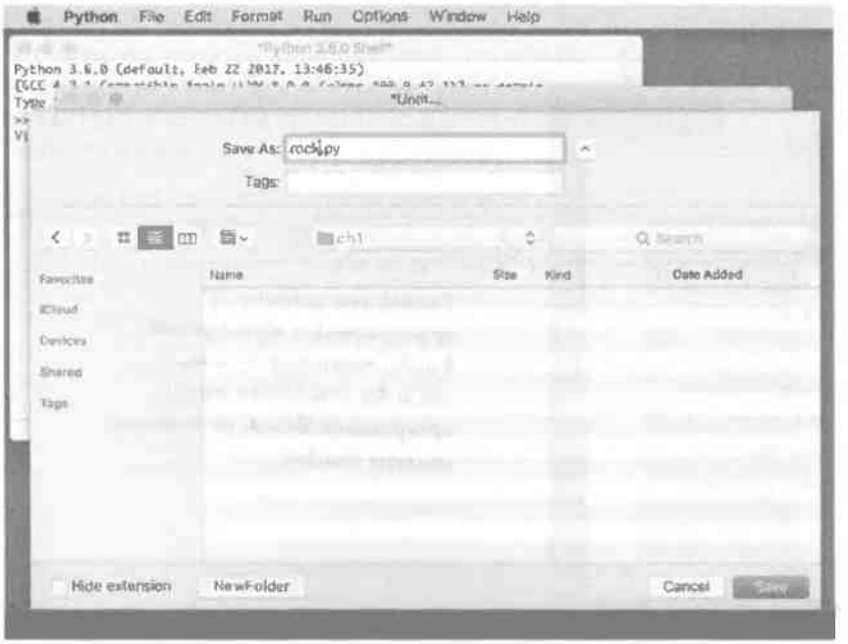


Прежде чем выполнять код, его необходимо сохранить. В IDLE команда Save находится в меню File. После вызова этой команды укажите имя, под которым будет сохранен файл. В случае кода Python необходимо добавить в конце имени расширение ".py"

Исходный код, исходный файл, код, программа — все это синонимы файлов, в которых хранится программный код

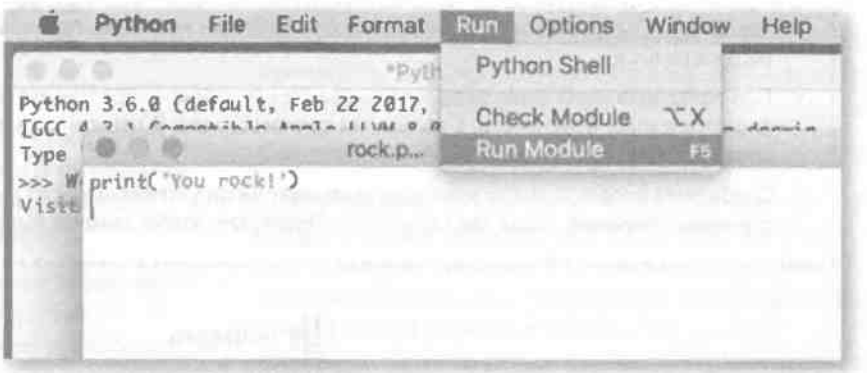
Дайте файлу имя, дополнив его расширением *.py*. В нашем случае файл называется *rock.py*. Кроме того, хоть это и не показано здесь, все файлы примеров данной главы сохраняются в специально созданной для них папке *ch1*. По возможности сделайте то же самое.

Помещайте файлы в те же папки, что и мы. Схема хранения файлов была описана во введении



После выбора папки и ввода имени файла щелкните на кнопке Save

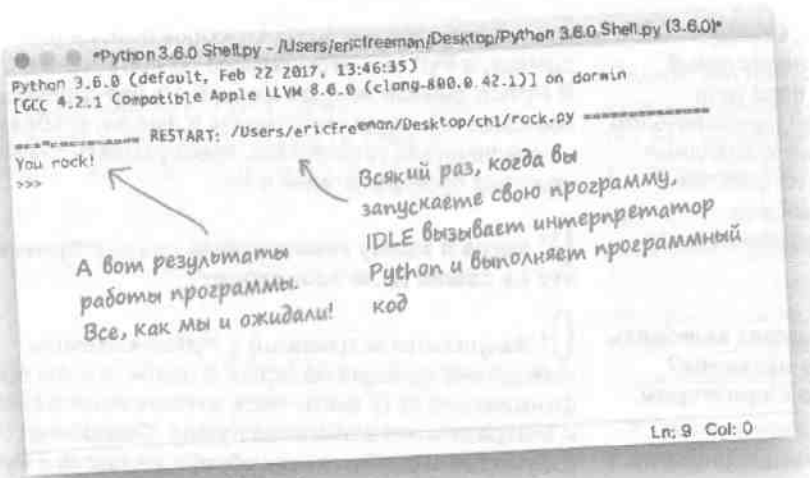
Наступил решающий момент. После сохранения кода в отдельном файле выполните команду **Run > Run Module**. Результат работы программы должен появиться в окне оболочки Python.



Чтобы запустить программу, выполните команду Run > Run Module. Если программа не была сохранена, появится соответствующий запрос

Первая программа готова!

Итак, вы установили Python, ввели в редакторе IDLE короткую инструкцию и успешно выполнили свою первую программу. Конечно, это не самый сложный код, но начало положено! Теперь мы можем переходить к более серьезным вещам.



Всякий раз, когда вы запускаете свою программу, IDLE вызывает интерпретатор Python и выполняет программный код

А вот результаты работы программы. Все, как мы и ожидали!

Получили другой результат?

Не секрет, что процесс написания программ чреват ошибками. Если ваша первая попытка оказалась неудачной, не отчаивайтесь: все разработчики регулярно занимаются устранением ошибок в собственном коде. Дадим ряд советов.

- При получении сообщения типа `invalid syntax` проверьте правильность расстановки в коде знаков препинания, например скобок. Найти подобные ошибки несложно, проверив цветовые выделения в окне редактора IDLE.
- При появлении сообщения вида `Python NameError: name 'prin' is not defined` проверьте наличие опечаток в коде, в частности, в написании ключевых слов. В данном случае с ошибкой введено ключевое слово `print`.
- Сообщение вида `EOL while scanning` указывает на отсутствие одной из парных кавычек в записях строковых значений, таких как `'You rock!`. Проверьте, чтобы кавычки были с обеих сторон: `'You rock!'`.

Не бойтесь задавать вопросы

В: Почему для запуска кода Python применяется команда Run Module?

О: Файлы с программным кодом Python называются *модулями*. Поэтому команда Run Module означает "выполнить весь программный код Python, содержащийся в данном файле". Модули часто применяются для более удобной организации программного кода, о чем мы поговорим позже.

В: Я часто слышу термины "ввод" и "вывод". Что это значит?

О: В нашем примере все очень просто. Вывод (или выходные данные) — это текст, генерируемый программой в окне оболочки Python. А ввод (или входные данные) — это текст, вводимый пользователем в данном окне. В общем случае входные и выходные данные могут быть связаны с любыми устройствами: данные можно вводить с помощью мыши или сенсорного экрана, а выводить — в графическом и звуковом виде.

В: Я понял, что команда `print` позволяет выводить текст на экран, но почему у нее такое название? Поначалу кажется, будто это связано с принтером.

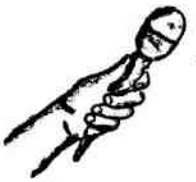
О: Были времена, когда данные чаще выводились на печать, чем на дисплей. В ту эпоху название `print` имело больше смысла, чем сейчас. Конечно, Python появился не настолько давно, но команда `print` исторически обозначала вывод на дисплей во многих языках программирования. Поэтому в Python она связана с выводом на экран (в нашем случае это окно оболочки Python), а аналогичная команда `input`, с которой вы могли познакомиться в одном из предыдущих примеров, служит для получения данных от пользователя в окне оболочки.

В: Можно ли вывести данные, не обращаясь к команде `print`?

О: Конечно. Команда `print` обеспечивает лишь самый простой способ. Компьютеры, а потому и языки программирования, поддерживают большое количество всевозможных способов вывода (и ввода) данных, и Python в этом смысле не исключение. В Python данные можно выводить на веб-страницы, передавать по сети, записывать в файлы, отображать на графических устройствах, прослушивать с помощью звуковых проигрывателей и т.п.

В: Когда я ввожу команду типа `print('Привет!')`, что на самом деле происходит?

О: Запускается встроенный в Python механизм вывода информации на экран. В частности, вы просите функцию `print()` взять текст, заключенный в кавычки, и отобразить его в оболочке Python. Подробнее о функциях и особенностях обработки текста в Python вы узнаете в следующих главах, а пока что достаточно знать, что функция `print()` применяется для вывода данных в окне оболочки Python.



Python в эфире

Интервью недели: "Вы серьезно?"

[Редакция Head First] Добро пожаловать в студию! Всем не терпится поскорее познакомиться с вами.

[Python] С удовольствием отвечу на ваши вопросы.

Итак, ваше имя заимствовано у комик-группы, и вы известны как язык, с которого начинают знакомство с программированием. Скажите честно: вас воспринимают серьезно?

Сами посудите. Меня применяют во множестве промышленных систем: от производства микросхем до голливудских фильмов и центров управления воздушным движением. И это далеко не полный список. Достаточно серьезно?

Хорошо, раз вы такой серьезный язык, то как новичкам вас освоить? Системы, которые вы только что перечислили, весьма сложные. Разве для их реализации не требуются узкоспециализированные, сложные решения?

Одна из причин, по которой меня ценят как новички, так и профессионалы, заключается в простоте и понятности моего кода. Когда-нибудь видели, к примеру, код того же Java? Боже мой, это же ужас! Сколько усилий нужно приложить, чтобы просто вывести "Hello World!" — а у меня это всего одна строка.

Не шутите? Все настолько просто?

Раз уж мы заговорили о Java, приведу пример. Предположим, вы хотите вывести строку `"Привет!"`. Вот как это делается на Java.

```
class HelloWorldApp {
   public static void main(String[] args) {
      System.out.println("Привет!");
   }
}
```

Видите, сколько всего? Я бы назвал это полностью нечитабельным кодом, особенно для того, кто только приступает к изучению программирования. Что вообще все это значит? Неужели все это настолько необходимо? А вот вам моя версия.

```
print('Привет!')
```

Думаю, вы согласитесь, что так намного проще и понятнее. Любой человек, взглянув на эту строку, поймет ее назначение. И это лишь один из примеров. Все, кто изучают Python, находят язык понятным, согласованным...

Согласованным? Что это значит?

Это означает, что вы навряд ли столкнетесь с неприятными сюрпризами. Другими словами, как только вы начнете более-менее понимать синтаксис языка, вы обнаружите, что большинство конструкций работает именно так, как вы ожидаете. Далеко не все языки программирования могут похвастаться этим.

Давайте вернемся к тем системам, которые вы перечислили. Управление воздушным движением, производство микросхем, ракетостроение и прочее — все это очень сложно и профессионально. Боюсь, для наших читателей это не лучший вариант.

Ракетостроение? Давайте не будем пугать людей, я ведь такого не говорил. Я лишь привел примеры отраслей, которые вполне можно считать серьезными, раз уж вы засомневались в моей серьезности. Но в основном Python применяется для создания сайтов, разработки игр и написания пользовательских приложений.

Ладно, поговорим о другом. Известно, что существуют аж две версии Python, причем, как бы это помягче сказать, они *несовместимы друг с другом*. Вы называете это согласованностью?

Как и любая технология, язык программирования постоянно развивается, и да, есть две версии Python: 2 и 3. Версия 3 содержит новые элементы, которых изначально не было в версии 2. Тем не менее существуют способы обеспечить обратную совместимость. Сейчас я покажу...

...к сожалению, наше время в эфире подошло к концу. Спасибо за столь содержательную беседу!

Что ж, до новых встреч!

Воспользовавшись программой Phrase-O-Matic, вы научитесь придумывать классные слоганы, как парни из отдела маркетинга.

Настало время заняться серьезными вещами и написать полноценное приложение на Python. Изучите код программы Phrase-O-Matic и оцените его изящество.

Выдохни

Да, именно так! Позвольте знаниям проникнуть в ваш мозг. Изучите каждую строку кода, прочитайте комментарии и зафиксируйте все в голове. Комментарии — это заметки к коду, начинающиеся с символа #. Можете считать их полезным псевдокодом. Мы поговорим о них в одной из последующих глав. Получив представление о том, как работает код, переходите к следующей странице, где мы все рассмотрим подробнее.

1
```
# Сообщаем Python о том, что собираемся
# использовать функции модуля random
import random
```

2
```
# Создаем три списка: глаголы, прилагательные
# и существительные
verbs = ['Leverage', 'Sync', 'Target',
         'Gamify', 'Offline', 'Crowd-sourced',
         '24/7', 'Lean in', '30,000 foot']

adjectives = ['A/B Tested', 'Freemium',
              'Hyperlocal', 'Siloed', 'B-to-B',
              'Oriented', 'Cloud-based',
              'API-based']

nouns = ['Early Adopter', 'Low-hanging Fruit',
         'Pipeline', 'Splash Page', 'Productivity',
         'Process', 'Tipping Point', 'Paradigm']
```

3
```
# Выбираем по одному глаголу, прилагательному
# и существительному из каждого списка
verb = random.choice(verbs)
adjective = random.choice(adjectives)
noun = random.choice(nouns)
```

4
```
# Составляем фразу, "суммируя" все три слова
phrase = verb + ' ' + adjective + ' ' + noun
```

5
```
# Выводим готовую фразу
print(phrase)
```

Phrase-O-Matic

В общих чертах работа программы заключается в выборе одного слова в каждом из трех списков и комбинировании их в определенной последовательности для получения готовой фразы, выводимой на экран. (Чем не слоган для новой компании?) Не расстраивайтесь, если не понимаете всех нюансов работы программы. Вы ведь только начали изучать программирование, не окончив даже первую главу. На данном этапе мы просто знакомимся с программным кодом.

1. Инструкция `import` сообщает Python о том, что мы собираемся использовать встроенные функции, содержащиеся в модуле `random`. Это позволяет расширить функционал программы. В данном случае мы получаем возможность выбирать произвольные элементы в обрабатываемых списках. В последующих главах мы детальнее познакомимся с инструкцией `import`.

2. Далее в программе создаются три списка. Это делается очень просто: в квадратных скобках перечисляются все элементы списка, заключенные в кавычки, как показано ниже.

```
verbs = ['Leverage', 'Sync', 'Target',
         'Gamify', 'Offline', 'Crowd-sourced',
         '24/7', 'Lean in', '30,000 foot']
```

   Обратите внимание на то, что каждому списку присваивается собственное имя (например, `verbs`), что позволяет впоследствии обращаться к содержимому списков.

3. Далее необходимо случайным образом выбрать по одному слову из каждого списка. Для этого мы применяем функцию `random.choice()`, которая возвращает произвольно выбранный элемент из переданного ей списка. Полученному элементу назначается соответствующее имя (`verb`, `adjective` или `noun`), что позволяет впоследствии сослаться на него.

   `random.choice()` — это встроенная функция Python. С такими функциями мы будем много работать в книге

4. После этого мы создаем фразу-слоган, соединяя три элемента (глагол, прилагательное и существительное). В Python это можно сделать с помощью знака +. Обратите внимание на использование дополнительных пробелов в качестве разделителей. Без них мы получим фразы вида `"Lean inCloud-basedPipeline"`.

   В программировании объединение текстовых строк называется конкатенацией. Данный термин еще не раз вам встретится

5. В конце мы выводим готовую фразу в окне оболочки Python с помощью команды `print`. Все — новый слоган готов!

Запуск кода на выполнение

Вам уже доводилось создавать программу в редакторе IDLE. Давайте проделаем это еще раз. Создайте новый файл Python, выполнив команду **File > New File**, а затем введите в пустом окне приведенный выше код.

Вот пример опечатки. Она будет подсвечена. В подобных случаях внимательно проверьте правильность кода

Уделите особое внимание орфографии и пунктуации. Мы поговорим об этом в последующих главах, но вырабатывать полезную привычку нужно уже сейчас

Заметьте, что IDLE выделяет цветом основные ошибки. Это может происходить по мере ввода кода или при запуске программы, в зависимости от типа ошибки. В данном случае компьютер сообщил о пропущенной кавычке. При наличии подобных выделений тщательно проверьте код и исправьте ошибки

Мы намеренно добавили здесь ошибку, чтобы показать, как работает IDLE. Если вы корректно наберете код, приведенный двумя страницами ранее, то ошибки быть не должно

Теперь сохраните программный код в виде отдельного файла, выполнив в окне редактора IDLE команду **File > Save**. Присвойте программе имя *phraseomatic.py*.

Помните о том, что IDLE подсвечивает элементы кода в зависимости от их назначения

IDLE также вставляет необходимые отступы после нажатия клавиши <Enter>

Дополнительные разделители, т.е. пробелы и символы новой строки, позволяют улучшить читабельность кода. Python в основном игнорирует такие разделители (об исключении мы поговорим позже)

Настало время запустить программу Phrase-O-Matic. Вы уже знаете, как это делается. Выполните команду **Run > Run Module**, предварительно сохранив программу на диске. Следите за обновлением окна оболочки Python — в нем появится новый слоган для будущего проекта.

Чтобы запустить программу, выберите в меню Run команду Run Module. Если программа не была сохранена, то появится соответствующий запрос

```
import random

verbs = ['Leverage', 'Sync', 'Target',
         'Qualify', 'Offline', 'Crowd-sourced',
         '24/7', 'Lean in', '10,000 foot']

adjectives = ['A/B Tested', 'Freemium',
              'Hyperlocal', 'Siloed', 'B-to-B',
              'Oriented', 'Cloud-based', 'API-based']

nouns = ['Early Adopter', 'Low-hanging Fruit',
         'Pipeline', 'Splash Page', 'Productivity',
         'Process', 'Tipping Point', 'Paradigm']

verb = random.choice(verbs)
adjective = random.choice(adjectives)
noun = random.choice(nouns)

phrase = verb + ' ' + adjective + ' ' + noun

print(phrase)
```

Мы запустили программу Phrase-O-Matic несколько раз и получили такие результаты

```
>>>
Sync B-to-B Early Adopter
>>> ========== RESTART ==========
>>>
Leverage Siloed Productivity
>>> ========== RESTART ==========
>>>
24/7 A/B Tested Low-hanging Fruit
>>> ========== RESTART ==========
>>>
Crowd-sourced API-based Tipping Point
>>> ========== RESTART ==========
>>>
Sync API-based Low-hanging Fruit
>>>
```

Чтобы повторно запустить программу Phrase-O-Matic, перейдите в окно редактора кода и снова выполните команду Run > Run Module.

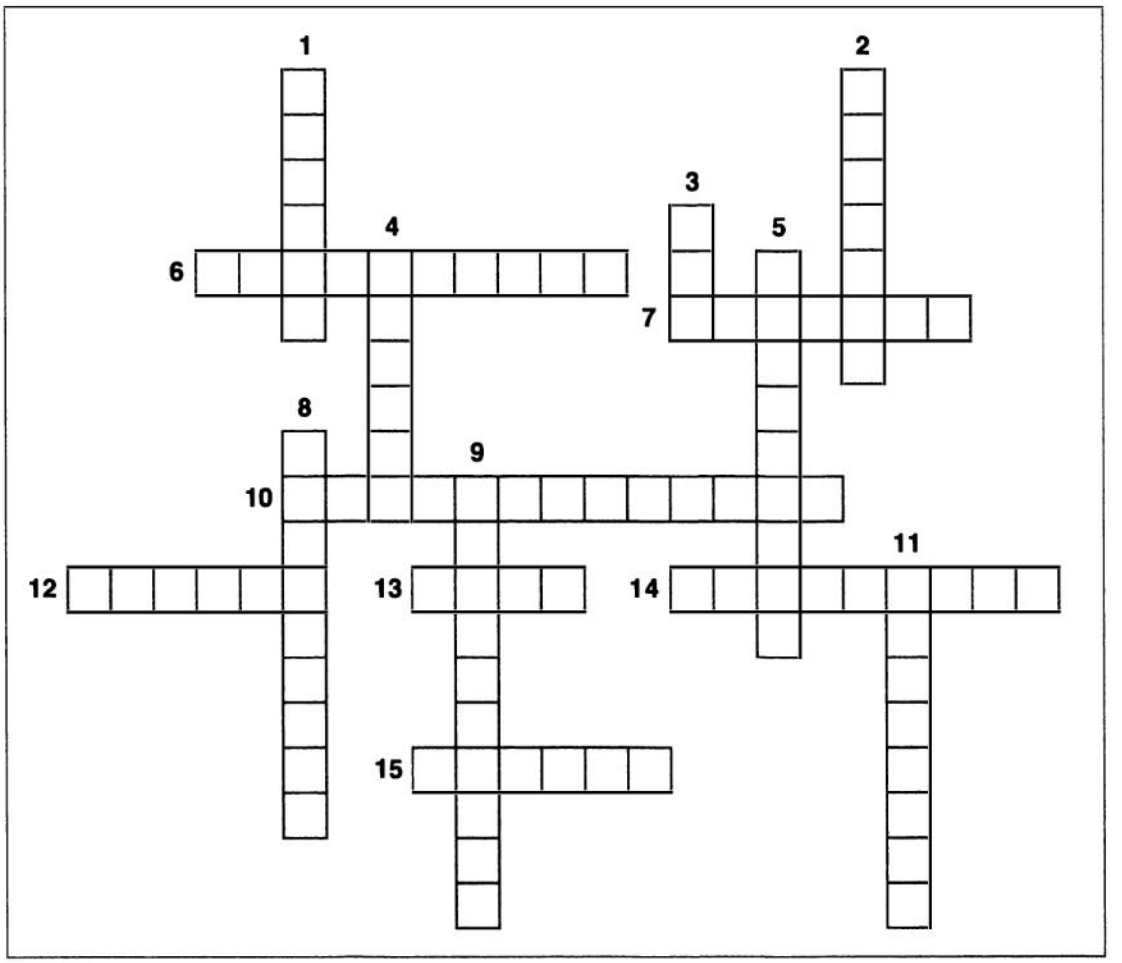
САМОЕ ГЛАВНОЕ

- Прежде чем начинать писать код, необходимо разбить задачу на набор простых операций.
- Подобный набор операций называется алгоритмом или (неформально) рецептом решения задачи.
- Операции реализуются в виде инструкций, которые позволяют выполнять простые действия, принимать решения и управлять ходом вычислений, повторяя те или иные участки кода.
- Вычислительное мышление — это умение представлять решение задачи в форме, которая может быть реализована с помощью компьютера.
- Кодирование — это описание алгоритма на языке программирования, понятном компьютеру.
- Прежде чем записывать алгоритм на языке программирования, его иногда выражают в виде псевдокода, более понятного человеку.
- Языки программирования предназначены для записи инструкций в понятном компьютеру формате.
- Английский язык плохо подходит для программирования, поскольку содержит слишком много двусмысленных понятий.
- Существует огромное количество языков программирования, у каждого из которых свои преимущества и недостатки, но примерно одинаковые возможности.
- Язык Python получил свое название не от змеи, а от британской комик-группы "Монти Пайтон", поклонником которой был создатель языка.
- Как новички, так и профессионалы ценят Python за простоту и согласованность.
- Существуют две версии языка: Python 2 и 3. В книге рассматривается Python 3, хотя в большинстве случаев разница между версиями совершенно незначительна.
- Код Python выполняется интерпретатором, преобразующим высокоуровневые команды языка в низкоуровневый машинный код, выполняемый компьютером на аппаратном уровне.
- Среда разработки Python включает редактор кода IDLE, специально предназначенный для написания программ на этом языке.
- Улучшить читаемость кода можно с помощью пробельных символов.
- Для простых операций ввода и вывода данных в окне оболочки Python применяются функции `input()` и `print()`.

Кроссворд

Дайте передышку своему мозгу, загрузив правое полушарие чем-то увлекательным.

Перед вами кроссворд, состоящий из слов, которые встречались на протяжении главы 1.



По горизонтали

6. Команда в коде
7. Устройство визуального отображения информации
10. Программа, которая преобразует код Python в машинный код
12. Операция подключения внешнего модуля в коде
13. На нем пишут программы
14. Файл, содержащий код
15. Внешний подключаемый файл Python

По вертикали

1. Процесс выполнения программы
2. Окно ввода команд интерпретатора и вывода результатов работы программы
3. Текст программы
4. Другое название алгоритма
5. Программа, записанная в произвольной форме
8. Правила написания кода
9. Процесс создания программы
11. Порядок выполняемых программой действий

Возьмите карандаш
Решение

В рецептах блюд содержатся не только инструкции по их приготовлению, но и перечисляются все необходимые ингредиенты (в программировании мы называем их *объектами*). Какие объекты используются в рецепте рыбной ловли?

1. Наживить червя на крючок.
2. Закинуть приманку в пруд.
3. Следить за поплавком, пока он не дернется.
4. Выудить рыбку.
5. Если рыбалка окончена, отправиться домой, иначе вернуться к п. 1.

Возьмите карандаш
Решение

Прежде всего вы должны понять, что компьютеры выполняют **в точности** те команды, которые вы им передаете, — ни больше ни меньше. Рассмотрим рецепт рыбной ловли, приведенный в начале главы. Если робот будет следовать ему буквально, то с какими трудностями он может столкнуться? Насколько успешной будет такая рыбалка?

- ☑ А. Если в пруду нет рыбы, то вы будете ее ловить очень долго (т.е. вечно).
- ☑ Б. Если червь сорвется с крючка, то вы не узнаете об этом и не сможете его заменить.
- ☑ В. Что, если у нас закончатся черви?

Похоже, правильный ответ: "Все вышеперечисленное"

- ☑ Г. Что делать с рыбой после того, как мы ее поймали?
- ☑ Д. А что делать с удочкой?
- ☑ Е. *Есть ли правила забрасывания удочки? Если червь попадет на лист кувшинки, то нужно ли все повторить?*

Обычно, как только поплавок уходит под воду, мы сначала "подсекаем" рыбу. Здесь об этом ничего не сказано.

Как мы узнаем, что рыбалка окончена? По времени? Когда у нас закончатся черви? Или еще как-то?

В этом рецепте мы сделали целый ряд предположений. Наверняка вы придумали множество других инструкций, которые здесь не указаны.

Код на магнитиках : решение

У нас был отличный рецепт приготовления омлета из трех яиц, записанный с помощью набора магнитиков на дверце холодильника. Вот только незадача: кто-то пришел и все перемешал. Сможете расположить магнитики в правильном порядке, чтобы восстановить алгоритм? Учтите, что наш рецепт описывает два вида омлета: обычный и с сыром.

Вот магнитики по порядку!

Здесь возможно несколько вариантов. Главное, чтобы вы поняли наше решение и чтобы ваше решение, если оно другое, имело смысл

Нагреть сковороду

Разбить три яйца в миску

Начинаем с того, что подогреваем сковороду и разбиваем яйца в миску

Пока все не перемешается:

Взбиваем яйца, пока они не перемешаются равномерно

Взбить яйца

Обратите внимание на отступ данного этапа: тем самым мы показываем, что действие выполняется, пока яйца не будут взбиты

Вылить яйца на сковороду

Далее выливаем яйца на сковороду...

Пока яичница не поджарится:

...и жарим их, пока яичница не будет готова

Перемешивать

Обратите внимание на отступ данного этапа: тем самым мы показываем, что действие выполняется, пока яичница не будет готова

Если клиент заказал сыр:

Если клиент захотел сыр, добавить его

Добавить сверху сыр

Мы тоже выделяем этот этап отступом, потому что он выполняется, только если клиент заказал сыр

Снять сковороду с плиты

Далее снимаем сковороду с плиты и выкладываем яичницу на тарелку

Выложить яичницу на тарелку

Подать

Наконец, подаем яичницу клиенту

Фразы одного сорта
Решение

Слева записан ряд инструкций на понятном для вас языке, а справа — они же на языке программирования. Соедините линией каждое из предложений с соответствующей инструкцией программного кода. Первую линию мы нарисовали за вас.

Напечатай "Привет!" на экране.

Если температура выше, чем 27 °C, напечатай "Надень шорты" на экране.

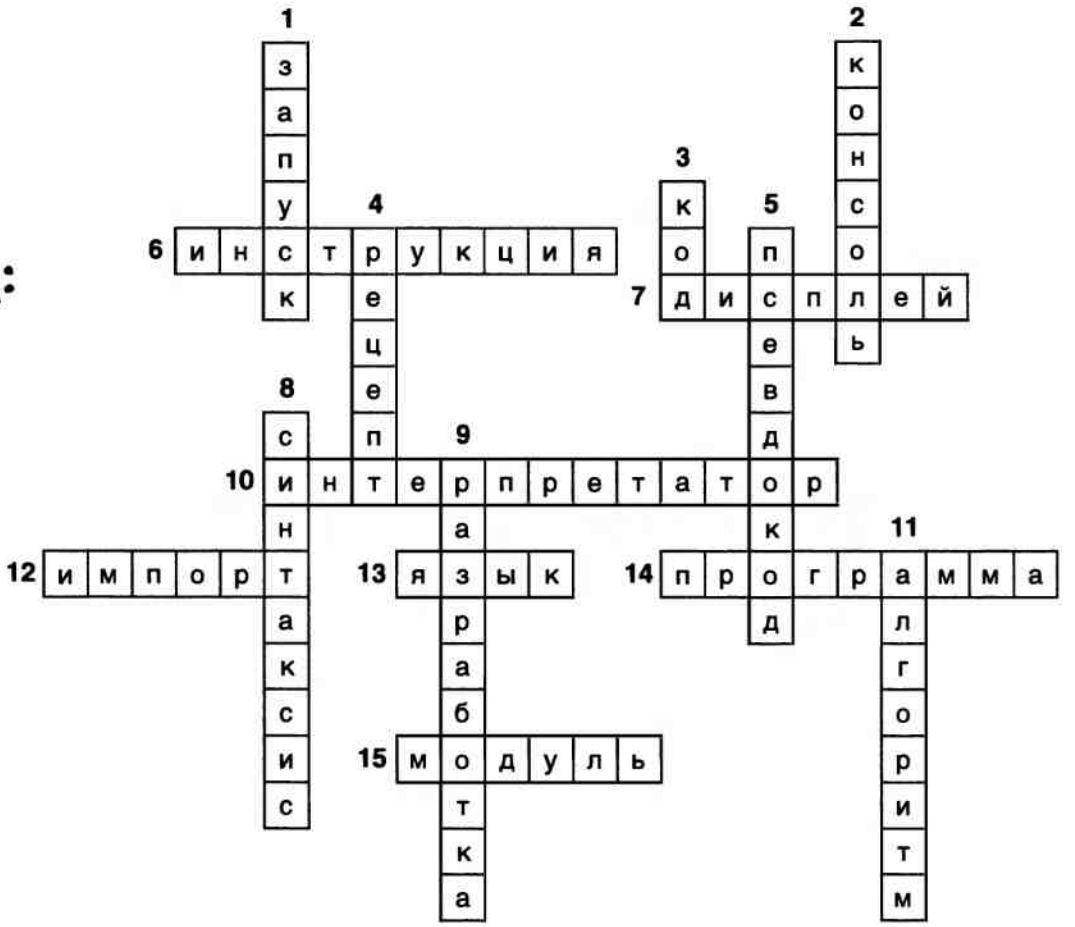
Создай список продуктов, включающий хлеб, молоко и яйца.

Налей пять напитков.

Спроси пользователя: "Как тебя зовут?"

```
for num in range(0, 5):
        pour_drink();

name = input('Как тебя зовут? ')

if temperature > 27:
        print('Надень шорты')

grocery_list = ['хлеб', 'молоко', 'яйца']

print('Привет!')
```

Кроссворд: решение



2 значения, переменные и типы данных

Сначала было значение

В реальности компьютеры умеют делать всего две вещи: сохранять значения и выполнять над ними операции. А как же редактирование текста, ретуширование фотографий и онлайн-покупки, спросите вы? Удивительно, но решение любых подобных задач в конечном счете сводится к выполнению **простых операций** над не менее **простыми значениями**. Человек, обладающий вычислительным мышлением, понимает, что это за операции и значения и как с их помощью создать что-то более сложное. В этой главе мы познакомимся с типами значений, узнаем, какие операции можно выполнять над ними и какую роль во всем этом играют **переменные**.

Калькулятор возраста собаки

Не бойтесь, мы не заставим вас изучать 50 страниц документации Python, где описываются значения и операции над ними. Лучше давайте напишем что-нибудь полезное и интересное!

Наш первый проект – **калькулятор возраста собаки**. Вы указываете, сколько лет собаке, а программа сообщает, какому возрасту это соответствует *по человеческим меркам*. Для этого всего лишь нужно умножить реальный возраст собаки на 7. Неужели все так просто? Что ж, посмотрим...

С чего же начать? Сразу приступим к написанию кода? Вспомните концепцию псевдокода, с которой вы познакомились в предыдущей главе. Псевдокод позволяет описать общий алгоритм решения задачи, прежде чем переходить непосредственно к программированию. Это будет наш первый шаг.

Тем более что вы еще даже не умеете программировать!

Но что такое псевдокод и как он выглядит? По сути, это текстовое описание алгоритма. Вы пошагово перечисляете все, что должна сделать программа для решения поставленной задачи (в нашем случае это вычисление возраста собаки по человеческим меркам).

Вы уже видели пример псевдокода в рецепте рыбной ловли из главы 1

Тузик, 12 лет

Возьмите карандаш

Итак, нам предстоит написать псевдокод. Для начала подумайте, как бы вы описали алгоритм, или рецепт, вычисления возраста собаки по человеческим меркам. Далее запишите все свои соображения в виде простых текстовых команд. Программа должна взаимодействовать с пользователем, спрашивая у него имя и биологический возраст собаки. В конце необходимо вывести что-то осмысленное, например: "Вашей собаке Рокки сейчас 72 по меркам людей".

В общем, просто запишите псевдокод на своем родном языке. **Важно**: прежде чем двигаться дальше, сравните свое решение с нашим в конце главы.

Если испытываете затруднения, не стесняйтесь подсмотреть решение

Запишите здесь свой псевдокод

Вот пример того, как будет работать калькулятор возраста собаки

```
Оболочка Python
Как зовут вашу собаку? Тузик
Сколько лет вашей собаке? 12
Вашей собаке Тузик сейчас 84 по меркам людей
>>>
```

Смайли, 1 год

Филя, 5 лет

От псевдокода к программному коду

Теперь, когда у нас есть псевдокод, мы четко представляем, какие действия должны выполняться в программе, реализующей калькулятор возраста собаки. Конечно, псевдокод не содержит всех необходимых подробностей – программа будет обрастать ими по мере *реализации* каждого обозначенного в алгоритме действия на выбранном нами языке программирования.

Запись умозрительных идей, алгоритмов и псевдокода на языке программирования часто называют реализацией

Таким образом, нам нужно пошагово проанализировать псевдокод и реализовать его в виде программного кода.

Возьмите карандаш

На первом этапе необходимо пошагово проанализировать составленный ранее алгоритм и постараться описать, что конкретно должна делать программа в каждом случае. Сверьтесь с нашим решением, прежде чем двигаться дальше. Первый ответ мы вписали за вас.

Как всегда, если возникнут трудности, не стесняйтесь заглянуть в ответ, но сначала постарайтесь пройти каждую строку

Псевдокод калькулятора возраста собаки

1. **Спросить у пользователя имя собаки.**
2. **Спросить у пользователя возраст собаки.**
3. **Умножить возраст собаки на 7, чтобы получить возраст по человеческим меркам.**
4. **Вывести на экран:**

   "Вашей собаке"
   имя собаки
   "сейчас"
   человеческий возраст собаки
   "по меркам людей"

1. *Сообщаем пользователю о том, что нужно ввести имя собаки, и просим его ввести строку. Это имя придется где-то сохранить, т.к. оно понадобится в п. 4.*

2.

3.

4.

Введите сюда свои комментарии по каждому этапу

Не бойтесь задавать вопросы

В: Зачем нужен псевдокод, если компьютер понимает только программы, написанные на языке программирования?

О: Создавая псевдокод, вы описываете сам алгоритм, не погружаясь в детали программного кода. Это дает возможность проанализировать решение и найти способы улучшить его, прежде чем реализовывать его в виде кода.

В: Неужели опытные разработчики используют псевдокод?

О: Да! Всегда полезно спланировать заранее, что вы будете делать, прежде чем приступать к написанию сложного программного кода. У некоторых разработчиков хорошо получается все держать в голове, но большинство используют псевдокод или похожие методики, чтобы создать план действий. Псевдокод — это еще и хороший способ обмена идеями между разработчиками.

Вы сейчас здесь

Этап 1: запрос входных данных

Теперь мы готовы реализовать первый этап — получение от пользователя имени собаки. Согласно разработанному ранее псевдокоду сначала нужно попросить пользователя ввести имя, а затем это имя необходимо где-то сохранить, т.к. оно понадобится в п. 4, когда мы выведем имя и возраст собаки по человеческим меркам. Итак, нам нужно сделать две вещи: запросить у пользователя имя собаки и сохранить имя на будущее. Сначала займемся запросом.

Псевдокод калькулятора возраста собаки

1. **Спросить у пользователя имя собаки.**
2. **Спросить у пользователя возраст собаки.**
3. **Умножить возраст собаки на 7, чтобы получить возраст по человеческим меркам.**
4. **Вывести на экран:**

   "Вашей собаке"
   имя собаки
   "сейчас"
   человеческий возраст собаки
   "по меркам людей"

В главе 1 вам уже встречались фрагменты кода, в которых использовалась функция `input()`, отвечающая за получение данных от пользователя. Это одна из многочисленных встроенных функций Python.

В книге мы будем регулярно иметь дело с функциями, и со временем вы поймете, как они работают, а пока что достаточно знать, что функция — это способ попросить Python выполнить то или иное действие за нас, не беспокоясь о деталях

Сначала рассмотрим синтаксис вызова функции `input()`, а затем разберемся, как она работает.

Синтаксис определяет способ записи инструкций на выбранном языке программирования

Начните с имени функции: `input`

Затем поставьте открывающую круглую скобку

Далее в кавычках укажите текст, который хотите вывести пользователю

Завершите инструкцию закрывающей круглой скобкой

```
input("Как зовут вашу собаку? ")
```

Как работает функция input()

Теперь, когда вы знаете синтаксис функции `input()` (т.е. как ее вводить), постараемся понять принцип ее работы.

1. Когда интерпретатор встречает вызов функции `input()`, он выводит текст запроса в окне оболочки Python.
2. После этого интерпретатор ждет, пока пользователь введет ответ и нажмет клавишу <Enter>.
3. Наконец, введенный пользователем текст передается обратно в программу Python.

Итак, введенный пользователем текст передается обратно в программу, но что это значит? Прежде чем завершиться, функция `input()` возвращает полученные от пользователя данные в качестве *результата* своей работы.

Впрочем, от полученного текста будет не слишком много пользы, если мы не сумеем его как-то сохранить в программе, ведь он понадобится нам на этапе 4, когда мы будем формировать выходное сообщение. Как это можно сделать в программе, написанной на Python?

Закрепим

Синтаксис говорит нам, как писать инструкции Python.

Семантика говорит нам, что означают инструкции Python.

Сохранение значений в переменных

В программировании мы регулярно сталкиваемся с необходимостью хранить значения. Для этого используются **переменные**, представляющие собой специальные имена, по которым можно обратиться к ранее сохраненным данным. Вот как *присвоить* значение переменной.

Подобная возможность имеется во всех языках программирования

Сначала назовите переменную. О выборе корректных имен мы поговорим позже

Далее поставьте знак равенства, после которого укажите значение, присваиваемое переменной

```
dog_name = 'Тузик'
```

Текст мы называем строкой. Строка — это просто набор символов. Такая терминология привычна для всех языков программирования. В Python есть множество других типов значений, например числа, о которых мы вскоре поговорим

Слева указывается переменная, т.е. имя, которое мы можем многократно использовать в программе

Справа стоит значение; в данном случае это строка `'Тузик'`

Сохранение вводимых данных в переменной

Немного разобравшись с переменными (мы продолжим знакомство с ними чуть позже), давайте попробуем сохранить вводимое пользователем значение в переменной. Для этого мы просто вызываем функцию `input()` и присваиваем возвращаемое ею значение некоторой переменной. Вот как это делается.

Будем использовать переменную `dog_name`

Мы вызываем функцию `input()`, которая спрашивает у пользователя: "Как зовут вашу собаку?"

Хотите узнать, как давать имена переменным? Или как правильно использовать одинарные и двойные кавычки? Вскоре мы все это обсудим

```
dog_name = input("Как зовут вашу собаку? ")
```

Когда пользователь заканчивает вводить имя, функция `input()` передает это имя назад в программу в виде возвращаемого значения

Возвращаемое значение присваивается переменной `dog_name`

Переходим к пункту 2

Псевдокод калькулятора возраста собаки

1. **Спросить у пользователя имя собаки.**
2. **Спросить у пользователя возраст собаки.**
3. **Умножить возраст собаки на 7, чтобы получить возраст по человеческим меркам.**
4. **Вывести на экран:**

   "Вашей собаке"
   имя собаки
   "сейчас"
   человеческий возраст собаки
   "по меркам людей"

Этап 2: еще один запрос

Кроме имени, у пользователя нужно еще запросить биологический возраст собаки. Как это сделать? Точно так же – с помощью функции `input()`, только на этот раз указав текст "Сколько лет вашей собаке?" Полученные данные мы сохраним в переменной `dog_age`. Хотя правильнее сказать не "мы сохраним", а "вы сохраните", т.к. вас ожидает очередное упражнение.

Теперь ваша очередь. Напишите код для получения биологического возраста собаки с помощью функции `input()`. Выдайте пользователю запрос `"Сколько лет вашей собаке?"` и сохраните результат в переменной `dog_age`. Сделайте пометки, уточняющие назначение каждого элемента кода. Сверьтесь с ответом, приведенным в конце главы, прежде чем двигаться дальше.

Пора выполнить код

Одно дело — изучать код в книге, и совсем другое — выполнять реальную программу. Давайте попробуем выполнить составленный ранее код, но только будем вводить его не в редакторе IDLE, а непосредственно в оболочке Python. Почему? Как вы увидите, оболочка Python является удобным инструментом для разного рода экспериментов с небольшими фрагментами кода. Не переживайте: когда дело дойдет до написания более крупных программ, мы вернемся в редактор IDLE.

Оболочка Python прекрасно подходит для тестирования небольших фрагментов кода

Итак, запустите привычным способом среду IDLE, только на этот раз мы будем вводить код прямо в оболочке Python.

Оболочка Python

```
Python 3.6.0 (default, Feb 22 2017, 13:46:35)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0] on darwin
Type "copyright", "credits" or "license()" for more information
>>>
```

Это приглашение интерпретатора команд. Он ждет, что вы что-то введете.

Номера версий и начальные сообщения могут отличаться, в зависимости от версии дистрибутива и операционной системы

Найдите в окне оболочки приглашение командной строки, которое выглядит как >>>. Введите 1 + 1 и нажмите <Enter>. Интерпретатор Python вычислит выражение и отобразит результат (в нашем случае 2), после чего снова выведет приглашение командной строки. Теперь вы знаете, как работать в оболочке Python! Далее попробуем выполнить интересующий нас фрагмент кода.

1 **Введите в оболочке код, запрашивающий имя собаки, и нажмите клавишу <Enter>.**

Оболочка Python

```
>>> dog_name = input("Как зовут вашу собаку? ")
Как зовут вашу собаку?
```

Введите эту строку и нажмите клавишу <Enter>

Python выведет эту строку

2 Введите какое-нибудь имя и снова нажмите <Enter>.

Оболочка Python

```
>>> dog_name = input("Как зовут вашу собаку? ")
Как зовут вашу собаку? Рокки
>>>
```

Введите имя своей собаки и нажмите клавишу <Enter>

Python получит имя собаки, сохранит его и присвоит переменной dog_name. После этого вы увидите новое приглашение командной строки

При выполнении операции присваивания не происходит никаких вычислений, в отличие, например, от операции 1 + 1. Вместо этого, как вы уже знаете, программа берет значение справа от знака = и присваивает его переменной, стоящей слева

СИЛА МЫСЛИ

Переменная dog_name будет хранить значение 'Рокки' или другое указанное вами имя собаки. Но как это проверить?

3 Нам уже доводилось применять команду print несколько раз. Воспользуемся ею, чтобы узнать значение переменной dog_name. Можно также просто ввести имя переменной в командной строке.

Оболочка Python

```
>>> dog_name = input("Как зовут вашу собаку? ")
Как зовут вашу собаку? Рокки
>>> print(dog_name)
Рокки
>>> dog_name
'Рокки'
>>>
```

С помощью функции print() можно отобразить текущее значение переменной dog_name, т.е. Рокки

Можно просто ввести имя переменной, и интерпретатор сообщит ее значение

Заметьте, что функция print() выводит строковое значение без кавычек, в отличие от интерпретатора, который заключает строку в кавычки

Я заметила, что одинарные и двойные кавычки используются непоследовательно. В одних случаях текст берется в одинарные кавычки, в других — в двойные? С чем это связано?

А вы наблюдательны! Прежде всего напомним, что заключенный в кавычки текст рассматривается интерпретатором Python как строковое значение. Вы правы: мы в основном заключали строки в одинарные кавычки, но в функции `input()` применяются двойные кавычки. Впрочем, для Python это не важно — оба варианта равнозначны, лишь бы кавычки были парными. Другими словами, если строка начинается одинарной кавычкой, то и заканчиваться она должна одинарной кавычкой. То же самое и с двойными кавычками.

В целом большинство разработчиков Python предпочитают одинарные кавычки, но есть одна ситуация, когда нужны именно двойные кавычки: это ситуация, когда одинарная кавычка является *частью строки*.

Предположим, мы хотим вывести запрос на английском языке:

```
dog_name = input("What is your dog's name? ")
```

Чтобы одинарную кавычку можно было использовать в строке, заключите текст в двойные кавычки

Точно так же, если текст строки включает двойные кавычки, то ее всю нужно заключить в одинарные кавычки.

Ввод кода в редакторе

Но оставим развлечения в оболочке Python: нам предстоит написать настоящую программу! Нам нужно ввести две строки кода (в которых запрашиваются имя и биологический возраст собаки) и протестировать их в окне редактора кода. После этого мы займемся остальной частью псевдокода.

Выполните команду **File New > File** в редакторе IDLE и введите в новом окне первые две строки калькулятора возраста собаки.

```
dog_name = input("Как зовут вашу собаку? ")
dog_age = input("Сколько лет вашей собаке? ")
```

Проверив правильность введенного кода, сохраните его в виде отдельного файла, воспользовавшись командой File Save. Назовите файл *dogcalc.py* и сохраните его в новой папке *ch2*.

Обратите внимание на пробел. Это пробел между подсказкой и тем, что вводит пользователь

Тест-драйв

Пиктограмма "Тест-драйв" означает, что нужно сделать паузу и протестировать имеющийся на данный момент код

Мы еще не закончили наш калькулятор, но уже сейчас можно протестировать имеющийся код, чтобы убедиться в отсутствии ошибок. После сохранения файла выполните команду **Run > Run Module** и посмотрите, что выводится в окне оболочки Python. Программа должна запросить имя и биологический возраст собаки.

Помните: на данном этапе программа лишь спрашивает имя и возраст собаки, после чего выводит приглашение командной строки

Оболочка Python

```
Как зовут вашу собаку? Рокки
Сколько лет вашей собаке? 12
>>>
```

Если при выполнении программы вы получите сообщение об ошибке "SyntaxError: EOL while scanning string literal", то проверьте использование одинарных и двойных кавычек в коде

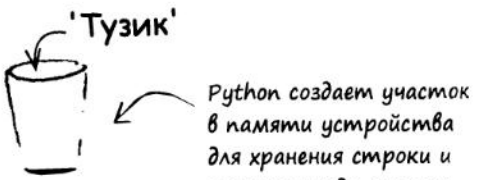
Подробнее о переменных

Вы уже знаете, как объявить переменную и присвоить ей значение. Но как работают переменные и как использовать их в программе? Давайте потратим немного времени, чтобы получше разобраться с переменными, после чего мы наконец завершим наш калькулятор возраста собаки. Для начала узнаем, что происходит в процессе присваивания переменной значения.

```
dog_name = 'Тузик'
```

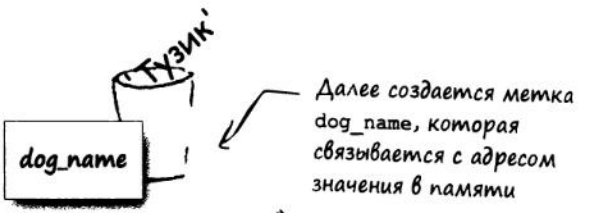
Такая команда обычно называется "присваивание"

1. Первое, что делает интерпретатор Python, — оценивает выражение, стоящее справа от знака равенства. В нашем случае это строка `'Тузик'`. Далее интерпретатор находит в памяти компьютера свободную ячейку и сохраняет в ней указанную строку. Представим это как пустую чашку, в которую помещается строка `'Тузик'`.



Python создает участок в памяти устройства для хранения строки и заносит туда 'Тузик'

2. После сохранения строки `'Тузик'` интерпретатор Python создает метку `dog_name`, которая прикрепляется к чашке.

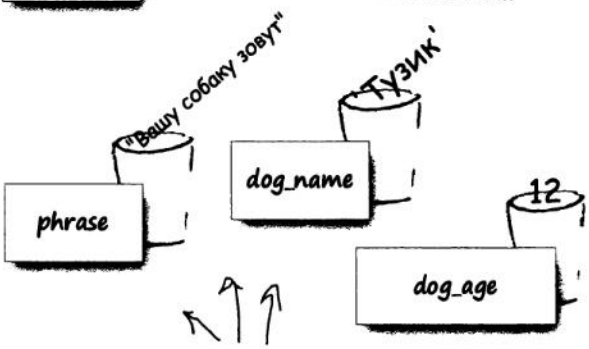


Далее создается метка dog_name, которая связывается с адресом значения в памяти

3. В программе можно создавать сколько угодно переменных. Вот еще две:

```
phrase = "Вашу собаку зовут "
dog_age = 12
```

Числа можно хранить так же, как и строки

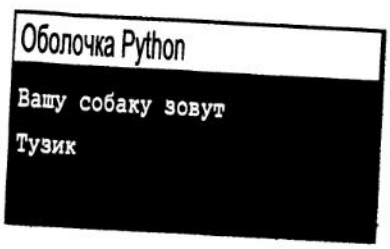


Мы можем создать столько значений переменных, сколько нам нужно. Они будут храниться в памяти, пока не понадобятся

4. Теперь для получения значения, сохраненного в памяти компьютера, достаточно обратиться к соответствующей переменной.

```
print(phrase)
print(dog_name)
```

Эти инструкции извлекают значения phrase и dog_name и выводят на экран...



Добавляем выражения

В приведенных выше примерах переменным присваивались только простые значения, но так будет далеко не всегда. Во многих случаях значения переменных представляются вычисляемыми *выражениями*, которые напоминают самые обыкновенные математические операции. Для записи таких операций применяются *операторы* +, –, * и /. Например, что если наш Тузик стал на год старше?

В программировании операция умножения обозначается символом "звездочка" (*)

Вот выражение

```
dog_age = 12 + 1
```

Мы называем знак + оператором, а числа 12 и 1 — операндами

Результатом будет единственное значение: 13

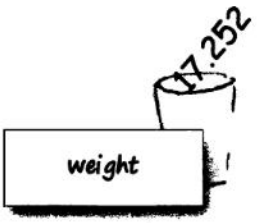
После вычисления выражения переменной dog_age будет присвоено значение 13

А что если необходимо перевести вес собаки из фунтов в килограммы?

Здесь мы используем оператор умножения

```
weight = 38 * 0.454
```

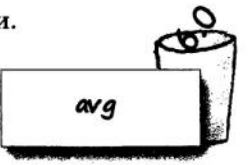
В результате выполнения этой инструкции переменной weight будет присвоено значение 17.252



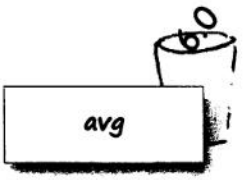
А как вычислить средний возраст трех собак: Тузика, Фили и Смайли.

```
avg = (12 + 5 + 1) / 3
```

Операции можно группировать с помощью скобок



Во всех этих выражениях вместо значений можно подставлять переменные. В качестве примера перепишем код вычисления среднего возраста собак.




```
tuzik = 12
philly = 5
smiley = 1
avg = (tuzik + philly + smiley) / 3
```

Везде, где стоит переменная, она заменяется своим значением при вычислении выражения

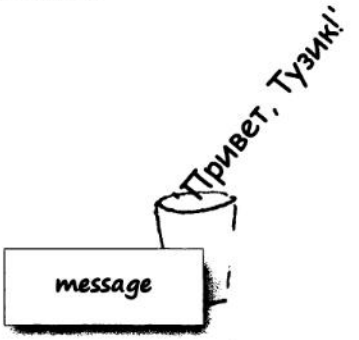
При составлении выражений вы не ограничены только математическими операциями с числами. Помните, в главе 1 рассказывалось о конкатенации строк? Эту операцию можно применить для объединения нескольких строк.

```
greeting = 'Привет,'
name = 'Тузик!'
message = greeting + ' ' + name
```

Давайте создадим несколько строк

И соединим (точнее, конкатенируем) их вместе

В результате получаем строку `'Привет, Тузик!'`, которая присваивается переменной `message`



Изменение переменных

Термин "переменная" означает, что значение может со временем измениться. Рассмотрим, к примеру, рост Тузика, который первоначально равен 22 см.

```
dog_height = 22
```

Создаем новую переменную и присваиваем ей значение 22

Предположим, Тузик подрос на сантиметр, и теперь значение переменной `dog_height` необходимо увеличить на 1.

```
dog_height = 22 + 1
```

Как всегда, сначала вычисляется выражение справа, которое равно 23. Оно затем присваивается переменной dog_height. Таким образом, значение этой переменной меняется с 22 на 23

Впрочем, существует более удобный способ обновить рост собаки. Вот, что можно сделать, если Тузик подрос еще на 2 см.

```
dog_height = dog_height + 2
```

❶ Переменная допускается везде, где ожидается значение, поэтому в правой части мы прибавляем 2 к текущему значению `dog_height`, получая 23 + 2 = 25

❷ После этого новым значением переменной `dog_height` становится 25

Приоритет операторов

Выполним следующую инструкцию:

```
mystery_number = 3 + 4 * 5
```

Чему будет равна переменная `mystery_number`: 35 или 23? Все зависит от того, какое действие будет выполняться первым: сложение чисел 3 и 4 или умножение чисел 4 и 5. В данном случае правильный ответ: 23.

Как определить порядок вычислений? Для этого существует *приоритет операторов*, который определяет, какие операции выполняются первыми, а какие — последними. Приоритет операторов придуман не программистами, а математиками, и вы наверняка изучали его в школе на уроках алгебры. Впрочем, если вы успели все позабыть, то не волнуйтесь: сейчас мы напомним, что к чему.

Операторы упорядочены по приоритету от наибольшего к наименьшему.

Наибольший

`**` — Две звездочки означают операцию возведения в степень, которая имеет наибольший приоритет

Если еще не забыли математику, то 2**3 означает 2^3

`-` — Далее по старшинству идет отрицание (т.е. знак 'минус' перед числом)

`* / %` — Далее умножение, деление и деление по модулю

Оператор % означает деление по модулю, т.е. взятие остатка от деления. Например, 7 % 3 = 1, потому что целочисленное деление 7 на 3 дает 2, и в остатке получаем 1

`+ -` — И наконец, сложение и вычитание

Наименьший

Вычисление выражений согласно приоритету операторов

Чтобы понять, как работает приоритет операторов, рассмотрим следующее выражение:

```
3 / 1 + 2 * 3 ** 2
```

3

9

18

21

Наибольший приоритет здесь имеет возведение в степень: $3^2 = 9$

Далее выполняются умножение и деление. У них равный приоритет, поэтому очередность может быть любой (результат один и тот же)

Наименьший приоритет имеет оператор сложения. Получаем 3 + 18 = 21

В интерпретаторе Python вы получите результат 21.0. Почему так, узнаете позже

Закрепим

Всегда помните о том, что операции выполняются слева направо в таком порядке: скобки, возведение в степень, затем умножение, деление, сложение и наконец вычитание.

Числовые выражения вычисляются в порядке приоритета операторов (от наиболее к наименее приоритетному). Для простоты в данных примерах используются только числа, хотя в общем случае выражения могут включать как числа, так и переменные — к ним применяются одинаковые правила приоритетности.

Но как быть, если нужно изменить заданный по умолчанию порядок выполнения математических операций? Как заставить компьютер сначала сложить числа 1 и 2 и только после этого приступить к умножению и делению остальных частей выражения? Конечно же, с помощью скобок! Скобки — это оператор специального типа, позволяющий изменять порядок выполнения операций.

```
3 / (1 + 2) * 3 ** 2
```

3

9

1

9

Наибольший приоритет здесь имеет сложение в скобках: 1 + 2 = 3

Далее необходимо выполнить возведение в степень ($3^2 = 9$), прежде чем выполнять умножение или деление

После этого нужно выполнить деление, так как операции с равным приоритетом (умножение и деление) выполняются слева направо: 3 / 3 = 1

Наконец, умножаем 1 на 9, получая 9

Добавлять в выражение скобки, не изменяющие порядок его вычисления, не запрещено, хотя это снижает читабельности кода

В выражение разрешается включать произвольное количество парных скобок, чтобы задать нужный порядок вычислений. Рассмотрим такой пример:

```
(((3 / 1) + 2) * 3) ** 2
```

В этом выражении сначала выполняется операция деления, затем — сложение, потом — умножение и только после этого — возведение в степень. Результат будет равен 225

Множество выражений Python потеряло свои значения. Не могли бы вы помочь в их восстановлении? Соедините каждое из выражений слева с соответствующим значением справа. Будьте бдительны — некоторые из значений лишние.

Выражения	Значения
'kit e' + ' ' + 'cat'	1
(14 – 9) * 3	2
3.14159265 * 3**2	15
42	21
'h' + 'e' + 'l' + 'l' + 'o'	28.27433385
8 % 3	42
7 – 2 * 3	-13
(7 – 2) * 3	'kit e cat'
	'hello'

И кто здесь обманщик?

Это операция деления по модулю. Она широко применяется в программировании, хоть и кажется странной. Другое название — деление с остатком

УПРАЖНЕНИЕ

В игре в наперстки используются три непрозрачных стаканчика и небольшой шарик. Для программной реализации игры нам понадобится несколько переменных и значений. Используя все свои знания о переменных, значениях и операциях присваивания, попробуйте сыграть в игру. Просмотрите код и определите переменную, которая после выполнения последней операции будет содержать значение 1. В каком стаканчике окажется шарик: 1, 2 или 3? Делайте ставки, господа!

```
cup1 = 0
cup2 = 1
cup3 = 0
cup1 = cup1 + 1
cup2 = cup1 - 1
cup3 = cup1
cup1 = cup1 * 0
cup2 = cup3
cup3 = cup1
cup1 = cup2 % 1
cup3 = cup2
cup2 = cup3 - cup3
```

Попробуйте в уме выполнить приведенный здесь код и определить переменную, которая в конце будет содержать значение 1. Сверьтесь с ответом, приведенным в конце главы (сам код вводить не обязательно).

Сила мысли

Чему равно следующее выражение? Или вы считаете, что такая операция ошибочна, ведь мы умножаем строку на число?

```
3 * 'ice cream'
```

Попробуйте ввести это выражение в оболочке Python

Взлом Кода

Сдавая экзамен в разведшколе, вам предстоит взломать пароль, который закодирован в программе. Мысленно проанализируйте программу и попытайтесь понять, каким будет пароль. Только не спешите: у разведчика нет права на ошибку. Желаем удачи!

```
word1 = 'ox'
word2 = 'owl'
word3 = 'cow'
word4 = 'sheep'
word5 = 'flies'
word6 = 'trots'
word7 = 'runs'
word8 = 'blue'
word9 = 'red'
word10 = 'yellow'
word9 = 'The ' + word9
passcode = word8
passcode = word9
passcode = passcode + ' f'
passcode = passcode + word1
passcode = passcode + ' '
passcode = passcode + word6
print(passcode)
```

Вывод пароля

Вот ваш пароль. Проанализируйте программу, чтобы узнать его

Руки прочь от клавиатуры!

Как вы уже знаете, у переменной есть имя и есть значение.

Но как следует называть переменные? Всякое ли имя подойдет? Вообще-то, нет, хотя правила именования переменных весьма просты. По сути, их всего два.

1. **Имя переменной должно начинаться с буквы или символа подчеркивания.**
2. **Далее может идти произвольное количество букв, чисел и символов подчеркивания.**

И конечно же, не допускается применять в качестве имен переменных *зарезервированные ключевые слова* Python, такие как **`False`**, **`while`** или **`if`**, чтобы не сбивать интерпретатор с толку. В последующих главах вы познакомитесь с их предназначением, а пока просто приведем их список.

Эти правила касаются только Python; в других языках программирования могут быть свои правила

False	as	continue	else	from	in	not	return	yield
None	assert	def	except	global	is	or	try	
True	break	del	finally	if	lambda	pass	while	
and	class	elif	for	import	nonlocal	raise	with	

Интерпретатор Python выдаст ошибку или предупреждение, если встретит одно из этих слов в качестве имени переменной

Не бойтесь задавать вопросы

В: Что такое ключевое слово?

О: Это слово, которое Python зарезервировал для собственных нужд. Такие слова являются частью синтаксиса языка, поэтому использовать их в качестве имен переменных не допускается.

В: А что если ключевое слово является лишь частью имени переменной? Можно ли, например, назвать переменную `if_only` (имя содержит ключевое слово `if`)?

О: Конечно же можно. Имя не должно *в точности* повторять ключевое слово Python. Желательно также не использовать имен, которые можно было бы спутать с ключевым словом, например **`elze`**, напоминающее **`else`**.

В: Различает ли Python имена `myvariable` и `MyVariable`?

О: В Python это имена разных переменных. Python *чувствителен к регистру символов*. Другими словами, прописная и строчная буквы считаются разными. Подобное поведение характерно для большинства современных языков программирования.

В: Есть ли какие-то соглашения об именах переменных? Какое имя предпочтительнее использовать для переменной: `myVar`, `MyVar` или `my_var`?

О: Да, такие соглашения существуют. Имена переменных принято набирать символами нижнего регистра. Если имя содержит несколько слов, то они разделяются символами подчеркивания, например **`max_speed`**, **`height`** или **`super_turbo_mode`**. Как будет показано далее, подобные соглашения существуют не только в отношении имен переменных. Кроме того, эти соглашения специфичны для Python. В других языках свои традиции.

В: Но как подобрать для переменной хорошее имя? И важно ли это?

О: Интерпретатору Python абсолютно все равно, какое у переменной имя, если оно соответствует требованиям синтаксиса. Имена переменных важны для программистов. Старайтесь выбирать простые и понятные имена, чтобы программный код был более читабельным. Слишком короткие имена неудобны, равно как и чересчур длинные. В целом называйте переменные так, как вам удобно. Не обязательно называть переменную **`num`**, если можно использовать более понятное имя **`number_of_hotdogs`**.

Этап 3: вычисление возраста собаки

Мы реализовали два этапа псевдокода, пора переходить к третьему. В данном случае нам нужно просто умножить биологический возраст собаки на 7, чтобы получить ее возраст по человеческим меркам. Задача кажется очень простой, поэтому выполним быструю проверку в оболочке Python.

Давайте вернемся к более важным вещам. У нас программа не закончена.

Джексон, 9 лет

Оболочка Python

```
>>> dog_age = 12
>>> human_age = dog_age * 7
>>> print(human_age)
84
>>>
```

Сначала определим переменную dog_age и присвоим ей значение 12

Теперь умножим ее на 7 и присвоим результат новой переменной human_age

Выведем значение human_age

Получили 84. Отлично, как раз то, что нужно!

Псевдокод калькулятора возраста собаки

1. **Спросить у пользователя имя собаки.**
2. **Спросить у пользователя возраст собаки.**
3. **Умножить возраст собаки на 7, чтобы получить возраст по человеческим меркам.**
4. **Вывести на экран:**

   "Вашей собаке"
   имя собаки
   "сейчас"
   человеческий возраст собаки
   "по меркам людей"

Мы сейчас здесь

Возьмите карандаш

Вот код нашего калькулятора, написанный на данный момент. Добавьте к нему код вычисления возраста по человеческим меркам, используя в качестве образца приведенный выше проверочный код.

```
dog_name = input("Как зовут вашу собаку? ")
dog_age = input("Сколько лет вашей собаке? ")
```

Запишите здесь свой код, а затем сверьтесь с ответом, прежде чем двигаться дальше.

Давайте протестируем код, написанный на третьем этапе. Сохраните программу в файле *dogcalc.py* и выполните команду **Run > Run Module**. После этого перейдите в оболочку Python, введите имя и биологический возраст собаки и убедитесь, что программа возвращает правильный результат.

Вот наш код.

```
dog_name = input("Как зовут вашу собаку? ")
dog_age = input("Сколько лет вашей собаке? ")
human_age = dog_age * 7
print(human_age)
```

Добавьте новый код в свой файл

По всей книге серая заливка фона обозначает новый код

Можно ли вывести `dog_age * 7` *напрямую? Например,* `print(dog_age * 7)`*? Попробуйте*

У нас проблема!

Вы получили то же, что и мы? Мы ввели имя `Тузик` и указали возраст `12`, рассчитывая получить результат `84`, но вместо этого программа выдала `12121212121212`! Что за ерунда?! Где же мы ошиблись? Проверили все еще раз – вроде бы все нормально. Почему же тогда результат отличается от того, что мы получили при построчном выполнении программы в командной строке?

Оболочка Python

```
Как зовут вашу собаку? Тузик
Сколько лет вашей собаке? 12
12121212121212
>>>
```

Тут явно что-то не так!

Взгляните еще раз на число `12121212121212`. Как вам кажется, каким образом оно было получено в Python?

Подсказка: сколько раз в нем повторяется число 12?

Неизбежность ошибок

Поиск и исправление ошибок – это неотъемлемая часть программирования. Независимо от того, насколько хорошо составлен алгоритм (псевдокод), процесс написания исходного кода программ всегда будет сопровождаться возникновением ошибок. Для программиста **отладка** – рутинная повседневная операция. Что такое отладка? Это процесс поиска ошибок в программе. Неформально такие ошибки называют *багами*.

В процессе написания программ вы будете сталкиваться с ошибками трех типов.

Термин баг (от англ. "bug" — жучок) возник из-за того, что одна из первых программных ошибок была вызвана жучком, застрявшим между контактами реле вычислительной машины. В результате процесс выявления программных ошибок стал называться debugging (отладка). См. https://ru.wikipedia.org/wiki/Программная_ошибка

Синтаксические ошибки

На возникновение синтаксической ошибки указывает сообщение `'SyntaxError'`, выводимое интерпретатором Python. **Синтаксические ошибки подобны грамматическим ошибкам в диктанте**. Они возникают при неправильном написании ключевых слов или программных конструкций Python. Исправлять синтаксические ошибки проще всего — достаточно определить строку, в которой возникла ошибка, и проанализировать ее синтаксис.

Ошибки выполнения

Ошибки выполнения возникают, когда Python не может выполнить синтаксически корректную программу. Такое может произойти, если в программе случайным образом происходит деление на нуль (недопустимая математическая операция в любом языке). Чтобы исправить ошибку такого типа, нужно отследить, какая операция приводит к ее возникновению.

Семантические (смысловые) ошибки

Семантические ошибки еще называют логическими ошибками. Программа, содержащая такого рода ошибки, будет прекрасно выполняться — интерпретатор не будет обнаруживать в ней ни синтаксических ошибок, ни ошибок выполнения, но она не будет выдавать ожидаемый результат. **Чаще всего причиной семантических ошибок является неправильная реализация алгоритма в коде**. Вам кажется, будто вы все правильно сформулировали, но в действительности это не так. Находить и исправлять семантические ошибки сложнее всего.

Сила мысли

Наш калькулятор возраста собаки возвращает ошибочный результат — 12121212121212 вместо 84. Какого типа это ошибка?

- ☐ А. Ошибка выполнения
- ☐ Б. Синтаксическая ошибка
- ☐ В. Семантическая ошибка
- ☐ Г. Ошибка вычисления
- ☐ Д. Ничего из перечисленного
- ☐ Е. Все перечисленное

Ответ Б, потому что в программе нет синтаксических ошибок и ошибок выполнения.

Я заметил, что в полученном результате семь раз повторяется число 12, т.е. каким-то образом, когда мы умножаем 12 на 7, Python вместо умножения просто повторяет число 12 семь раз. Немного поэкспериментировав в оболочке Python, я выяснил, что, когда я умножаю **строку** `"12"` на `7`, я получаю результат `12121212121212`. А вот если я умножаю **число** `12` на `7`, то получаю ожидаемый результат — `84`.

Прекрасная отладка! Причина, по которой мы получили 12121212121212 вместо 84, именно в том и заключается, что интерпретатор Python распознал возраст собаки как строку, а не как число. С чем это связано? Давайте разберемся, как в Python выполняется операция умножения.

Еще немного отладки

Попробуем поэкспериментировать с отладкой и посмотрим, что происходит. Для этого выполним несколько команд в оболочке Python.

Оболочка Python

```
>>> '3' * 7
'3333333'
>>> 3 * 7
21
>>> 'ice cream ' * 3
'ice cream ice cream ice cream'
>>> num = input('Введите число: ')
Введите число: 12
>>> num
'12'
>>> num * 7
'12121212121212'
>>>
```

Заметьте, что это строка

А это число

Как уже говорилось, если Python умножает строку на число, то строка повторяется соответствующее число раз

Если Python умножает два числа, то получаем произведение

Мы можем умножить ЛЮБУЮ строку на число, получив в результате повторяющуюся строку

Полученная строка содержит три повторения строки `'ice cream'`

Давайте получим от пользователя число с помощью функции `input()`

Похоже, что функция `input()` возвращает строку

Умножаем на 7

Все верно

Почему Python воспринимает число, введенное с помощью функции `input()`, как строку? Ведь мы же ввели цифры, а не буквы. Причина заключается в том, что функция `input()` всегда возвращает строковое значение. Откуда это известно? Не только из экспериментов с командной строкой. Есть и более непосредственный способ узнать синтаксис функции. Прежде всего, можно заглянуть в спецификацию Python, где сказано следующее:

> **`input(`*приглашение*`)`** **После вывода приглашения функция считывает вводимые данные, преобразует их в строку и возвращает эту строку.**

Это определение из спецификации Python (мы чуть сократили его)

Есть также способы проверить тип значения прямо в программе. Подробнее обо всем этом мы поговорим позже, а пока что констатируем, что нам удалось определить источник проблемы: *функция `input()` возвращает строку, тогда как нам нужно число.*

Почему все так сложно? Неужели Python не способен понять, что если я умножаю число на число в виде строки, то хочу выполнить **операцию умножения**, а вовсе не повторить строку сколько-то там раз? Что это за глупости?

Запросы нужно формулировать корректно...

Конечно, Python мог бы принять решение за нас и посчитать вводимую строку числом, но тогда мы не будем застрахованы от сюрпризов другого рода, когда нам понадобится действительно передать число в строковом виде. В том-то и проблема: интерпретатор не может всякий раз угадывать наши мысли, поэтому он старается не делать НИКАКИХ предположений. Он следует вашим инструкциям буквально: если инструкция сообщает, что введенное значение – строка, то он и трактует его как строку. И наоборот: если инструкция возвращает число, то Python работает с числом. Чего Python не будет делать никогда, так это *угадывать* тип передаваемых данных, поскольку в большинстве случаев такие догадки окажутся неправильными, что лишь усложнит отладку.

Чтобы стать программистом, вам следует научиться правильно определять типы данных, с которыми вы имеете дело. Это означает, что нам следует поближе познакомиться с типами данных.

Типы данных Python

Любые данные в Python относятся к тому или иному типу. **Тип данных** очень важен, поскольку от него зависит, какие операции будут выполняться с данными. Как вы могли убедиться на примере операции умножения, результат будет совершенно разным при умножении строки на число и числа на число. Следовательно, при написании программы необходимо четко понимать, с данными какого типа вы имеете дело.

Некоторые типы данных вам уже знакомы. Вы знаете, что такое **строки**, представляющие собой последовательности символов. Нам также встречались и **числовые типы**. В Python поддерживаются два основных числовых типа: **целые числа** и **числа с плавающей точкой**. Рассмотрим их подробнее.

В Python есть и другие типы данных, в частности, **булевы значения**, **списки**, **словари** и **объекты**, о которых мы поговорим в следующих главах.

Не бойтесь задавать вопросы

В: Какие еще операции можно выполнять над строковыми значениями, кроме конкатенации и вывода на экран?

О: Не стоит недооценивать возможности простых типов. Обработка строк лежит в основе современных вычислений (Google и Facebook не дадут соврать). По мере изучения Python вы узнаете о том, как осуществлять поиск в строках, менять их, форматировать и выполнять множество других интересных операций.

В: При выполнении операций с числами разных типов могут ли возникать те же проблемы, как в случае чисел и строк?

О: Нет. Если в выражении встречаются целые числа и числа с плавающей точкой, то Python преобразует все числа в формат с плавающей точкой.

В: Можно ли сначала присвоить переменной целочисленное значение, а затем — строку?

О: В Python такое вполне допускается. Помните: тип есть у значения, но не у переменной. В переменную можно записывать любые значения, даже если при этом меняется их тип. Впрочем, это не считается хорошей практикой программирования, ведь программу, в которой переменные меняют свой тип, труднее понимать.

По-серьезному

В Python имеются средства для работы с научными данными. В частности, поддерживаются комплексные числа и экспоненциальный способ записи числа.

Исправление ошибки

Итак, мы выяснили, что причиной ошибки является неправильное предположение о типе данных результата, возвращаемого функцией `input()`. Мы считали, что если в ответ на запрос мы вводили число, то функция должна была возвращать число. Однако отладка показала, что функция всегда возвращает строку. Впрочем, беспокоиться тут совершенно не о чем. Потому-то мы и выполняем отладку на каждом шаге — чтобы выявлять вот такие ошибки. К счастью, в Python строковое значение можно легко преобразовать в число.

Вот как это делается.

1. Сначала создадим строковое представление числа.

```
answer = '42'
```

Переменной answer присваивается строка '42'

2. Теперь воспользуемся функцией `int()`, передав ей строку в качестве аргумента.

```
answer = int('42')
```

Ух ты, я теперь целое!

Здесь строка '42' преобразуется в целое число, которое присваивается переменной answer

Не бойтесь задавать вопросы

В: Что произойдет, если передать функции `int()` строку, не содержащую число, например `int('hi')`?

О: В подобной ситуации возникнет ошибка выполнения. Интерпретатор сообщит о том, что значение не является числом.

В: Будет ли функция `int()` корректно обрабатывать числа с плавающей точкой, например `int('3.14')`?

О: Функция `int()` работает только с целыми числами (точнее, со строками, содержащими целые числа). Если нужно преобразовать строку, содержащую число с плавающей точкой, например `'3.14'`, воспользуйтесь функцией `float()`: `float('3.14')`.

Возьмите карандаш

Итак, мы знаем, что ошибка вызвана предположением, будто переменная `dog_age` содержит числовое значение, тогда как Python воспринимает данную переменную как строку. Добавьте в приведенный ниже код функцию `int()`, чтобы устранить ошибку.

Есть несколько вариантов решения проблемы, поэтому сверьтесь с ответом в конце главы

```
dog_name = input("Как зовут вашу собаку? ")
dog_age = input("Сколько лет вашей собаке? ")
human_age = dog_age * 7
print(human_age)
```

Тест-драйв

Разобравшись с принципами преобразования строковых значений в целочисленные (см. предыдущее упражнение), давайте внесем необходимые изменения в код программы. Добавьте выделенные ниже элементы в код программы, чтобы получить целочисленный возраст собаки, после чего сохраните программу и выполните команду **Run > Run Module**. Затем перейдите к консоли, введите имя и биологический возраст собаки и убедитесь, что теперь программа возвращает правильный результат.

Вот наш код (изменения выделены).

Мы добавили функцию `int()`, *чтобы преобразовать значение* `dog_age` *в целое число, прежде чем умножать его на 7*

```
dog_name = input("Как зовут вашу собаку? ")
dog_age = input("Сколько лет вашей собаке? ")
human_age = int(dog_age) * 7
print(human_age)
```

Ура, заработало!

Наконец-то программа возвращает ожидаемый результат – 84. Проверьте работу калькулятора, подставив в него другие числовые значения. Нам осталось реализовать завершающий этап псевдокода, и программа будет готова.

Оболочка Python

```
Как зовут вашу собаку? Тузик
Сколько лет вашей собаке? 12
84
>>>
```

Вот что мы получили

Внимание!

Если программа так и не заработала...

Как вы уже знаете, работе программы могут мешать синтаксические и семантические ошибки, а также ошибки выполнения. Получив сообщение `invalid syntax` от интерпретатора Python, проверьте исходный код на наличие синтаксических ошибок (например, правильность расстановки скобок). Если проблема не выявлена, попробуйте просмотреть программу наоборот — из конца в начало — или попросите друга, разбирающегося в программировании, сверить код с тем, который приведен в книге. Убедитесь в том, что функция `int()` набрана правильно и вы не забыли поставить скобки. Вы всегда можете скачать исходный код с сайта книги (`http://wickedlysmart.com/hflearntocode`) и сравнить его со своим.

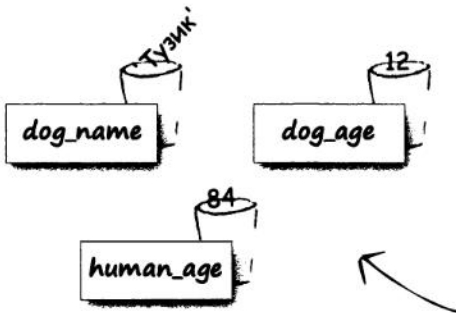
Этап 4: Вывод результата

Вот мы и добрались до последнего этапа псевдокода. Мы получили все необходимые данные от пользователя и вычислили возраст собаки по человеческим меркам. Теперь осталось вывести полученный результат. Для этого нужно воспользоваться функцией `print()` и отобразить на экране текущие значения программных переменных.

Псевдокод калькулятора возраста собаки

1. **Спросить у пользователя имя собаки.**
2. **Спросить у пользователя возраст собаки.**
3. **Умножить возраст собаки на 7, чтобы получить возраст по человеческим меркам.**
4. **Вывести на экран:**

   "Вашей собаке"
   имя собаки
   "сейчас"
   человеческий возраст собаки
   "по меркам людей"



Теперь у нас есть все значения, чтобы завершить последний этап псевдокода

В псевдокоде указано, что конкретно должно быть выведено. Но прежде чем писать код, давайте рассмотрим, как эффективно применять функцию `print()`. До этого мы всегда передавали ей единственный *аргумент*.

Значения, передаваемые функции, называются аргументами

```
print('Привет!')
```

или

```
print(42)
```

Пока что мы передавали функции `print()` только один аргумент: строку или число

или

```
print('До' + ' свидания!')
```

Мы также применяли конкатенацию для формирования строкового аргумента функции `print()`, но это был одиночный аргумент

Однако у функции `print()` может быть множество аргументов.

Функции `print()` можно передавать сколько угодно аргументов, разделяя их запятыми

При наличии нескольких аргументов функция `print()` разделяет их одиночными пробелами

```
print('Привет!', 42, 3.7, 'До свидания!')
```

Аргумент — это замысловатый термин, которым обозначают значения, передаваемые функции

Оболочка Python

```
Привет! 42 3.7 До свидания!
>>>
```

Возьмите карандаш

Вооружившись знаниями о возможностях функции `print()`, напишите код для вывода нужного нам результата.

Вывести на экран:
"Вашей собаке"
имя собаки
"сейчас"
человеческий возраст собаки
"по меркам людей"

Вот что вам нужно вывести

Финальный тест-драйв

Итак, мы реализовали последний этап псевдокода. Внесите в программу необходимые изменения, сохраните ее и выполните команду **Run > Run Module**. После этого перейдите в оболочку Python и проверьте работу калькулятора.

Ниже приведен его полный код.

```
dog_name = input("Как зовут вашу собаку? ")
dog_age = input("Сколько лет вашей собаке? ")
human_age = int(dog_age) * 7
print(human_age)
print('Вашей собаке',
      dog_name,
      'сейчас',
      human_age,
      'по меркам людей')
```

Удалите этот код

Добавьте этот код в свой файл dogcalc.py

Полученные результаты

```
Оболочка Python
Как зовут вашу собаку? Тузик
Сколько лет вашей собаке? 12
Вашей собаке Тузик сейчас 84 по меркам людей
>>>
```

Как мы и хотели!

Тренируем мозг

Глава почти завершена — осталось только подвести итоги и решить кроссворд. Почему бы напоследок еще немного не потренировать мозг?

Итак, вот наше последнее упражнение. Предположим, имеются две переменные: `first` и `last`. Сможете ли вы поменять местами их значения? Напишите соответствующий код. Решение приведено в конце главы.

```
first = 'где-то'
last = 'над радугой'
print(first, last)
```

Запишите здесь свой код

```
print(first, last)
```

Вот результат, который вы должны получить

Оболочка Python

```
где-то над радугой
над радугой где-то
>>>
```

СИЛА МЫСЛИ

Что произойдет, если в операции присваивания переменной `dog_name` не взять имя собаки в кавычки? Чему будет равна переменная?

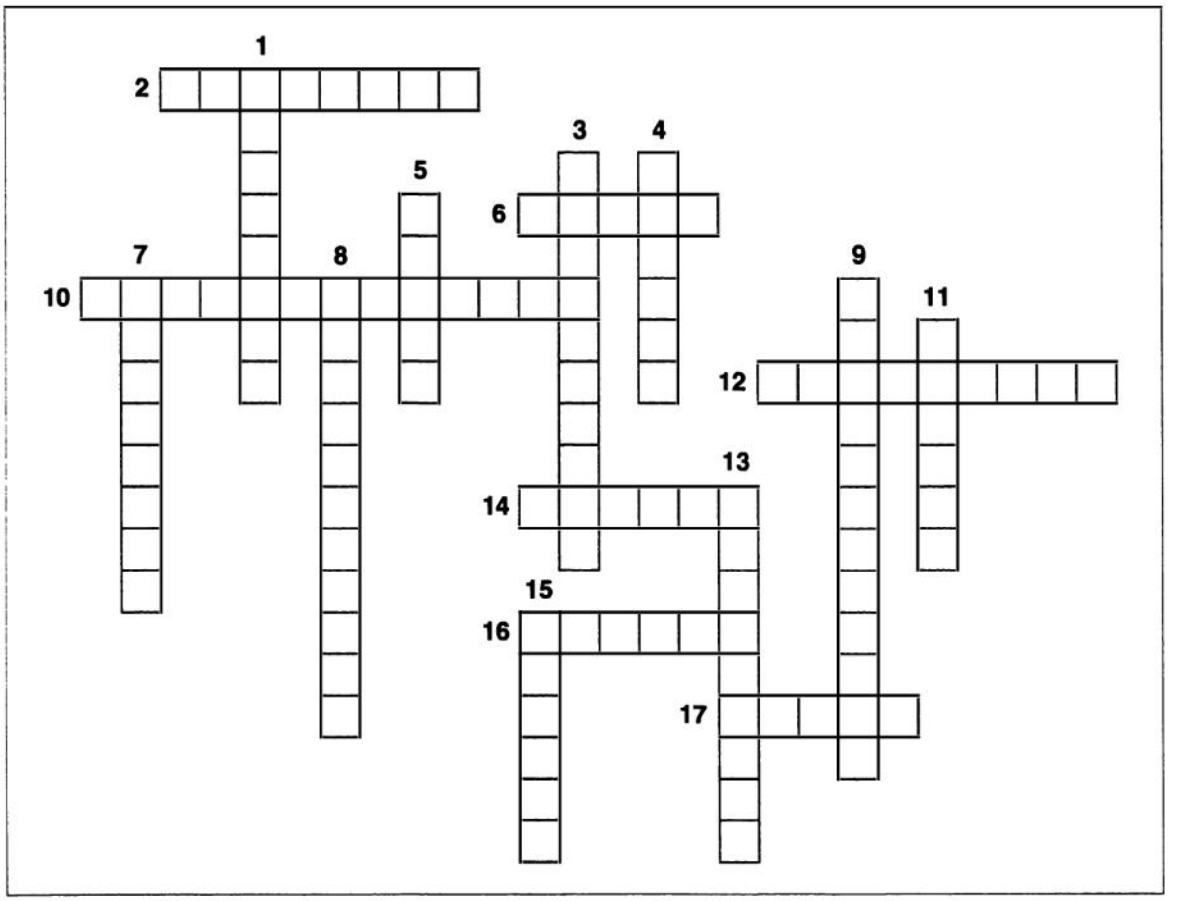
```
dog_name = Тузик
```

САМОЕ ГЛАВНОЕ

- Компьютеры умеют делать две вещи: хранить значения и выполнять над ними операции.
- Составление псевдокода — отличный способ начать разработку программы.
- Синтаксис задает правила написания инструкций Python, а семантика определяет назначение инструкций.
- Файл, в котором хранится программный код Python, называется *модулем*.
- В оболочке Python можно выполнять одиночные инструкции, проверяя их работу.
- Функция `input()` запрашивает данные у пользователя и возвращает их в программу в виде строкового значения.
- Функции возвращают значения в качестве результата.
- Значение можно сохранить для будущего использования, присвоив его переменной.
- Когда значение присваивается переменной, Python сохраняет значение в памяти, помечая его с помощью имени переменной.
- Значение, хранящееся в переменной, можно менять.
- В Python используются простые правила именования переменных. В качестве имени не допускается использовать ключевые слова.
- Ключевые слова зарезервированы в синтаксисе языка программирования.
- Строка — это тип данных Python, состоящий из последовательности символов.
- В Python поддерживаются два типа чисел: целые и с плавающей точкой. Есть также ряд дополнительных типов, предназначенных для научных расчетов.
- Функция `int()` преобразует строку в целое число, а функция `float()` преобразует строку в число с плавающей точкой.
- Для раннего выявления ошибок в программе необходимо тестировать код.
- Существуют три основных вида ошибок: синтаксические, семантические и ошибки выполнения.
- Если умножить строку на число *n*, то Python повторит строку *n* раз.
- Строки можно объединять с помощью оператора конкатенации.
- В Python оператором конкатенации служит знак "плюс" (+).
- Значения, передаваемые функции, называются *аргументами*.

Кроссворд

Пора немного отвлечься. Перед вами кроссворд, состоящий из слов, которые встречались на протяжении главы 2.



По горизонтали

2. То, что присваивается переменной
6. Один из числовых типов
10. Специальный символ, часто используемый в именах переменных
12. Устанавливает правила написания инструкций
14. То, что выводит функция `input()`
16. Текстовый тип данных
17. Имя нашей собаки

По вертикали

1. То, что передается функции
3. В ней хранится значение
4. Программный файл Python
5. Знак присваивания
7. Утилита для ввода команд Python
8. Программа, вычисляющая возраст собаки
9. Операция объединения строк
11. Место, где хранятся значения переменных
13. Описывает назначение инструкций
15. Еще один наш пес

Возьмите карандаш
Решение

Итак, нам предстоит написать псевдокод. Для начала подумайте, как бы вы описали алгоритм, или рецепт, вычисления возраста собаки по человеческим меркам. Далее запишите все свои соображения в виде простых текстовых команд. Программа должна взаимодействовать с пользователем, спрашивая у него имя и биологический возраст собаки. В конце необходимо вывести что-то осмысленное, например: "Вашей собаке Рокки сейчас 72 по меркам людей".

В общем, просто запишите псевдокод на своем родном языке.

Спросить у пользователя имя собаки.

Спросить у пользователя возраст собаки.

Сначала нам нужно получить информацию от пользователя

Умножить возраст собаки на 7, чтобы получить возраст по человеческим меркам.

Затем нам нужно вычислить возраст собаки по человеческим меркам

Вывести на экран:

"Вашей собаке"

имя собаки

"сейчас"

человеческий возраст собаки

"по меркам людей"

Наконец, необходимо вывести результаты пользователю

Смайли, 1 год

Филя, 5 лет

Возьмите карандаш
Решение

На первом этапе необходимо пошагово проанализировать составленный ранее алгоритм и постараться описать, что конкретно должна делать программа в каждом случае.

Псевдокод калькулятора возраста собаки

1. **Спросить у пользователя имя собаки.**

2. **Спросить у пользователя возраст собаки.**

3. **Умножить возраст собаки на 7, чтобы получить возраст по человеческим меркам.**

4. **Вывести на экран:**

   "Вашей собаке"
   имя собаки
   "сейчас"
   человеческий возраст собаки
   "по меркам людей"

1. *Сообщаем пользователю о том, что нужно ввести имя собаки, и просим его ввести строку. Это имя придется где-то сохранить, т.к. оно понадобится в п. 4.*

2. *Сообщаем пользователю о том, что нужно ввести возраст собаки, и просим его ввести число. Это число тоже придется сохранить, т.к. оно понадобится в п. 3.*

3. *Берем число, введенное в п. 2, и умножаем его на 7. Полученное значение тоже нужно где-то сохранить, чтобы его можно было использовать в п. 4.*

4. *Сначала выводим "Вашей собаке", затем — имя, полученное в п. 1, далее — строку "сейчас", после этого — число, полученное в п. 3, и в конце — "по меркам людей".*

УПРАЖНЕНИЕ (РЕШЕНИЕ)

Теперь ваша очередь. Напишите код для получения биологического возраста собаки с помощью функции `input()`. Выдайте пользователю запрос `"Сколько лет вашей собаке?"` и сохраните результат в переменной `dog_age`. Сделайте пометки, уточняющие назначение каждого элемента кода.

```
dog_age = input("Сколько лет вашей собаке? ")
```

Вот наша новая переменная `dog_age`

А вот вызов функции `input()`*, в которой мы просим пользователя указать возраст собаки, после чего сохраняем полученное значение в переменной* `dog_age`

Кто я? Решение

Множество выражений Python потеряло свои значения. Не могли бы вы помочь их восстановить? Соедините каждое из выражений слева с соответствующим значением справа. Будьте бдительны — некоторые значения лишние.

Выражения	Значения
'kit e' + ' ' + 'cat'	1
(14 – 9) * 3	2
3.14159265 * 3**2	15
42	21 — Обманщик!
'h' + 'e' + 'l' + 'l' + 'o'	28.27433385
8 % 3	42
7 – 2 * 3	–13 — Обманщик!
(7 – 2) * 3	'kit e cat'
	'hello'

УПРАЖНЕНИЕ (РЕШЕНИЕ)

В игре в наперстки используются три непрозрачных стаканчика и небольшой шарик. Для программной реализации игры нам понадобится несколько переменных и значений. Используя все свои знания о переменных, значениях и операциях присваивания, попробуйте сыграть в игру. Просмотрите код и определите переменную, которая после выполнения последней операции будет содержать значение 1. В каком стаканчике окажется шарик: 1, 2 или 3? Делайте ставки, господа!

```
cup1 = 0                cup1 = 0
cup2 = 1                cup1 = 0, cup2 = 1
cup3 = 0                cup1 = 0, cup2 = 1, cup3 = 0
cup1 = cup1 + 1         cup1 = 1, cup2 = 1, cup3 = 0
cup2 = cup1 - 1         cup1 = 1, cup2 = 0, cup3 = 0
cup3 = cup1             cup1 = 1, cup2 = 0, cup3 = 1
cup1 = cup1 * 0         cup1 = 0, cup2 = 0, cup3 = 1
cup2 = cup3             cup1 = 0, cup2 = 1, cup3 = 1
cup3 = cup1             cup1 = 0, cup2 = 1, cup3 = 0
cup1 = cup2 % 1         cup1 = 0, cup2 = 1, cup3 = 0
cup3 = cup2             cup1 = 0, cup2 = 1, cup3 = 1
cup2 = cup3 - cup3      cup1 = 0, cup2 = 0, cup3 = 1
```

Победитель!

Взлом Кода: решение

Сдавая экзамен в разведшколе, вам предстоит взломать пароль, который закодирован в программе. Мысленно проанализируйте программу и попытайтесь понять, каким будет пароль. Только не спешите: у разведчика нет права на ошибку. Желаем удачи!

```
word1 = 'ox'
word2 = 'owl'
word3 = 'cow'
word4 = 'sheep'
word5 = 'flies'
word6 = 'trots'
word7 = 'runs'
word8 = 'blue'
word9 = 'red'
word10 = 'yellow'
word9 = 'The ' + word9
passcode = word8
passcode = word9
passcode = passcode + ' f'
passcode = passcode + word1
passcode = passcode + ' '
passcode = passcode + word6
print(passcode)
```

```
word9 = 'The red'
passcode = 'blue'
passcode = 'The red'
passcode = 'The red f'
passcode = 'The red fox'
passcode = 'The red fox '
passcode = 'The red fox trots'
```

Оболочка Python

```
The red fox trots
>>>
```

Если выполнить этот код, то вот что вы получите

The red fox trots

Наш пароль!

Возьмите карандаш
Решение

Вот код нашего калькулятора, написанный на данный момент. Добавьте к нему код вычисления возраста по человеческим меркам, используя в качестве образца приведенный выше проверочный код.

```
dog_name = input("Как зовут вашу собаку? ")
dog_age = input("Сколько лет вашей собаке? ")
human_age = dog_age * 7
print(human_age)
```

По всей книге серая заливка фона обозначает новый код

Берем значение dog_age, введенное пользователем, и умножаем его на 7, присваивая результат переменной human_age

В конце выводим значение human_age, чтобы увидеть, каким получился результат

Возьмите карандаш
Решение

Итак, мы знаем, что ошибка вызвана предположением, будто переменная `dog_age` содержит числовое значение, тогда как Python воспринимает данную переменную как строку. Добавьте в приведенный ниже код функцию `int()`, чтобы устранить ошибку.

Существует несколько способов добавления функции `int()`. Вот несколько из них:

```
dog_name = input("Как зовут вашу собаку? ")
dog_age = input("Сколько лет вашей собаке? ")
dog_age = int(dog_age)
human_age = dog_age * 7
print(human_age)
```

Сначала получаем значение `dog_age` в виде строки, после чего применяем к нему функцию `int()` и повторно присваиваем результат переменной `dog_age`

```
dog_name = input("Как зовут вашу собаку? ")
dog_age = input("Сколько лет вашей собаке? ")
human_age = int(dog_age) * 7
print(human_age)
```

Функцию `int()` можно применить именно тогда, когда нам требуется целочисленное значение `dog_age`

```
dog_name = input("Как зовут вашу собаку? ")
dog_age = int(input("Сколько лет вашей собаке? "))
human_age = dog_age * 7
print(human_age)
```

Наконец, функцией `int()` можно обернуть вызов функции `input()`. В этом случае переменной `dog_age` всегда присваивается целое число

Возможно, все эти вызовы функций слегка сбили вас с толку. В последующих главах мы поговорим о том, как работать с функциями. А пока достаточно знать, что функция `int()` получает строку и преобразует ее в число.

Возьмите карандаш
Решение

Вооружившись знаниями о возможностях функции `print()`, напишите код для вывода нужного нам результата.

Вот что вам нужно вывести

Вывести на экран:
"Вашей собаке"
имя собаки
"сейчас"
человеческий возраст собаки
"по меркам людей"

```
print('Вашей собаке',
      dog_name,
      'сейчас',
      human_age,
      'по меркам людей')
```

Мы собрали все нужные нам аргументы функции `print()`, разделив их запятыми

Аргументы функции можно выстраивать в столбик для повышения наглядности кода

Тренируем мозг: решение

Глава почти завершена — осталось только подвести итоги и решить кроссворд. Почему бы напоследок еще немного не потренировать мозг?

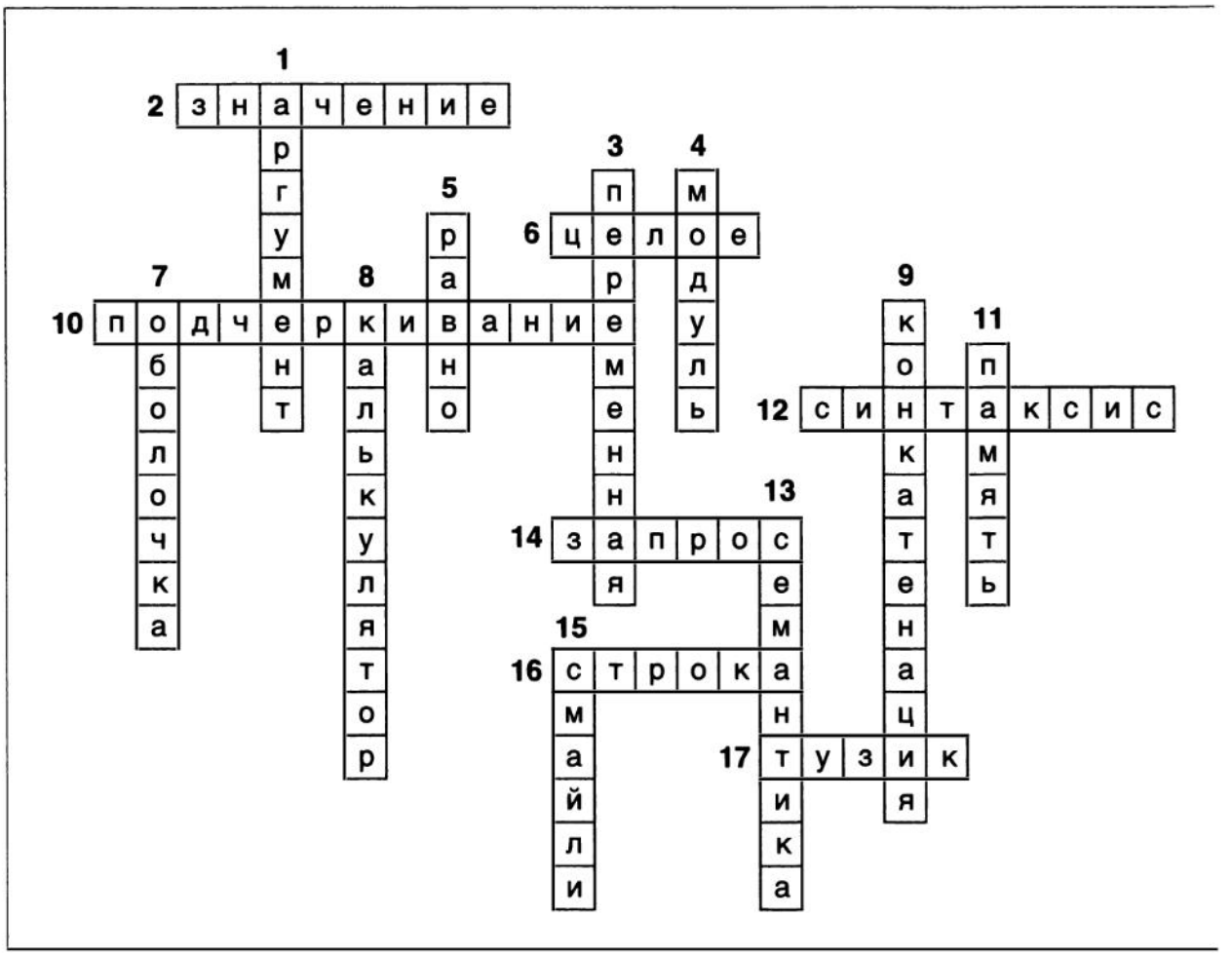
Итак, вот наше последнее упражнение. Предположим, имеются две переменные: `first` и `last`. Сможете ли вы поменять местами их значения? Напишите соответствующий код.

```
first = 'где-то'
last = 'над радугой'
print(first, last)
temp = first
first = last
last = temp
print(first, last)
```

Распространенный прием — использовать временную переменную, в которой сохраняется значение первого элемента. После этого первый элемент заменяется вторым, а во второй копируется значение из временной переменной

Проследите за изменением переменных, чтобы понять, как это работает

Кроссворд: решение



3 Булевы значения, условия и циклы

Принятие решений

Думаю, вы заметили, что наши предыдущие программы были не особо интересными. Содержащийся в них код представлял собой простую последовательность инструкций, выполняемых интерпретатором **сверху вниз**. Никаких тебе поворотов сюжета, внезапных изменений, сюрпризов — в общем, ничего оригинального. Чтобы программа могла стать более интересной, она нуждается в средствах **принятия решений**, позволяющих **управлять ходом ее выполнения** и **многократно повторять** одни и те же действия. Этим мы и займемся в данной главе. По ходу дела вы научитесь играть в загадочную игру шоушилин, познакомитесь с персонажем по имени Буль и увидите, насколько полезным может быть тип данных, содержащий всего два возможных значения. Вы также узнаете, как побороть пресловутый **бесконечный цикл**. Что ж, начнем!

Ради интереса мы даже создадим такой цикл

Сыграем в игру?

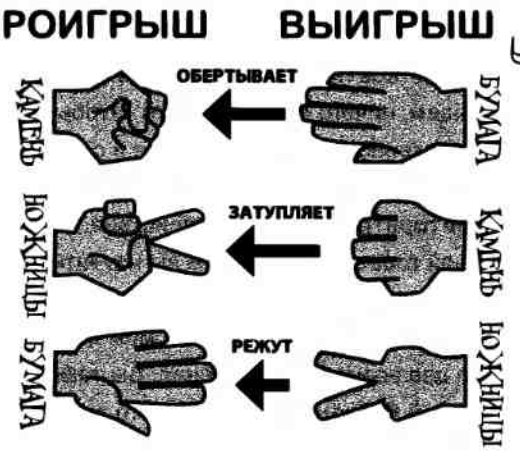
Игра под названием *шоушилин* появилась еще в эпоху древнекитайской династии Хань. С ее помощью регулировали судебные споры, заключали сделки, кидали жребий...

Сегодня мы знаем эту игру как "Камень, ножницы, бумага". Именно ее нам предстоит реализовать, что даст вам возможность сыграть против очень серьезного противника: *вашего компьютера.*

В игре "Камень, ножницы, бумага" используются три жеста, называемых знаками

Правила игры в „Камень, ножницы, бумага"

Для тех, кто еще не знаком с игрой, мы опишем ее простые правила. Если же вам доводилось играть в нее, то не помешает освежить правила в памяти. В игре участвуют двое, каждый из которых заранее выбирает один из трех знаков: камень, ножницы или бумага. Игроки считают вслух до трех, после чего одновременно показывают сделанный выбор. Победитель определяется по следующей схеме.



Если один игрок выбирает ножницы, а другой — камень, то камень затупляет ножницы и побеждает

Если один игрок выбирает бумагу, а другой — камень, то бумага обертывает камень и побеждает

Если оба игрока делают одинаковый выбор, тогда ничья

Если один игрок выбирает бумагу, а другой — ножницы, то ножницы режут бумагу и побеждают

СИЛА МЫСЛИ

Если вы еще не играли в "Камень, ножницы, бумага", то обязательно сыграйте с кем-то из друзей. Даже если у вас есть опыт, позовите друга, чтобы освежить игру в памяти.

Как играть против компьютера

Поскольку у компьютера нет рук, нам придется играть чуть по-другому. Мы попросим компьютер заранее сделать свой выбор, но не сообщать его нам. После этого мы объявим свой выбор, а компьютер сравнит его со своим и определит победителя.

Нагляднее всего привести пример розыгрыша. Ниже показано окно оболочки Python, в котором было сыграно несколько раундов с тремя возможными исходами: побеждает пользователь, побеждает компьютер и ничья.

Оболочка Python

```
камень, ножницы или бумага? камень
Пользователь выиграл, я выбрал ножницы.
камень, ножницы или бумага? камень
Мы оба выбрали камень, играем снова.
камень, ножницы или бумага? бумага
Компьютер выиграл, я выбрал ножницы.
```

Компьютер просит выбрать камень, ножницы или бумагу, и вы выбираете камень

Компьютер присуждает победу вам

Играем дальше

Ничья, сыграем повторно

На этот раз победил компьютер

Общий алгоритм игры

Первое, что мы сделаем, — составим общий план игры в виде псевдокода. На этот раз мы добавили кое-что новое: блок-схему, на которой схематически представлена логика игры.

1 Пользователь начинает игру.

- **А** Компьютер определяет, каким будет его выбор: камень, ножницы или бумага.

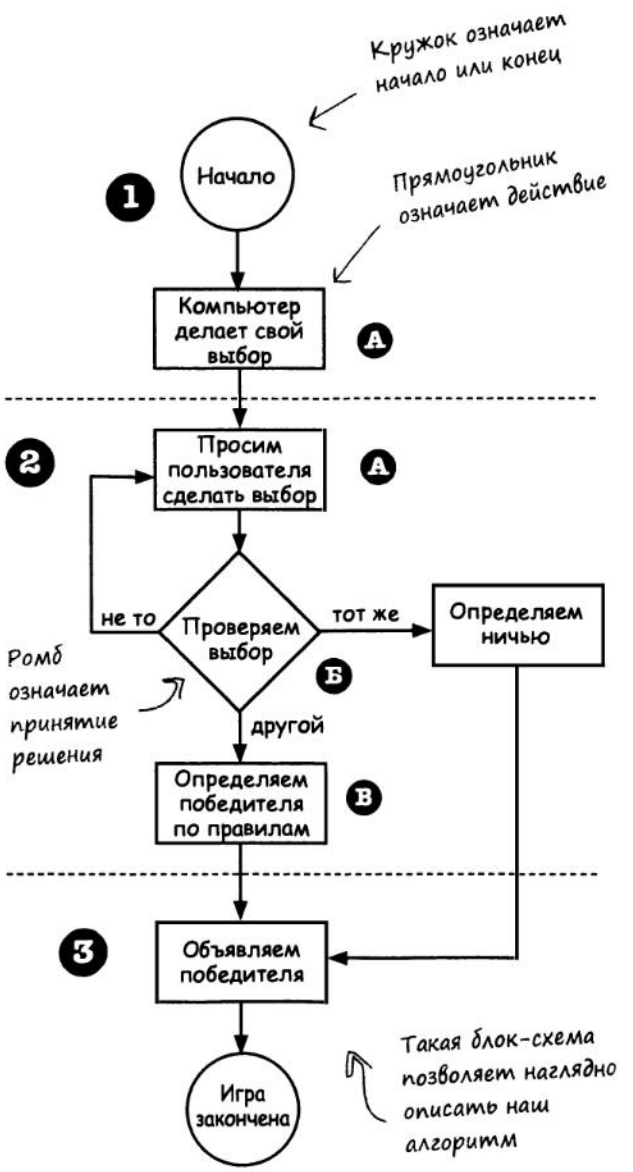
2 Начинается игра.

- **А** Просим пользователя сделать выбор.
- **Б** Проверяем выбор пользователя. Если он неверный (не камень, не ножницы и не бумага), то возвращаемся к п. 2А.

  Если выбрано то же, что и у компьютера, то объявляем ничью и переходим к п. 3.
- **В** Определяем победителя по правилам игры.

3 Игра завершается.

Сообщаем о том, кто победил и что выбрал компьютер.



Теперь у нас есть общее представление о том, что должна делать программа. Далее мы пройдем каждый этап и детально все распланируем.

Компьютер делает свой выбор

Согласно нашему плану первый этап — выбор компьютером выбрасываемого им знака. Другими словами, компьютер должен выбрать камень, ножницы или бумагу. Причем игра станет интереснее, если выбор будет произвольным, чтобы пользователь не мог его предугадать.

Выбор случайных чисел — задача, встречающаяся во многих программах, поэтому во всех языках программирования имеются средства генерирования таких чисел. Рассмотрим, как получить случайное число в Python и как на его основе выбрать произвольный игровой вариант.

Генерирование случайного числа

В Python имеется множество готового кода, который не нужно писать самому. Как правило, такой код распространяется в виде *модуля*, называемого также *библиотекой* (об этом мы еще поговорим в последующих главах). В данном проекте нам понадобится модуль `random`, который нужно *импортировать* в программу с помощью инструкции `import`.

Сначала указываем ключевое слово `import`

За ним указываем имя требуемого модуля, в данном случае `random`

```
import random
```

Обычно инструкции импорта стоят в начале файла, чтобы можно было легко отследить все импортируемые модули

Помните: модуль — это просто файл с кодом Python

После того как модуль `random` импортирован, в программе становятся доступными все содержащиеся в нем функции. Мы используем лишь одну из них.

О функциях и модулях мы поговорим позже, а пока что достаточно знать, что это встроенные инструменты Python

Сначала указывается имя модуля, в данном случае `random`

Затем ставится разделительная точка

Далее указывается имя функции, `randint`

```
random.randint(0, 2)
```

Функция `randint()` вернет 0, 1 или 2

О деталях работы функций мы поговорим позже, а пока что просто используйте функцию

Мы передаем функции `randint()` два числа

Это диапазон, поэтому функция `randint()` вернет целое число в диапазоне 0–2

УПРАЖНЕНИЕ

Проведите небольшой тест. Откройте оболочку Python, импортируйте модуль `random` и сгенерируйте несколько случайных чисел с помощью вызова `random.randint(0, 2)`.

Вызовите функцию `randint()` несколько раз и посмотрите, что она выдаст

Оболочка Python

```
>>> import random
>>> random.randint(0, 2)
```

Что вы получите?

Применение случайных чисел

Итак, мы знаем, как сгенерировать случайное число из набора 0, 1 и 2, которое будет представлять выбор компьютера. Сразу договоримся, что число 0 будет соответствовать камню, 1 — бумаге, а 2 — ножницам. Запишем код генерирования случайного числа.

Не забудьте импортировать модуль `random`

Мы добавили пустую строку, чтобы визуально отделить операцию импорта модуля от основного кода. Это улучшает наглядность больших листингов

```
import random

random_choice = random.randint(0, 2)
```

Сохраняем случайное число в переменной, чтобы его можно было использовать впоследствии

Здесь мы генерируем случайное число, представляющее выбор компьютера

Не бойтесь задавать вопросы

В: Как нам помогут случайные числа?

О: Генерирование случайного числа можно сравнить с броском игральной кости. Разница лишь в том, что у нас всего три возможных варианта (камень, ножницы и бумага), как если бы у кубика было три стороны, а не шесть. Полученное случайное число сопоставляется с выбором компьютера: 0 — камень, 1 — бумага и 2 — ножницы.

В: Почему цепочка случайных чисел начинается с 0? Разве не логичнее было бы использовать числовой ряд 1, 2, 3?

О: Только не в программировании! Программисты привыкли начинать счет с нуля, а не с единицы. Со временем вы поймете, почему это так, а пока просто примите как есть.

В: Действительно ли случайные числа генерируются совершенно случайным образом?

О: Нет, генерируемые компьютером числа будут *псевдослучайными*, т.е. не полностью случайными. Такие числа генерируются по определенному предсказуемому шаблону, которого не существует в случае истинно случайных чисел. Чтобы сгенерировать полностью случайное число, нам придется использовать какое-то природное явление, например радиоактивный распад, что, согласитесь, не слишком удобно с практической точки зрения. Для большинства задач программирования псевдослучайных чисел вполне достаточно.

В: Правильно ли я понимаю, что импорт модуля позволяет получить доступ к коду Python, написанному кем-то другим?

О: Разработчики Python привыкли сохранять полезный код в виде модулей. С помощью инструкции `import` вы сможете использовать этот код в своих проектах. В частности, модуль `random` содержит множество функций, связанных с генерированием случайных чисел. Пока что мы используем только одну из них — `randint()`, а с остальными познакомимся позже.

Чтобы проверить работу программы, сохраните ее в файле *rock.py* и выполните команду **Run > Run Module**.

Вот что мы имеем на данный момент.

Добавьте новый код в свой файл

```
import random

random_choice = random.randint(0, 2)
print('Компьютер выбирает', random_choice)
```

Мы добавили одну строку кода, чтобы что-то вывести

Оболочка Python

```
Компьютер выбирает 2
>>>
```

Вот что мы получили. Запустите программу несколько раз и убедитесь, что она делает случайный выбор

Дальнейшие действия

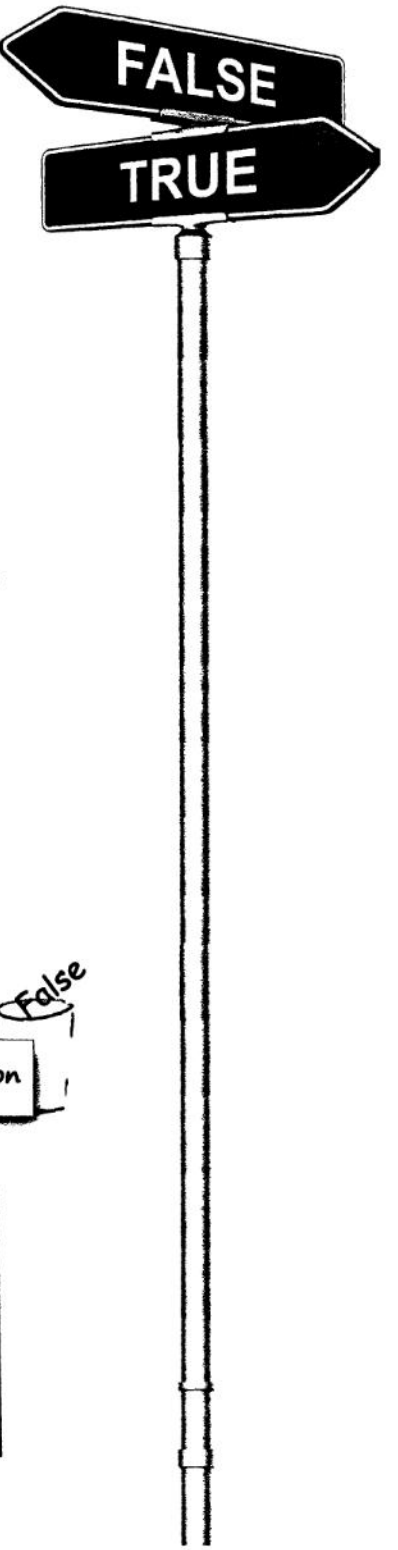
Итак, воспользовавшись модулем `random`, мы реализовали выбор компьютером случайного варианта, но это не совсем то, что нам нужно. Почему? Наша цель — заставить компьютер выбрать камень, ножницы или бумагу; мы сопоставляем эти варианты с числами 0, 1 и 2. Но разве не было бы удобнее иметь переменную, содержащую строку `"камень"`, `"ножницы"` или `"бумага"`? Давайте поступим именно так, только для этого нам придется вернуться немного назад и разобраться, как принимать решения в программах, написанных на Python.

Возьмите карандаш

Предполагая, что переменная `random_choice` равна 0, 1 или 2, напишите псевдокод, в котором переменной `computer_choice` присваивается значение `"камень"`, `"ножницы"` или `"бумага"` в зависимости от значения `random_choice`.

Вы пока не знаете, как реализовать это на Python, но вспомните, что псевдокод близок к обычному разговорному языку. Так что все не настолько сложно.

Истина или ложь?



Python принимает решения на основании вопросов, предполагающих ответ "да" или "нет". В программировании это называется "истина" или "ложь". Сами по себе такие вопросы являются обыкновенными выражениями, только результатом выражения в данном случае будет не строка или число, а значение `True` или `False`. Рассмотрим пример:

В этом выражении сравниваются две переменные, содержащие числа

Это один из операторов сравнения, в данном случае оператор "больше", возвращающий True, если первый операнд больше, чем второй, и False в противном случае

Вскоре мы рассмотрим и другие операторы сравнения, такие как "меньше", "равно" и др.

```
bank_balance > ferrari_cost
```

Выражение можно прочитать как "больше ли bank_balance, чем ferrari_cost"

Результат данного выражения равен либо True, либо False, в зависимости от того, превышает ли счет в банке стоимость Ferrari

Результат выражения можно присвоить переменной и даже вывести на экран.

Это тоже выражение

```
decision = bank_balance > ferrari_cost
print(decision)
```

Переменная decision содержит результат выражения (True или False)

Вот результат

```
Оболочка Python
False
>>>
```

Сегодня без Ferrari

Значения `True` и `False` относятся к *булевому типу данных*. Познакомимся с ним поближе...

Булевы значения

Простите, нехорошо получилось: мы говорим о совершенно новом типе данных, но не представили его формально. **Булев тип данных** очень прост: он поддерживает всего два возможных значения, которыми, как вы уже поняли, являются `True` и `False`.

Булевы значения не отличаются от любых других значений в том смысле, что их можно хранить в переменных, выводить на экран и включать в выражения. Давайте поработаем с ними, чтобы понять, как применять их для принятия решений.

Не бойтесь задавать вопросы

В: Булевы?

О: Звучит необычно, не правда ли? Булевы значения обязаны своим названием Джорджу Булю — английскому математику XIX века, разработавшему принципы математической логики. Несмотря на режущее слух название, булевы значения находят широкое применение в программировании. Уверены, вы быстро привыкнете к этому термину и будете постоянно им ользоваться.

Возьмите карандаш

Пришло время взять карандаш и немного поупражняться. Вычислите значение каждого из приведенных ниже выражений и впишите свой ответ, после чего сверьтесь с ответами, приведенными в конце главы. Помните: булевы выражения всегда равны `True` либо `False`.

Проверяем, будет ли первое значение больше, чем второе. Можно также использовать оператор >=, чтобы проверить, будет ли первое значение больше или равно второму

```
your_level > 5
```

Если `your_level=2`, то чему равно выражение? _____

Если `your_level=5`, то чему равно выражение? _____

Если `your_level=7`, то чему равно выражение? _____

Оператор == проверяет равенство двух значений. Если они равны, то результат True, иначе — False

```
color == "orange"
```

Если переменная `color` содержит `"pink"`, то выражение равно `True` или `False`? _____

А если переменная содержит `"orange"`? _____

Оператор != проверяет, будут ли значения НЕ равны друг другу

```
color != "orange"
```

Если переменная `color` содержит `"pink"`, то выражение равно `True` или `False`? _____

По-серьезному

Это два знака равенства подряд

Обратите внимание на то, что проверка равенства в приведенных выше примерах выполняется с помощью оператора ==, а не =, который обозначает присваивание. Другими словами, мы используем один знак равенства, чтобы присвоить значение переменной, и два знака равенства, чтобы проверить, равны ли две переменные друг другу. Новички (а порой и более опытные программисты) часто путают эти операторы.

Принимаем решения

Теперь, когда мы познакомились с булевыми значениями и операторами сравнения, такими как >, < и ==, можно применить их для принятия решений в программе. Для этого используется ключевое слово `if` (если) совместно с булевым выражением. Рассмотрим пример.

Начинаем с ключевого слова `if`

Затем идет булево выражение, называемое условным выражением. Оно равно либо True, либо False

Далее ставим двоеточие

```
if bank_balance >= ferrari_cost:
    print('Почему бы и нет?')
    print('Эх, покупаю!')
```

Далее идут инструкции, выполняемые, когда условие равно True

Заметьте: все инструкции, которые мы хотим выполнить, когда условие равно True, имеют отступ

В Python принято делать отступ шириной 4 пробела

Инструкцию можно расширить, указав с помощью ключевого слова `else` альтернативный набор инструкций, выполняемых, когда условное выражение равно `False`.

В Python после двоеточия всегда идет блок инструкций с отступом

Добавляем ключевое слово `else`

```
if bank_balance >= ferrari_cost:
    print('Почему бы и нет?')
    print('Эх, покупаю!')
else:
    print('Жаль.')
    print('Может, через неделю.')
```

Далее ставим двоеточие

Далее идут инструкции, выполняемые, когда условие равно False

Заметьте, что все инструкции, которые мы хотим выполнить, когда условие равно False, тоже имеют отступ в 4 пробела

Дополнительные условия

В случае необходимости решение может приниматься на основе целого набора условий – каждое следующее из них указывается после ключевого слова `elif`. Оно выглядит странным, но в действительности это просто сокращение фразы "else if" (иначе если). Рассмотрим пример.

Начинаем с первого условия, используя ключевое слово `if`

```
if number_of_scoops == 0:
   print("Не желаете мороженое?")
   print('У нас много разных сортов.')
elif number_of_scoops == 1:
    print('Один шарик, сейчас сделаем.')
elif number_of_scoops == 2:
    print('Прекрасно, два шарика!')
elif number_of_scoops >= 3:
    print("Ого, это много шариков!")
else:
    print("Простите, но у нас нет отрицательных шариков.")
```

Затем идет ключевое слово `elif` со вторым условием

Далее можно добавить произвольное число блоков `elif` со своими условиями

Помните, что в каждом блоке `if`, `elif` и `else` может быть любое число инструкций

В конце идет завершающий блок `else`, выполняемый, если ни одно из предыдущих условий не сработало

Учтите, что будут выполнены инструкции только для первого условия, равного True. Если таких условий нет, выполняется блок `else`

Возьмите карандаш

В приведенной ниже таблице указано несколько возможных значений переменной `number_of_scoops`. Выпишите для каждого из них соответствующее сообщение, выдаваемое программой. Первый результат мы вписали за вас.

При таком значении `number_of_scoops` выводится...

`number_of_scoops = 0`	Не желаете мороженое? У нас много разных сортов.
`number_of_scoops = 4`	
`number_of_scoops = 1`	
`number_of_scoops = 3`	
`number_of_scoops = 2`	
`number_of_scoops = -1`	

Возьмите карандаш

Прочитайте вслух приведенный выше код, переведя его на родной язык, а затем запишите, что у вас получилось.

Вернемся к игре

Мы все еще не завершили первый этап игры "Камень, ножницы, бумага". Вспомните: перед тем как отвлечься на тему булевых значений, мы собирались улучшить программу, чтобы компьютер мог выбрать строку "камень", "ножницы" или "бумага", а не число 0, 1 или 2. Теперь, когда вы познакомились с инструкцией `if`, мы сможем легко это сделать. Нам нужно написать код, который в зависимости от значения переменной `random_choice` будет помещать в новую переменную `computer_choice` соответствующую строку.

Это наш код на данный момент

```
import random

random_choice = random.randint(0, 2)
print('Компьютер выбирает', random_choice)

if random_choice == 0:
    computer_choice = 'камень'
elif random_choice == 1:
    computer_choice = 'бумага'
else:
    computer_choice = 'ножницы'

print('Компьютер выбирает', computer_choice)
```

Эта инструкция нам больше не нужна, удаляем ее

Добавьте новый код в свой файл rock.py

Если переменная random_choise равна 0, значит, компьютер выбирает 'камень'

В противном случае проверяем, не равна ли переменная 1. Если это так, значит, компьютер выбирает 'бумага'

В противном случае единственный оставшийся выбор — 'ножницы'

Чтобы протестировать код, выведем значение переменной computer_choice

Тест-драйв

Не забудьте добавить приведенный выше код в свой программный файл и протестировать его.

Это наш первый запуск. Сделайте несколько запусков

Оболочка Python

```
Компьютер выбирает ножницы
>>>
```

Пользователь делает свой выбор

После того как компьютер сделал свой выбор, пора определиться с выбором пользователя. В главе 2 вы узнали, как получать данные от пользователя с помощью функции `input()`. Нам нужно выдать пользователю запрос и сохранить полученный ответ в переменной `user_choice`.

```
import random

random_choice = random.randint(0, 2)

if random_choice == 0:
    computer_choice = 'камень'
elif random_choice == 1:
    computer_choice = 'бумага'
else:
    computer_choice = 'ножницы'
```

Эта отладочная инструкция нам больше не нужна

Мы сохраняем строку, возвращаемую функцией `input()`, в переменной `user_choice`

```
~~print('Компьютер выбирает', computer_choice)~~

user_choice = input('камень, ножницы или бумага? ')
print('Вы выбрали', user_choice + ', компьютер выбрал', computer_choice)
```

Мы снова используем функцию `input()` и просим пользователя сделать выбор

Добавим инструкцию вывода, чтобы следить за состоянием основных переменных

Тест-драйв

Как быстро развиваются события! Не забудьте добавить этот код в файл *rock.py* и протестировать программу.

Это наш первый запуск. Сделайте несколько запусков

Оболочка Python

```
камень, ножницы или бумага? камень
Вы выбрали камень, компьютер выбрал ножницы
>>>
```

В одном из примеров главы 1 мы применяли функцию random.choice(). Может, она и здесь пригодится?

Хорошая идея!

Как уже говорилось, модуль random содержит большое количество полезных функций. Одна из них — choice(), которую мы и применим

Обратите внимание на квадратные скобки; о списках мы поговорим в следующей главе

Сначала создаем список вариантов, т.е. просто список строк

```
choices = ['камень', 'бумага', 'ножницы']
computer_choice = random.choice(choices)
```

Затем мы передаем список функции choice(), которая случайным образом выбирает один элемент

Это именно то, что нам нужно, поскольку таким образом мы можем сократить код, сделав его более понятным. По большому счету, если бы мы с самого начала применили функцию choice(), нам не пришлось бы заботиться о принятии решений, булевых значениях, операторах сравнения, условных выражениях, типах данных... вы понимаете, к чему мы клоним.

Как бы там ни было, с функцией choice() вам еще только предстоит познакомиться. Она особенно полезна при обработке списков, о которых вы узнаете в следующей главе.

ПРИМЕЧАНИЕ: те, кому не терпится переписать имеющийся код с учетом функции random.choice(), могут сделать это прямо сейчас. Вам придется заменить все строки между инструкцией import и вызовом функции input(). Пока что в этом нет особого смысла, но пытливого исследователя ничего не остановит.

Булевы значения в эфире

Интервью недели: "Знакомство с булевыми значениями"

[Редакция Head First] Приветствуем вас в студии! Знаем, что вы чрезвычайно заняты, участвуя во множестве проектов Python, поэтому спасибо, что нашли время на интервью.

[Булевы значения] Всегда рады! Да, мы действительно очень загружены последнее время, люди постоянно к нам обращаются, и это замечательно!

Что особенно удивительно, ведь вас всего двое: Да и Нет.

Вообще-то, нас зовут `True` (Истина) и `False` (Ложь), не путайте.

Да, простите, но, так или иначе, вас лишь двое. Достаточно ли этого, чтобы считаться типом данных?

Более чем. Мы используемся в любом алгоритме и любой программе. Мы есть во всех языках программирования, не только в Python. Нас мало, но мы везде.

Хорошо, в чем тогда секрет вашей популярности?

Мы можем представить любое условие в программе. Данные загружены? Список отсортирован? Платеж прошел? В каждом из этих случаев ответом будет булево значение `True` или `False`, которое определяет, как программа будет дальше выполняться.

Вы сейчас говорите об условных выражениях. Мы проверяем условия, чтобы определить, какой код будет выполняться далее.

Официальный термин — *булево выражение*, а условия — их неотъемлемая часть. Сначала проверяется условие, а затем с помощью инструкции типа `if` указывается, что должно произойти дальше. Вот так мы управляем *порядком выполнения* программы.

Вы применяетесь еще где-то?

Как вскоре узнают ваши читатели, можно многократно повторять код, пока булево выражение равно `True`. Например, можно повторно запрашивать пароль, пока пользователь не введет его правильно; можно продолжать считывать данные, пока не будет достигнут конец файла, и т.п.

И все же удивительно, что вас только двое. Думаете, вас воспринимают серьезно?

Мы не просто два значения. В честь нас назван целый раздел математики — булева алгебра. Слышали о таком?

Нет.

Да ладно! Булева алгебра описывает операции с булевыми значениями.

Которых всего два: `True` и `False`!

Ну хорошо, вот вы, к примеру, сейчас программируете игру "Камень, ножницы, бумага". В ней множество логических инструкций. Как вы преобразуете блок-схему в код определения победителя?

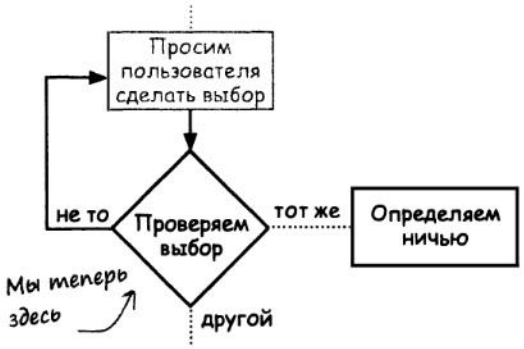
Ладно, вы нас убедили.

Это ведь несложно. Сейчас я покажу...

...к сожалению, наше время в эфире подошло к концу. Было приятно побеседовать с вами. До новых встреч!

Спасибо за приглашение!

Проверяем, что выбрал пользователь



После того как пользователь сделал свой выбор, необходимо проанализировать его. Согласно блок-схеме у нас три варианта.

- ☐ Если пользователь и компьютер сделали одинаковый выбор, получаем ничью.
- ☐ Если пользователь и компьютер сделали разный выбор, нужно определить, кто выиграл.
- ☐ Если пользователь сделал некорректный выбор, нужно попросить его повторить.

Имеется в виду, что пользователь ввел слово, не равное "камень", "ножницы" или "бумага"; этот вариант мы обработаем позже

Мы пройдем все три варианта по очереди (последний из них будет рассмотрен позже), но сначала необходимо создать переменную, которая будет хранить введенное значение. Мы назовем ее `winner`. Она может содержать строку `'Ничья'`, `'Пользователь'` или `'Компьютер'`. Переменная инициализируется следующим образом:

```
winner = ''
```

Это пустая строка: между кавычками нет пробела

Вскоре мы добавим эту переменную в нашу программу

Изначально переменной `winner` назначается пустая строка, т.е. строка, не содержащая ни одного символа (такое допускается). Это можно сравнить с пустым кошельком: даже если в нем нет денег, он все равно остается кошельком. Подобные возможности есть во всех языках программирования: пустые строки, пустые списки, пустые файлы и т.п. Для нас инициализация переменной `winner` пустой строкой означает, что переменная будет содержать строку, которая пока еще не определена.

Позже будет рассмотрен альтернативный подход к определению начального значения

Теперь, когда у нас есть переменная `winner`, хранящая результат игры, можно приступить к реализации остального кода. Согласно первому варианту, если компьютер и пользователь делают одинаковый выбор, необходимо записать в переменную `winner` значение `'Ничья'`. Для этого нужно написать код для сравнения строк, выбранных пользователем и компьютером.

Возьмите карандаш

Снова ваша очередь проявить себя. Завершите приведенный ниже код согласно нашему плану. Программа должна определять ничью и присваивать переменной `winner` значение `'Ничья'`.

```
if _________ == _________:
    winner = ______
```

Код определения ничьей

Итак, у нас есть новая переменная `winner` и новый код, проверяющий, не равны ли друг другу выбранные пользователем и компьютером варианты. Если это так, объявляется ничья. Сведем все вместе.

Это наша новая переменная, `winner`. *Пока что она содержит пустую строку, но позже в нее будет записываться результат игры:* `'Пользователь'`, `'Компьютер'` *или* `'Ничья'`

```
import random

winner = ''

random_choice = random.randint(0, 2)

if random_choice == 0:
    computer_choice = 'камень'
elif random_choice == 1:
    computer_choice = 'бумага'
else:
    computer_choice = 'ножницы'

user_choice = input('камень, ножницы или бумага? ')
print('Вы выбрали', user_choice + ', компьютер выбрал', computer_choice)

if computer_choice == user_choice:
    winner = 'Ничья'
```

Эту инструкцию можно удалить

Если компьютер и пользователь делают одинаковый выбор, то объявляется ничья

УПРАЖНЕНИЕ

Внесите указанные изменения в файл *rock.py*. Мы все протестируем позже, после внесения дополнительных изменений. А пока что просто выполните программу и убедитесь в отсутствии синтаксических ошибок. Учтите, что после удаления инструкций `print` программа ничего не выводит на экран.

Кто выиграл?

На данный момент мы написали код, определяющий ничью, и нас ждет самое интересное: *определение победителя*. Мы располагаем всем необходимым для этого: выбор компьютера хранится в переменной `computer_choice`, а выбор пользователя — в переменной `user_choice`. Осталось определиться с *логикой* принятия решения о победителе. Для этого нам нужно изучить приведенную ранее блок-схему игры и попытаться разбить процесс определения победителя на ряд простых правил. Кое-что можно сразу упростить. Дело в том, что нам не нужно применять правила к обоим игрокам поочередно, ведь мы знаем: если компьютер проиграл, значит, пользователь победил, и наоборот. Это уменьшает объем кода, так как мы проверяем только одного игрока.

С учетом этого рассмотрим все варианты, при которых компьютер побеждает.

Теперь нужно взяться за это

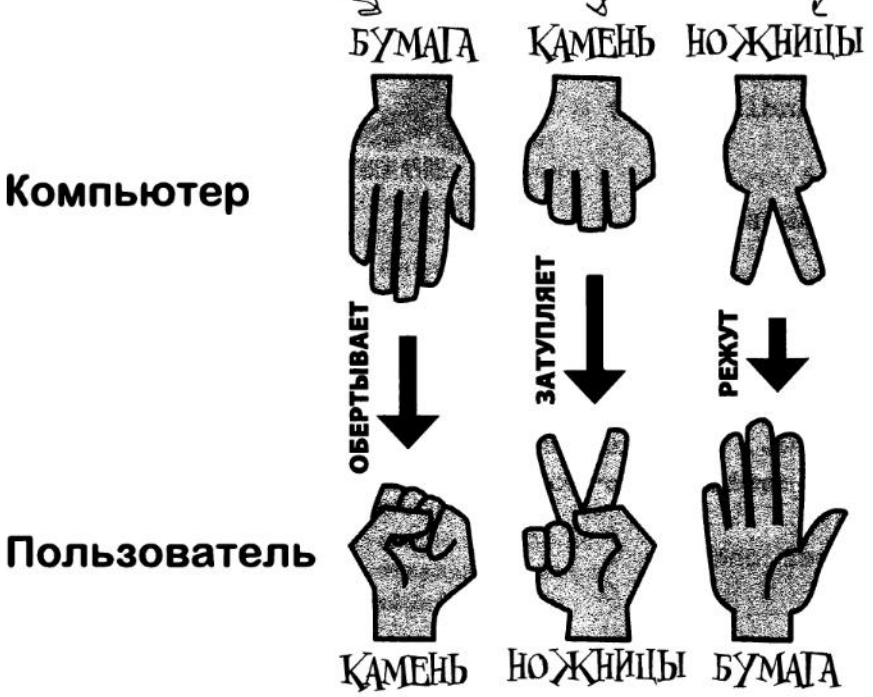
Список задач:

- [x] Если пользователь и компьютер сделали одинаковый выбор, получаем ничью.
- [ ] Если пользователь и компьютер сделали разный выбор, нужно определить, кто выиграл.
- [ ] Если пользователь сделал некорректный выбор, нужно попросить его повторить.

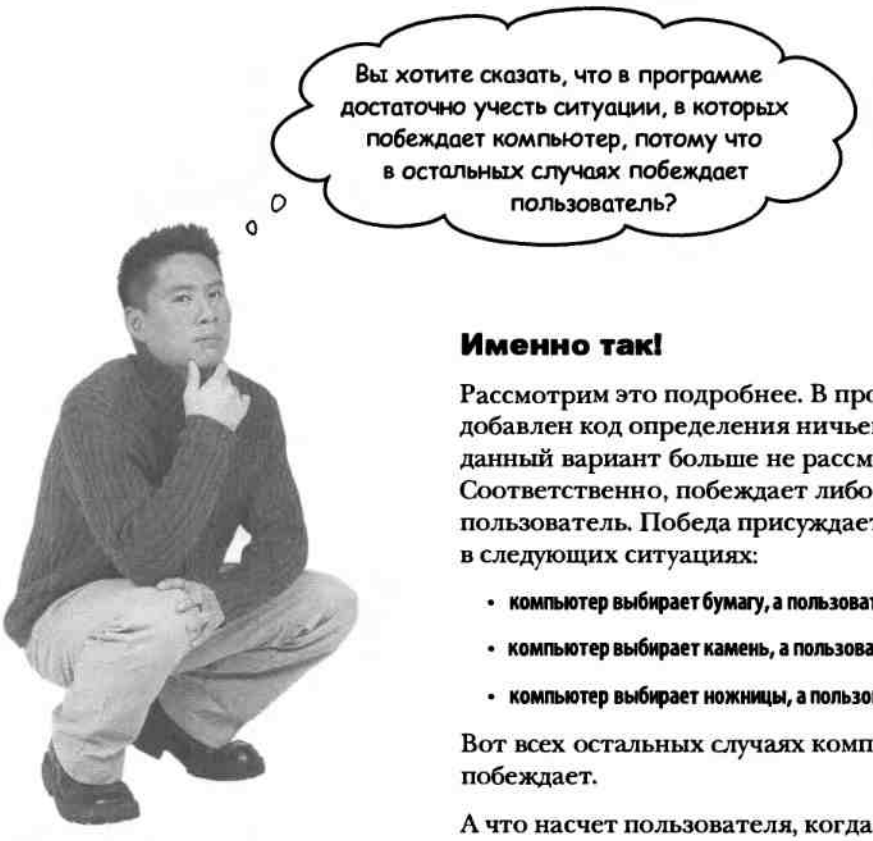
Если компьютер выбирает бумагу и пользователь выбирает камень, то побеждает компьютер

Если компьютер выбирает камень и пользователь выбирает ножницы, то побеждает компьютер

Если компьютер выбирает ножницы и пользователь выбирает бумагу, то побеждает компьютер



Во всех остальных случаях побеждает пользователь или объявляется ничья



Именно так!

Рассмотрим это подробнее. В программу уже добавлен код определения ничьей, так что данный вариант больше не рассматриваем. Соответственно, побеждает либо компьютер, либо пользователь. Победа присуждается компьютеру в следующих ситуациях:

- **компьютер выбирает бумагу, а пользователь — камень;**
- **компьютер выбирает камень, а пользователь — ножницы;**
- **компьютер выбирает ножницы, а пользователь — бумагу.**

Вот всех остальных случаях компьютер не побеждает.

А что насчет пользователя, когда побеждает он? Можно было бы привести три аналогичных правила, только зачем? Раз у нас точно не ничья (данное условие уже проверено) и вариант с победой компьютера не подходит, значит, побеждает пользователь! Таким образом, нам не нужно писать код, проверяющий выбор пользователя. Если выясняется, что компьютер не выиграл, то победителем объявляется пользователь.

Осталось реализовать это в программе.

Реализация логики игры

Как видите, у нас есть три варианта, при которых побеждает компьютер, и в каждом из них нужно проверить два условия, например "выбрал ли компьютер бумагу" И "выбрал ли пользователь камень". До сих пор мы еще не проверяли два условия одновременно. Мы умеем делать только одиночные проверки. Вот как проверить, выбрал ли компьютер бумагу:

```
computer_choice == 'бумага'
```

Это уже привычное для нас булево выражение, позволяющее проверить, выбрал ли компьютер бумагу

А вот как проверить, выбрал ли пользователь камень:

```
user_choice == 'камень'
```

Еще одно булево выражение, проверяющее, выбрал ли пользователь камень

Но как проверить сразу оба условия?

Для этого применяются **булевы операторы** (их также называют **логическими операторами**), которые позволяют объединять булевы выражения. Пока что вам достаточно знать три таких оператора: **and** (логическое И), **or** (логическое ИЛИ) и **not** (логическое НЕ).

С еще одним булевым оператором мы познакомимся позже

Для проверки того, что компьютер выбрал бумагу И пользователь выбрал камень, необходимо воспользоваться булевым оператором and, объединив две проверки.

Это наше первое условие

Это наше второе условие

```
computer_choice == 'бумага' and user_choice == 'камень'
```

Вся фраза — это булево выражение, равное либо True, либо False

Добавление оператора and между условиями означает, что все выражение равно True тогда и только тогда, когда оба условия равны True

Данное выражение можно добавить в инструкцию `if`.

Теперь условие инструкции `if` является составным булевым выражением

```
if computer_choice == 'бумага' and user_choice == 'камень':
    winner = 'Компьютер'
```

Этот фрагмент кода определяет одно из условий, когда побеждает компьютер

Если выражение равно True, то выполняется эта инструкция

Подробнее о булевых операторах

Как вы смогли только что убедиться, оператор `and` возвращает `True`, только когда *оба* его условия (их называют *операндами*) истинны. А как работают операторы `or` и `not`? Оператор `or` тоже объединяет два булевых выражения, возвращая `True`, когда *любой* из операндов истинен.

По-серьезному

Операторы сравнения (>, <, == и т.п.) имеют более высокий приоритет, чем булевы операторы `and`, `or` и `not`. Используйте скобки, если нужно изменить стандартный порядок операций или сделать выражение более понятным.

Вы сможете купить Ferrari, если на банковском счете достаточно денег...

ИЛИ вы берете кредит, равный стоимости машины

```
if bank_balance > ferrari_cost or loan == ferrari_cost:
    print('Покупаю!')
```

Для покупки Ferrari достаточно, чтобы одно из этих условий было равно True, но если они оба истинны, то этот вариант тоже подходит. Если оба условия равны False, то с Ferrari придется подождать

Оператор `not` ставится перед булевым значением или выражением, меняя результат на противоположный. Другими словами, если операнд равен `True`, оператор возвращает `False`, и наоборот. Такая операция называется *логическим отрицанием*.

Оператор not ставится перед булевым выражением

Сначала вычисляется булево выражение (True или False), а затем благодаря оператору not его значение меняется на противоположное

```
if not bank_balance < ferrari_cost:
    print('Покупаю!')
```

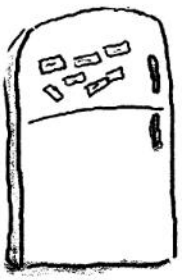
Выражение можно прочитать как "если счет в банке НЕ меньше, чем стоимость Ferrari, тогда покупаем"

Возьмите карандаш

Возьмите карандаш и запишите, чему равно каждое из приведенных ниже булевых выражений.

`age > 5 and age < 10`	Если `age=6`, то чему равно выражение? _____
	Если `age=11`, то чему равно выражение? _____
	Если `age=5`, то чему равно выражение? _____
`age > 5 or age == 3`	Если `age=6`, то чему равно выражение? _____
	Если `age=2`, то чему равно выражение? _____
	Если `age=3`, то чему равно выражение? _____
`not (age > 5)`	Если `age=6`, то чему равно выражение? _____
	Если `age=2`, то чему равно выражение? _____

Обратите внимание на добавление скобок для наглядности



Код на магнитиках

Мы составили код игры "Камень, ножницы, бумага" с помощью набора магнитиков, прикрепленных к дверце холодильника, но кто-то пришел и сбросил их на пол. Сможете расположить магнитики в правильном порядке, чтобы нам удалось определить победителя? Учтите, что некоторые магнитики могут оказаться лишними. Не забудьте свериться с ответом в конце главы.

```
:   elif   if   or   and   and   or   and   and   if   elif   =   elif   if   :   :
```

```
if computer_choice == user_choice :
    winner = 'Ничья'
elif computer_choice == 'бумага' and user_choice == 'камень' :
    winner = 'Компьютер'
else :
    winner == 'Пользователь'
```

Это первый случай, когда компьютер и пользователь сделали одинаковый выбор, поэтому у нас ничья

К счастью, на дверце остались магнитики с первой проверкой, где побеждает компьютер, и самой последней, где побеждает пользователь

Поместите сюда свои магнитики

```
winner = 'Компьютер'   winner = 'Компьютер'   computer_choice == 'камень'
user_choice == 'бумага'   user_choice == 'ножницы'   computer_choice == 'ножницы'   winner = 'Пользователь'
```

Итак, нам удалось воссоздать логику определения победителя в игре "Камень, ножницы, бумага". Внесите код в файл *rock.py* и запустите программу пару раз. Мы пока еще не добавили поясняющих сообщений, поэтому мы знаем, кто победил, но не знаем, каким был выбор компьютера. Этим мы займемся дальше.

Мы сейчас здесь

```
import random

winner = ''

random_choice = random.randint(0, 2)

if random_choice == 0:
    computer_choice = 'камень'
elif random_choice == 1:
    computer_choice = 'бумага'
else:
    computer_choice = 'ножницы'

user_choice = input('камень, ножницы или бумага? ')

if computer_choice == user_choice:
    winner = 'Ничья'
elif computer_choice == 'бумага' and user_choice == 'камень':
    winner = 'Компьютер'
elif computer_choice == 'камень' and user_choice == 'ножницы':
    winner = 'Компьютер'
elif computer_choice == 'ножницы' and user_choice == 'бумага':
    winner = 'Компьютер'
else:
    winner = 'Пользователь'

print(winner, ' выиграл!')
```

Вот код, описывающий логику нашей игры. Введите его

Сыграйте несколько раз и убедитесь, что все работает. Далее мы добавим более понятные сообщения

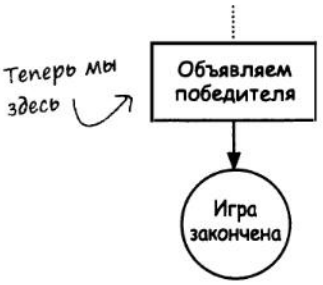
Возможно появление сообщения "Ничья выиграл", но вскоре мы это исправим

Оболочка Python

```
камень, ножницы или бумага? камень
Компьютер выиграл!
>>>
```

Вывод имени победителя

Пришло время вывести на экран имя победителя. Игра завершается победой пользователя, ничьей или победой компьютера.



Оболочка Python

```
камень, ножницы или бумага? камень

Пользователь выиграл, я выбрал ножницы.

камень, ножницы или бумага? камень

Мы оба выбрали камень, играем снова.

камень, ножницы или бумага? бумага

Компьютер выиграл, я выбрал ножницы.
```

В первую очередь добавим код, в котором проверяется ничья. В этом случае переменная `winner` будет иметь значение `'Ничья'`. Запишем соответствующее условие.

Помните о том, что функция `print()` вставляет пробел между аргументами, разделенными запятой. Поэтому иногда мы используем конкатенацию, если хотим отказаться от пробелов (например, после слова должен стоять знак препинания)

Если победителем стала `'Ничья'`, то...

```
if winner == 'Ничья':
    print('Мы оба выбрали', computer_choice + ', играем снова.')
```

Вывести сообщение с указанием того, какой выбор сделали оба игрока

Здесь можно использовать и переменную `user_choice`, но поскольку у нас ничья, обе переменные содержат одно и то же

Если же у нас другой исход игры, то необходимо объявить победителя, имя которого хранится в переменной `winner`.

Этот код будет выполнен, только если у нас не ничья

```
else:
    print(winner, 'выиграл, я выбрал', computer_choice + '.')
```

Объявляем победителя

А затем сообщаем, какой выбор сделал компьютер

Теперь у нас полнофункциональная игра! Добавьте новый код в файл *rock.py* и запустите программу.

```
import random

winner = ''

random_choice = random.randint(0, 2)

if random_choice == 0:
    computer_choice = 'камень'
elif random_choice == 1:
    computer_choice = 'бумага'
else:
    computer_choice = 'ножницы'

user_choice = input('камень, ножницы или бумага? ')

if computer_choice == user_choice:
    winner = 'Ничья'
elif computer_choice == 'бумага' and user_choice == 'камень':
    winner = 'Компьютер'
elif computer_choice == 'камень' and user_choice == 'ножницы':
    winner = 'Компьютер'
elif computer_choice == 'бумага' and user_choice == 'бумага':
    winner = 'Компьютер'
else:
    winner = 'Пользователь'

print(winner, 'выиграл!')

if winner == 'Ничья':
    print('Мы оба выбрали', computer_choice + ', играем снова.')
else:
    print(winner, 'выиграл, я выбрал', computer_choice + '.')
```

Удалите это

В этом фрагменте объявляется результат игры

Теперь у нас полнофункциональная игра!

Оболочка Python

```
камень, ножницы или бумага? ножницы
Компьютер выиграл, я выбрал камень.
>>>
```

А где документация?

Пора взять паузу и просмотреть все, что мы написали на данный момент. Программа получилась достаточна длинной, и если в будущем вы захотите к ней вернуться, то наверняка вам придется вспоминать, каково назначение тех или иных фрагментов, как они взаимосвязаны, какие решения принимались на этапе разработки алгоритма и почему.

Обратите внимание на то, что программа имеет определенную структуру, в которой четко выделяется несколько функциональных блоков, соответствующих элементам блок-схемы. Давайте обозначим эти блоки, добавив к ним пояснения, что позволит быстро вспомнить их назначение в будущем.

Программу также полезно документировать для других программистов, которые могут захотеть поработать с ней в будущем

Начинаем с того, что импортируем модуль `random` *и объявляем переменную* `winner`

```
import random

winner = ''
```

Компьютер делает произвольный выбор, генерируя случайное целое число в диапазоне от 0 до 2, а затем сопоставляя его с соответствующей строкой

```
random_choice = random.randint(0, 2)

if random_choice == 0:
    computer_choice = 'камень'
elif random_choice == 1:
    computer_choice = 'бумага'
else:
    computer_choice = 'ножницы'
```

С помощью функции `input()` *запрашиваем выбор пользователя*

```
user_choice = input('камень, ножницы или бумага? ')
```

Здесь реализуется логика игры: мы определяем победителя и соответствующим образом меняем переменную `winner`

```
if computer_choice == user_choice:
    winner = 'Ничья'
elif computer_choice == 'бумага' and user_choice == 'камень':
    winner = 'Компьютер'
elif computer_choice == 'камень' and user_choice == 'ножницы':
    winner = 'Компьютер'
elif computer_choice == 'ножницы' and user_choice == 'бумага':
    winner = 'Компьютер'
else:
    winner = 'Пользователь'
```

Здесь мы объявляем либо ничью, либо победителя и сообщаем о выборе компьютера

```
if winner == 'Ничья':
    print('Мы оба выбрали', computer_choice + ', играем снова.')
else:
    print(winner, 'выиграл, я выбрал', computer_choice + '.')
```

Но вам не кажется странным, что программа документируется в *книге*? В конце концов, программа ведь содержится в компьютере. Почему бы не задокументировать *сам код*, чтобы документация всегда была под рукой? Рассмотрим, как это делается.

Добавление комментариев в код

В Python, как и в любом другом языке программирования, в программный код можно включать текстовые комментарии. Чтобы добавить комментарий в программу, введите символ решетки (#), а после него — текст комментария. Все, что стоит после символа решетки, будет проигнорировано интерпретатором Python. В комментариях вы оставляете заметки себе и другим программистам о написанной программе — ее назначении, общей структуре и примененных решениях. Рассмотрим пример.

Комментарии — один из способов документирования программы. Позже мы рассмотрим справочную документацию, предназначенную для программистов, которые будут применять программу, но не обязательно должны понимать ее работу

Начните комментарий со знака решетки, после чего введите описательный текст. Каждая строка комментария должна начинаться со знака решетки

```
# Этот код реализует мою привычку еженедельно проверять,
# могу ли я купить Ferrari. Код сравнивает величину
# моего банковского счета с текущей стоимостью Ferrari.

if bank_balance >= ferrari_cost:
    # Если счет больше, то можно купить
    print('Почему бы и нет?')
    print('Эх, покупаю!')
else:
    print('Жаль.')
    print('Может, через неделю.') # не судьба
```

Комментарий может начинаться в любом месте строки

Он даже может стоять после строки кода

УПРАЖНЕНИЕ

Добавьте в редакторе IDLE комментарии к файлу *rock.py*. Используйте пояснения на предыдущей странице в качестве образца, но можете вводить и свои комментарии. Не забывайте, что комментарии должны быть понятны любому (включая вас самого!), кто в будущем захочет просмотреть исходный код программы. С нашим решением вы сможете ознакомиться в конце главы.

Внимание!

Следуйте нашим рекомендациям, но не занимайтесь слепым копированием

Документирование кода — важный этап в программировании, однако в данной книге мы не слишком активно к этому прибегаем. Причина заключается в использовании большого числа поясняющих выносок к коду, которые служат характерным элементом серии *Head First*. К тому же, если комментировать каждую строку кода в книге, то погибнет намного больше деревьев.

Имеются в виду такие выноски

СИЛА МЫСЛИ

Анализируя игру "Камень, ножницы, бумага", давайте представим, что у пользователя "хромает" правописание. Что произойдет, если вместо "бумага" ввести "бамага"? Как поведет себя программа в подобных случаях? Считаете ли вы такое поведение допустимым?

Завершение игры

Не кажется ли вам, что программа все еще не завершена? Проверим список задач. Так и есть: мы не учли возможность ввода пользователем некорректных данных! В программе *предполагается*, что пользователь вводит "камень", "ножницы" или "бумага", но ведь он может ввести что-то другое, например допустить опечатку ("ножнецы" вместо "ножницы") или попытаться соригинальничать, введя "собака", "нож" или "нет". В реальных приложениях необходимо учитывать, что пользователям свойственно ошибаться.

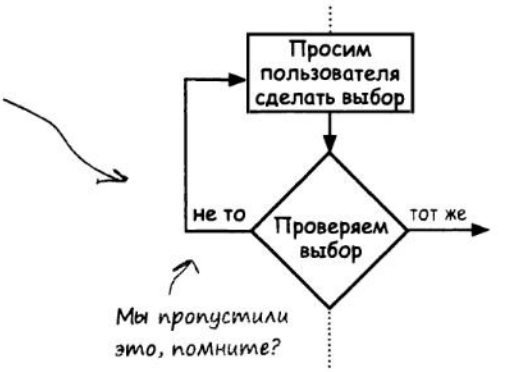
Список задач:

- [x] Если пользователь и компьютер сделали одинаковый выбор, получаем ничью.
- [x] Если пользователь и компьютер сделали разный выбор, нужно определить, кто выиграл.
- [ ] Если пользователь сделал некорректный выбор, нужно попросить его повторить.

Мы кое-что забыли

Давайте добавим в программу соответствующую проверку.

Но сначала нужно понять, как программа должна реагировать на ввод пользователем неправильного ответа. В соответствии с нашей блок-схемой ожидается, что программа будет выдавать повторный запрос в случае ввода некорректных данных.



Мы пропустили это, помните?

Это будет выглядеть примерно следующим образом.

Оболочка Python

```
камень, ножницы или бумага? камин
камень, ножницы или бумага? бамага
камень, ножницы или бумага? камень
Пользователь выиграл, я выбрал ножницы.
>>>
```

Если получена неправильная строка, можно продолжать выдавать запрос

В дальнейшем реакцию программы можно будет усложнить, а пока что мы просто повторяем запрос до тех пор, пока пользователь не введет корректный ответ.

Итак, нам нужно написать еще один фрагмент кода. Но сначала необходимо понять, как решить две задачи.

1. Как определить, что введены неправильные данные?
2. Как повторять вывод запроса до тех пор, пока не будет получен корректный ответ?

Пользователи часто совершают ошибки. Следите за тем, чтобы все они должным образом обрабатывались в программе, даже если у нее один пользователь — вы.

Проверка правильности введенных данных

Как узнать, что введенные пользователем данные ошибочны? Конечно же, воспользовавшись булевой логикой! Но как должно выглядеть выражение, позволяющее определить неправильный ответ? Иногда лучше все обсудить в команде. Мы знаем, что выбор пользователя неверен, если...

Возьмите карандаш

Запишите для этих фраз соответствующие булевы выражения. Первую часть мы сделали за вас.

```
user_choice != 'камень' and _____
```

Завершите булево выражение

Проверка и уточнение выражения

Надеемся, составленное вами в предыдущем упражнении выражение оказалось близким к нашему решению. Повторим его, на этот раз в составе инструкции `if`.

Слишком длинно и нечитабельно!

```
if user_choice != 'камень' and user_choice != 'бумага' and user_choice != 'ножницы':
```

Общий смысл данной инструкции: если пользователь ввел неверную строку, то...

Конечно, такое выражение вполне допустимо. Просто оно становится слишком громоздким, если вводить его в редакторе кода, да и впоследствии его будет трудновато понять. Для наглядности его лучше переписать в более компактном виде.

Так намного лучше и удобнее читать!

```
if user_choice != 'камень' and
        user_choice != 'бумага' and
        user_choice != 'ножницы':
```

Но что-то пошло не так

Оболочка Python

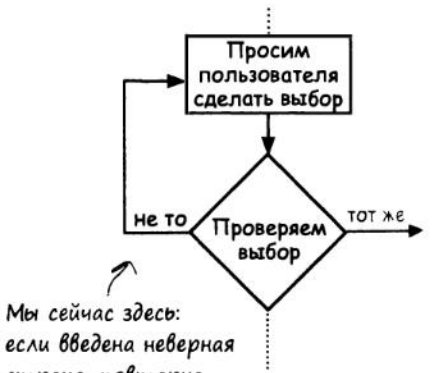
```
    if user_choice != 'камень' and
                                  ^
    SyntaxError: invalid syntax
>>>
```

Проблема в том, что при попытке разбить инструкцию на несколько строк интерпретатор Python выдает сообщение об ошибке.

Тогда поступим по-другому — возьмем выражение в скобки:

```
if (user_choice != 'камень' and
        user_choice != 'бумага' and
        user_choice != 'ножницы'):
```

Заключите выражение в скобки, прежде чем разбивать его на строки



Мы сейчас здесь: если введена неверная строка, повторно выводим приглашение

Теперь, когда программа умеет определять неправильно введенные данные, осталось понять, как выдать повторный запрос пользователю. Давайте немного поразмышляем, как это лучше всего сделать...

Нет, но это можно реализовать, и это разумная идея. В чем тут загвоздка? В том, что чувствительный к регистру символов язык программирования Python считает строки "КАМЕНЬ" и "камень" совершенно разными. Другими словами, в Python, как и в большинстве других языков программирования, следующая проверка вернет `False`.

```
'камень' == 'КАМЕНЬ'
```

← False

Таким образом, если пользователь введет "Камень", а не "камень", программа посчитает это значение недопустимым.

Однако такое поведение программы не кажется пользователям логичным. Для нас удобнее, чтобы слово "камень" распознавалось независимо от того, в каком регистре оно записано.

Как этого добиться? Можно добавить дополнительные правила для проверки всех возможных вариантов написания букв в словах "камень", "ножницы" и "бумага". Но это чересчур усложнит код, к тому же есть более простые способы решения данной задачи, о которых мы узнаем в последующих главах.

Впрочем, пока что нам лучше требовать, чтобы пользователь вводил ответ в нижнем регистре, а корректное решение проблемы будет предложено позже.

Возьмите карандаш

Пока вы раздумывали над проблемой с верхним и нижним регистром, мы забежали немного наперед и написали код для вывода повторного запроса. Только в наше решение закралась ошибка. Сможете ли вы определить, в чем ее причина? Ниже показан пример работы программы вместе с нашими комментариями.

```
user_choice = input('камень, ножницы или бумага? ')
if (user_choice != 'камень' and
        user_choice != 'бумага' and
        user_choice != 'ножницы'):
    user_choice = input('камень, ножницы или бумага? ')
print('Пользователь выбрал', user_choice)
```

Сначала просим пользователя сделать выбор

Затем проверяем, введена ли допустимая строка

Если нет, просим пользователя повторить

Оболочка Python

```
камень, ножницы или бумага? бамага
камень, ножницы или бумага? камин
Пользователь выбрал камин
>>>
```

Выводим повторный запрос

Но затем снова получаем неверную строку

Не то!

Мы думали, что все сделали правильно, так что же мы упустили?

Повторный вывод запроса

Наш первый блин вышел комом. Приведенный выше код позволяет вывести повторный запрос только единожды. Если пользователь снова введет некорректную строку, она будет принята как допустимая.

Можно, конечно, добавить инструкции `if` для второй, третьей и четвертой попыток, но это вызовет настоящую путаницу с кодом. К тому же нам нужно, чтобы повторный запрос выводился столько раз, сколько раз пользователь пытается ввести некорректную строку.

Проблема в том, что на данном этапе наши знания о Python позволяют нам повторить действие всего один раз. А нам нужен код, который смог бы повторяться *многократно*, столько раз, сколько потребуется.

Мы здесь: необходимо повторять запрос, пока не будет получена корректная строка

```
while juggling:
    keep_balls_in_air()
```

Многократное повторение действий

В повседневной жизни мы часто повторяем одни и те же действия:

Завтрак, обед, ужин...

Каждый день ходить на работу...

Читать книгу страница за страницей, пока она не закончится.

В программах часто возникает необходимость в повторном выполнении одних и тех же действий. В Python такая возможность реализована в циклах `while` и `for`. В этой главе мы познакомимся с циклом `while`.

Вам уже знакомы условные выражения, возвращающие булевы значения. Именно на таких выражениях и основывается работа цикла `while`. Вот как он выглядит.

Инструкция начинается с ключевого слова `while`

Условием цикла является булево выражение (как в инструкции `if`)

Если условие истинно, то выполняется весь блок кода

```
while scoops > 0:
    print('Еще шарик!')
    scoops = scoops - 1
```

А что такое блок кода? Формально мы еще не вводили данный термин, но по сути это все, что набрано с отступом после двоеточия

После выполнения блока кода возвращаемся в начало цикла и снова проверяем условие: если оно по-прежнему истинно, то выполняем блок кода повторно, если же оно ложно, то цикл завершен

В Python блок кода определяется по отступам, что довольно нетипично, поскольку в других языках для этого используются фигурные или круглые скобки

Как работает цикл while

Поскольку это наше первое знакомство с циклом `while`, давайте пройдем его шаг за шагом, чтобы понять, как он работает. Обратите внимание на то, что перед началом цикла `while` объявляется переменная `scoops`, которая инициализируется значением 5.

Начнем выполнение кода. Сначала задаем переменную `scoops` равной 5.

```
scoops = 5
while scoops > 0:
    print('Еще шарик!')
    scoops = scoops - 1
print("Жизнь без мороженого скучна.")
```

Примечание от опытных читателей: читайте следующие несколько страниц медленно и внимательно. Здесь много чего происходит, поэтому вся информация должна уложиться в голове.

Далее входим в цикл `while`. В первую очередь проверяем, равно ли условное выражение `True` или `False`.

```
scoops = 5
while scoops > 0:
    print('Еще шарик!')
    scoops = scoops - 1
print("Жизнь без мороженого скучна.")
```

Число шариков больше нуля? Для нас это выглядит так!

Поскольку условие равно `True`, начинаем выполнение тела цикла. Первая инструкция цикла выводит сообщение `"Еще шарик!"`.

```
scoops = 5
while scoops > 0:
    print('Еще шарик!')
    scoops = scoops - 1
print("Жизнь без мороженого скучна.")
```

Следующая инструкция вычитает единицу из значения scoops**, после чего оно становится равно** 4**.**

1 шарик убрали, 4 осталось!

```
scoops = 5
while scoops > 0:
    print('Еще шарик!')
    scoops = scoops - 1
print("Жизнь без мороженого скучна.")
```

Это была последняя инструкция в цикле, поэтому возвращаемся к условию и начинаем все заново.

Заметили, что сам код не меняется? А вот значение переменной меняется по ходу цикла. На данный момент переменная scoops равна 4

```
scoops = 5
while scoops > 0:
    print('Еще шарик!')
    scoops = scoops - 1
print("Жизнь без мороженого скучна.")
```

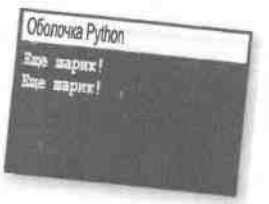
Повторно проверяем условие цикла. На этот раз переменная scoops **равна** 4**, т.е. больше** 0**.**

Еще много осталось!

```
scoops = 5
while scoops > 0:
    print('Еще шарик!')
    scoops = scoops - 1
print("Жизнь без мороженого скучна.")
```

В окне оболочки выводится еще одно сообщение "Еще шарик!"**.**

```
scoops = 5
while scoops > 0:
    print('Еще шарик!')
    scoops = scoops - 1
print("Жизнь без мороженого скучна.")
```



Следующая инструкция вычитает единицу из значения scoops**, после чего оно становится равно** 3.

2 шарика убрали,
3 осталось!

```
scoops = 5
while scoops > 0:
    print('Еще шарик!')
    scoops = scoops - 1
print("Жизнь без мороженого скучна.")
```

Это была последняя инструкция в цикле, поэтому возвращаемся к условию и начинаем все заново.

```
scoops = 5
while scoops > 0:
    print('Еще шарик!')
    scoops = scoops - 1
print("Жизнь без мороженого скучна.")
```

Повторно проверяем условие цикла. На этот раз переменная scoops **равна** 3**, т.е. больше** 0.

Еще много осталось!

```
scoops = 5
while scoops > 0:
    print('Еще шарик!')
    scoops = scoops - 1
print("Жизнь без мороженого скучна.")
```

В окне оболочки выводится еще одно сообщение "Еще шарик!".

```
scoops = 5
while scoops > 0:
    print('Еще шарик!')
    scoops = scoops - 1
print("Жизнь без мороженого скучна.")
```

Как видите, в цикле все время повторяется одно и то же. На каждом шаге цикла мы уменьшаем значение переменной `scoops` на 1, выводим сообщение на экран и продолжаем дальше.

3 шарика убрали, 2 осталось!

```
scoops = 5
while scoops > 0:
    print('Еще шарик!')
    scoops = scoops - 1
print("Жизнь без мороженого скучна.")
```

Продолжаем...

4 шарика убрали, 1 остался!

```
scoops = 5
while scoops > 0:
    print('Еще шарик!')
    scoops = scoops - 1
print("Жизнь без мороженого скучна.")
```

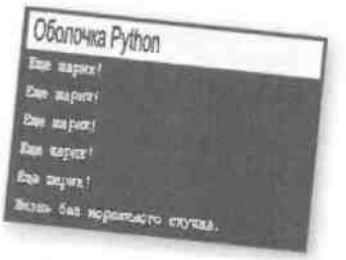
И вот мы достигли конца. На этот раз все по-другому. Переменная `scoops` равна `0`, а значит, условное выражение возвращает `False`. На этом все: больше в цикл мы не заходим, переходя к инструкции, которая расположена сразу после него.

5 шариков убрали, 0 осталось!

```
scoops = 5
while scoops > 0:
    print('Еще шарик!')
    scoops = scoops - 1
print("Жизнь без мороженого скучна.")
```

Теперь выполняется команда `print`, которая выводит сообщение `"Жизнь без мороженого скучна."`.

```
scoops = 5
while scoops > 0:
    print('Еще шарик!')
    scoops = scoops - 1
print("Жизнь без мороженого скучна.")
```



Напишите небольшую игру. Правила просты: вы спрашиваете у пользователя "Какой цвет я загадал?" и считаете, за сколько попыток он его угадает.

```
color = 'синий'
guess = ''
guesses = 0

while ____________________:
   guess = input('Какой цвет я загадал? ')
   guesses = guesses + 1
print('Угадали! Вы сделали', guesses, 'попытки')
```

Как использовать цикл while для повторного вывода запроса

Теперь, когда вы познакомились с циклом `while`, можно воспользоваться им для повторного вывода запроса. По сравнению с предыдущей неудачной попыткой мы **внесем** два простых изменения. Прежде всего, мы инициализируем переменную `user_choice` пустой строкой. А кроме того, мы заменим `if` на `while`.

Результат выглядит так:

Сначала записываем в переменную `user_choice` пустую строку

Мы записываем в переменную `user_choice` пустую строку, потому что в начале цикла `while` она должна иметь какое-то значение

```
user_choice = ''
while (user_choice != 'камень' and
       user_choice != 'бумага' and
       user_choice != 'ножницы'):
    user_choice = input('камень, ножницы или бумага? ')
```

Пока переменная `user_choice` не содержит корректную строку, продолжаем выполнять блок кода

На каждом шаге цикла просим пользователя сделать выбор и записываем этот выбор в переменную `user_choice`

Когда пользователь наконец делает правильный выбор, цикл `while` завершается, и в переменной `user_choice` оказывается записана введенная строка

Замените в файле *rock.py* вызов функции `input()` нашим новым циклом `while` и протестируйте полученный код. Если все корректно, то программу можно считать завершенной.

```
import random

winner = ''

random_choice = random.randint(0, 2)

if random_choice == 0:
    computer_choice = 'камень'
elif random_choice == 1:
    computer_choice = 'бумага'
else:
    computer_choice = 'ножницы'

user_choice = input('камень, ножницы или бумага? ')
user_choice = ''
while (user_choice != 'камень' and
       user_choice != 'бумага' and
       user_choice != 'ножницы'):
    user_choice = input('камень, ножницы или бумага? ')

if computer_choice == user_choice:
    winner = 'Ничья'
elif computer_choice == 'бумага' and user_choice == 'камень':
    winner = 'Компьютер'
elif computer_choice == 'камень' and user_choice == 'ножницы':
    winner = 'Компьютер'
elif computer_choice == 'ножницы' and user_choice == 'бумага':
    winner = 'Компьютер'
else:
    winner = 'Пользователь'

if winner == 'Ничья':
    print('Мы оба выбрали', computer_choice + ', играем снова.')
else:
    print(winner, 'выиграл, я выбрал', computer_choice + '.')
```

Удалите старую инструкцию ввода

Вот новый код для обработки пользовательского ввода. Мы продолжаем выводить запрос, пока пользователь не введет строку 'камень', 'ножницы' или 'бумага'

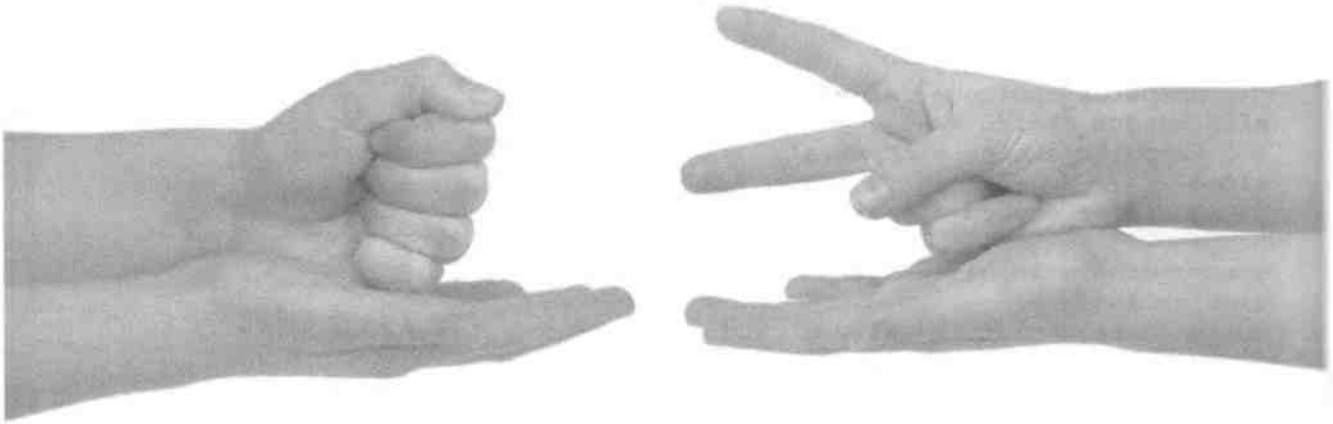
Теперь у нас полнофункциональная игра. Программа случайным образом делает выбор за компьютер и выводит запрос пользователю до тех пор, пока не получит корректную строку, после чего объявляет победителя (или ничью)

Оболочка Python

```
камень, ножницы или бумага? ножнецы
камень, ножницы или бумага? камень
Пользователь выиграл, я выбрал ножницы.
RESTART: /ch3/rock.py
камень, ножницы или бумага? бамага
камень, ножницы или бумага? камин
камень, ножницы или бумага? бумага
Компьютер выиграл, я выбрал ножницы.
```

Итак, наша первая игра готова!

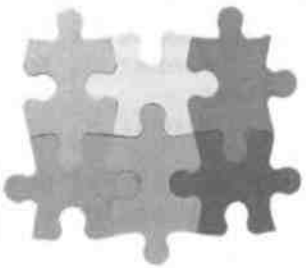
Что лучше всего сделать после написания новой игры? Конечно же, сыграть несколько розыгрышей! Сядьте поудобнее, расслабьтесь, и пусть все, о чем вы узнали в этой главе, разложится по полочкам, пока вы пытаетесь победить компьютер. Нам еще предстоит выполнить одно упражнение, подвести итоги и решить кроссворд, но пока можно просто насладиться игрой.



Еще кое-что

УПРАЖНЕНИЕ

Помните игру в угадывание цвета? В ней есть ошибка. Если цвет угадан с первой попытки, программа сообщит: "Угадали! Вы сделали 1 попытки". Исправьте код, чтобы при угадывании цвета с первого раза программа выводила слово "попытка" в единственном числе.



```
color = 'синий'
guess = ''
guesses = 0

while guess != color:
    guess = input('Какой цвет я загадал? ')
    guesses = guesses + 1
print('Угадали! Вы сделали', guesses, 'попытки')
```

Введите здесь свой код

Опасность:
УЖАСНЫЙ БЕСКОНЕЧНЫЙ ЦИКЛ

Прежде чем завершить главу, нужно поговорить о бесконечных циклах. В программах, лишенных циклических структур, все выполняется линейно, и вы рано или поздно достигаете последней инструкции программы. Однако в случае циклов ситуация не столь однозначна.

Предположим, вы написали программу и не ждете от нее никакого подвоха. Вы спокойно запускаете ее... Но что это?! Ничего не происходит! Похоже, программа что-то делает, но вы не понимаете, что именно. Как бы там ни было, все выполняется слишком долго.

Вы только что вошли в бесконечный цикл, т.е. цикл, который выполняется снова и снова, никогда не заканчиваясь.

Попасть в такую ситуацию намного проще, чем кажется. Это не настолько редкая ошибка, как выясняется. Рассмотрим следующий пример.

```
counter = 10

while counter > 0:
    print('Счетчик равен', counter)
    counter = counter + 1
print('Поехали!')
```

Попробуйте запустить программу

Оболочка Python

```
Счетчик равен 814304
Счетчик равен 814305
Счетчик равен 814306
Счетчик равен 814307
Счетчик равен 814308
Счетчик равен 814309
Счетчик равен 814310
```

Ой, кажется, все вышло из-под контроля!

Что делать в ситуации, когда программа вышла из-под контроля? Если вы работаете в редакторе IDLE, просто закройте окно интерпретатора, чтобы завершить программу. Если же вы работаете в командной строке, помогает обычно комбинация клавиш <Ctrl+C>.

Но как поступить с программным кодом? По сути, бесконечный цикл — это логическая ошибка. Вы написали конструкцию, которая никогда не позволит циклу завершиться, поэтому изучите условие цикла и постарайтесь выяснить, что с ним не так. В приведенном выше примере достаточно заменить строку `counter + 1` **на** `counter - 1`**, чтобы счетчик цикла уменьшался, а не увеличивался.**

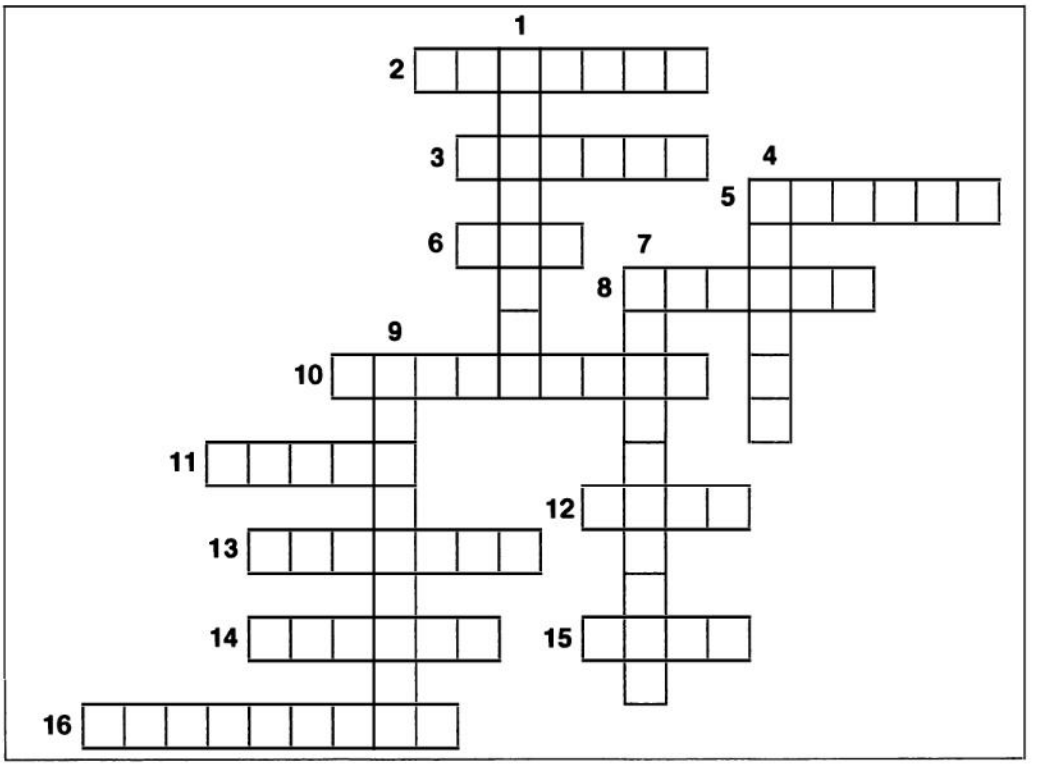
САМОЕ ГЛАВНОЕ

- Случайные числа широко применяются в программировании.
- Во всех языках программирования есть средства генерирования случайных чисел.
- В Python для работы со случайными числами предназначен модуль `random`.
- Чтобы подключить модуль `random` к программе, воспользуйтесь инструкцией `import`.
- Булев тип данных представлен всего двумя значениями: `True` и `False`.
- Булевы, или условные, выражения возвращают либо `True`, либо `False`.
- Условные операторы, такие как `==`, `>` и `<`, позволяют сравнивать два значения.
- Условные операторы можно применять как к числами, так и к строкам.
- Булевы выражения применяются в инструкции `if`.
- Инструкция `if` проверяет булево выражение, и, если оно равно `True`, выполняется блок кода.
- Блок кода — это набор инструкций, выполняемых вместе.
- Блоки кода выделяются в программе одинаковыми отступами.
- Ключевое слово `elif` позволяет проверять дополнительные условия в инструкции `if`.
- Ключевое слово `elif` представляет собой сокращение от "else if".
- Ключевое слово `else` позволяет задать финальную альтернативу в инструкции `if`.
- Булевы выражения можно объединять с помощью логических операторов `and` и `or`.
- Булев оператор `not` применяется для отрицания исходного логического условия.
- Блок кода, содержащийся в цикле `while`, выполняется до тех пор, пока истинно условие цикла.
- Строка, не содержащая ни одного символа, называется *пустой*.
- Оператор `=` означает присваивание, а оператор `==` — проверку на равенство.
- Комментарии в программный код добавляются после символа решетки (`#`).
- Комментарии позволяют документировать код, чтобы впоследствии вы смогли вспомнить, как была написана программа, и чтобы другие программисты могли разобраться в ней.
- Обработка неправильно введенных данных — важная задача при разработке пользовательских приложений, таких как игры.
- Причиной бесконечных циклов обычно становятся логические ошибки.

Кроссворд

Дайте передышку своему мозгу, загрузив правое полушарие чем-то увлекательным.

Перед вами кроссворд, состоящий из слов, которые встречались на протяжении главы 3.



По горизонтали

2. Символ начала комментария
3. Один из знаков в игре "Камень, ножницы, бумага"
5. Операция подключения модуля
6. Формат данных
8. Строка, лишенная символов
10. Операция сопоставления значений
11. Логический тип данных
12. Одно из булевых значений
13. Еще один знак в игре "Камень, ножницы, бумага"
14. Выделяет блок кода
15. Программная конструкция, обеспечивающая многократное выполнение инструкций
16. Произвольное значение

По вертикали

1. Исходное название игры "Камень, ножницы, бумага"
4. Одно из булевых значений
7. Запись значения в переменную
9. Идентичность значений

УПРАЖНЕНИЕ (РЕШЕНИЕ)

Проведите небольшой тест. Откройте оболочку Python, импортируйте модуль `random` и сгенерируйте несколько случайных чисел с помощью вызова `random.randint(0, 2)`.

Ваш вывод будет немного другим, потому что это случайные числа! Возможно, вы увидите 2, 2, 0, 0, или 1, 2, 0, 1, или 1, 1, 1, 1 (так тоже бывает!), или еще что-то. В этом вся прелесть случайных чисел!

Оболочка Python

```
>>> import random
>>> random.randint(0, 2)
1
>>> random.randint(0, 2)
0
>>> random.randint(0, 2)
1
>>> random.randint(0, 2)
2
>>>
```

Возьмите карандаш

Решение

Предполагая, что переменная random_choice равна 0, 1 или 2, напишите псевдокод, в котором переменной `computer_choice` присваивается значение `"камень"`, `"ножницы"` или `"бумага"` в зависимости от значения `random_choice`.

Если `random_choice` *равно 0, то присвоить* `computer_choice` *значение* `"камень"`

иначе, если `random_choice` *равно 1, то присвоить* `computer_choice` *значение* `"бумага"`,

иначе присвоить `computer_choice` *значение* `"ножницы"`

Возьмите карандаш
Решение

Пришло время взять карандаш и немного поупражняться. Вычислите значение каждого из приведенных ниже выражений и впишите свой ответ, после чего сверьтесь с ответами, приведенными в конце главы. Помните: булевы выражения всегда равны `True` либо `False`.

Проверяем, будет ли первое значение больше, чем второе. Можно также использовать оператор >=, чтобы проверить, будет ли первое значение больше или равно второму

```
your_level > 5
```

Если `your_level=2`, то чему равно выражение? **False**

Если `your_level=5`, то чему равно выражение? **False**

Если `your_level=7`, то чему равно выражение? **True**

Оператор == проверяет равенство двух значений. Если они равны, то результат True, иначе — False

```
color == "orange"
```

Если переменная `color` содержит `"pink"`, то выражение равно `True` или `False`? **False**

А если переменная содержит `"orange"`? **True**

Оператор != проверяет, будут ли значения НЕ равны друг другу

```
color != "orange"
```

Если переменная `color` содержит `"pink"`, то выражение равно `True` или `False`? **True**

Возьмите карандаш
Решение

В приведенной ниже таблице указано несколько возможных значений переменной `number_of_scoops`. Выпишите для каждого из них соответствующее сообщение, выдаваемое программой. Первый результат мы вписали за вас.

При таком значении number_of_scoops выводится...	
`number_of_scoops = 0`	*Не желаете мороженое? У нас много разных сортов.*
`number_of_scoops = 4`	*Ого, это много шариков!*
`number_of_scoops = 1`	*Один шарик, сейчас сделаем.*
`number_of_scoops = 3`	*Ого, это много шариков!*
`number_of_scoops = 2`	*Прекрасно, два шарика!*
`number_of_scoops = -1`	*Простите, но у нас нет отрицательных шариков.*

Возьмите карандаш
Решение

Прочитайте вслух приведенный выше код, переведя его на родной язык, а затем запишите, что у вас получилось.

Если число шариков равно 0, вывести `'Не желаете мороженое? У нас много разных сортов.'`

Иначе если число шариков равно 1, вывести `'Один шарик, сейчас сделаем.'`

Иначе если число шариков равно 2, вывести `'Прекрасно, два шарика!'`

Иначе если число шариков больше или равно 3, вывести `'Ого, это много шариков!'`

Иначе вывести `"Простите, но у нас нет отрицательных шариков."`

Возьмите карандаш
Решение

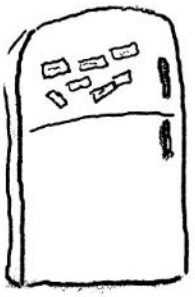
Снова ваша очередь проявить себя. Завершите приведенный ниже код согласно нашему плану. Программа должна определять ничью и присваивать переменной `winner` значение `'Ничья'`.

```
if computer_choice == user_choice :
    winner = 'Ничья'
```

Возьмите карандаш
Решение

Возьмите карандаш и запишите, чему равно каждое из приведенных ниже булевых выражений.

`age > 5 and age < 10`	Если `age=6`, то чему равно выражение?	True
	Если `age=11`, то чему равно выражение?	False
	Если `age=5`, то чему равно выражение?	False
`age > 5 or age == 3`	Если `age=6`, то чему равно выражение?	True
	Если `age=2`, то чему равно выражение?	False
	Если `age=3`, то чему равно выражение?	True
`not age > 5`	Если `age=6`, то чему равно выражение?	False
	Если `age=2`, то чему равно выражение?	True



Код на магнитиках: решение

Мы составили код игры "Камень, ножницы, бумага" с помощью набора магнитиков, прикрепленных к дверце холодильника, но кто-то пришел и сбросил их на пол. Сможете расположить магнитики в правильном порядке, чтобы нам удалось определить победителя? Учтите, что некоторые магнитики могут оказаться лишними.

```
if computer_choice == user_choice :
    winner = 'Ничья'
elif computer_choice == 'бумага' and user_choice == 'камень' :
    winner = 'Компьютер'
elif computer_choice == 'камень' and user_choice == 'ножницы' :
    winner = 'Компьютер'
elif computer_choice == 'ножницы' and user_choice == 'бумага' :
    winner = 'Компьютер'
else :
    winner = 'Пользователь'
```

УПРАЖНЕНИЕ (РЕШЕНИЕ)

Добавьте в редакторе IDLE комментарии к файлу *rock.py*. Используйте пояснения на предыдущей странице в качестве образца, но можете вводить и свои комментарии. Не забывайте, что комментарии должны быть понятны любому (включая вас самого!), кто в будущем захочет просмотреть исходный код программы.

Это наша попытка задокументировать игру "Камень, ножницы, бумага"

```
# КАМЕНЬ, НОЖНИЦЫ, БУМАГА
# Игра шоушилин, изобретенная в Китае во времена династии
# Хань, сегодня известна как "Камень, ножницы, бумага".
# Эта версия игры рассчитана на одного игрока, оппонентом
# которого выступает компьютер.

# Начинаем с того, что импортируем модуль random
# и объявляем переменную winner

import random

winner = ''

# Компьютер делает произвольный выбор, генерируя
# случайное целое число в диапазоне от 0 до 2,
# а затем сопоставляя его с соответствующей строкой

random_choice = random.randint(0, 2)

if random_choice == 0:
    computer_choice = 'камень'
elif random_choice == 1:
    computer_choice = 'бумага'
else:
    computer_choice = 'ножницы'

# С помощью функции input() запрашиваем выбор пользователя

user_choice = input('камень, ножницы или бумага? ')

# Здесь реализуется логика игры: мы определяем победителя
# и соответствующим образом меняем переменную winner

if computer_choice == user_choice:
    winner = 'Ничья'
elif computer_choice == 'бумага' and user_choice == 'камень':
    winner = 'Компьютер'
elif computer_choice == 'камень' and user_choice == 'ножницы':
    winner = 'Компьютер'
elif computer_choice == 'ножницы' and user_choice == 'бумага':
    winner = 'Компьютер'
else:
    winner = 'Пользователь'

# Здесь мы объявляем либо ничью, либо победителя
# и сообщаем о выборе компьютера

if winner == 'Ничья':
    print('Мы оба выбрали', computer_choice + ', играем снова.')
else:
    print(winner, 'выиграл, я выбрал', computer_choice + '.')
```

Возьмите карандаш

Решение

Запишите для этих фраз соответствующие булевы выражения. Первую часть мы сделали за вас.

```
user_choice != 'камень' and user_choice != 'бумага' and user_choice != 'ножницы'
```

Возьмите карандаш

Решение

Пока вы раздумывали над проблемой с верхним и нижним регистром, мы забежали немного наперед и написали код для вывода повторного запроса. Только в наше решение закралась ошибка. Сможете ли вы определить, в чем ее причина? Ниже показан пример работы программы вместе с нашими комментариями.

```
user_choice = input('камень, ножницы или бумага? ')
if (user_choice != 'камень' and
      user_choice != 'бумага' and
      user_choice != 'ножницы'):
   user_choice = input('камень, ножницы или бумага? ')
print('Пользователь выбрал', user_choice)
```

Сначала просим пользователя сделать выбор

Затем проверяем, введена ли допустимая строка

Если нет, просим пользователя повторить

Мы думали, что все сделали правильно, так что же мы упустили?

Оболочка Python

```
камень, ножницы или бумага? бамага
камень, ножницы или бумага? камин
Пользователь выбрал камин
>>>
```

Выводим повторный запрос

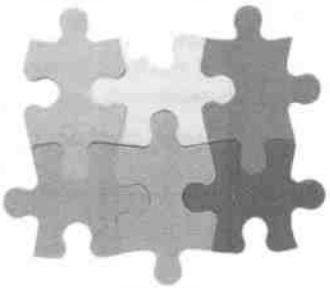
Но затем снова получаем неверную строку

Не то!

АНАЛИЗ: *с одной стороны, программа прекрасно справляется с определением неправильно введенных данных, а с другой — повторный запрос выводится всего один раз. Нужно придумать, как выводить такой запрос до тех пор, пока не будет введено допустимое значение.*

УПРАЖНЕНИЕ (РЕШЕНИЕ)

Напишите небольшую игру. Правила просты: вы спрашиваете у пользователя "Какой цвет я загадал?" и считаете, за сколько попыток он его угадает.



```
color = 'синий'
guess = ''
guesses = 0

while guess != color:
    guess = input('Какой цвет я загадал? ')
    guesses = guesses + 1
print('Угадали! Вы сделали', guesses, 'попыток')
```

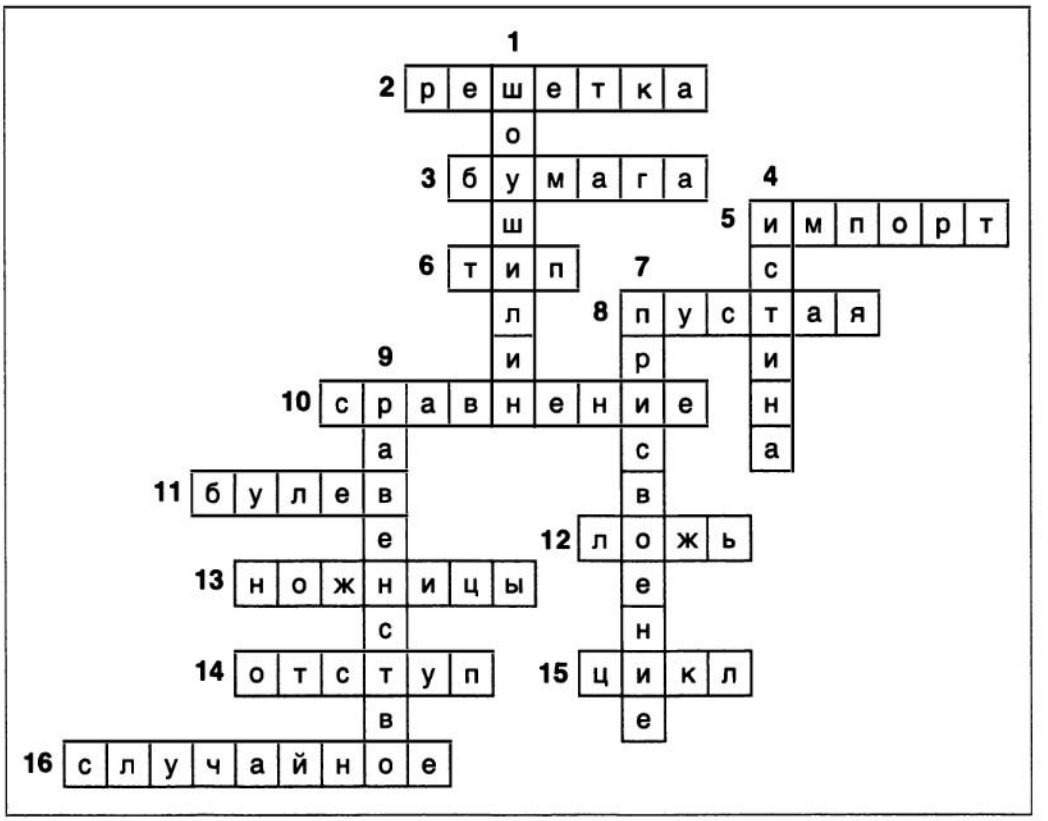
Еще кое-что

УПРАЖНЕНИЕ (РЕШЕНИЕ)

Помните игру в угадывание цвета? В ней есть ошибка. Если цвет угадан с первой попытки, программа сообщит: "Угадали! Вы сделали 1 попытки". Исправьте код, чтобы при угадывании цвета с первого раза программа выводила слово "попытка" в единственном числе.

```
color = 'синий'
guess = ''
guesses = 0

while guess != color:
    guess = input('Какой цвет я загадал? ')
    guesses = guesses + 1
if guesses == 1:
    print('Угадали! Вы сделали 1 попытку')
else:
    print('Угадали! Вы сделали', guesses, 'попытки')
```

Кроссворд: решение



4 списки и циклы

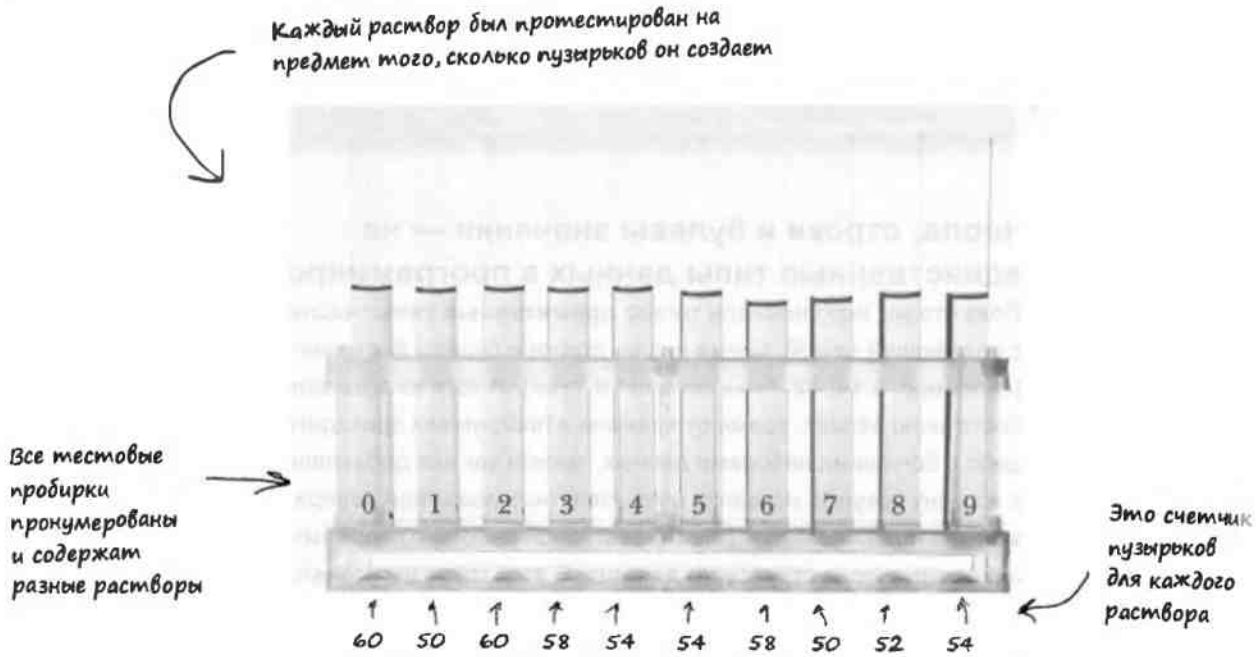
Структуры данных

Числа, строки и булевы значения — не единственные типы данных в программировании.

Пока что мы использовали только **примитивные типы**: числа с плавающей точкой, целые числа, строки и булевы значения (например, `3.14`, `42`, `"имя собаки"` и `True`). И хотя их возможности достаточно велики, время от времени в программах приходится иметь дело с большими наборами данных, такими как все добавленные в корзину покупки, названия всех известных звезд или содержимое всего каталога товаров. Для эффективной работы с подобными наборами нужны **структуры данных**. В этой главе вы познакомитесь с новым типом данных — **списком**, хранящим коллекцию значений. Списки позволяют структурировать данные, вместо того чтобы хранить каждый элемент в отдельной переменной. Вы узнаете о том, как работать со списком в целом и как поочередно перебирать его элементы в цикле `for`. Все это расширит ваши возможности по работе с данными.

Поможете разобраться?

Нас ждет путешествие в офис компании "Пузырь-ОК", которая проводит исследования пузырьковых растворов, стремясь сделать их максимально эффективными. Сегодня они тестируют "пузырьковую мощность" нескольких вариантов формул своего нового раствора. Другими словами, они оценивают, сколько пузырьков можно получить из каждого раствора. Вот результаты тестового стенда.



Разумеется, полученные при проведении опытов данные нужно будет передать Python для программной обработки. Но исходных данных слишком много. Как обрабатывать их в коде?

Сохранение нескольких значений в одной переменной

Вы уже знаете, как работать с отдельными числами, строками и булевыми значениями в Python. Но как представить *набор* значений, например все оценки, полученные пузырьковыми растворами? Для этого в Python существуют **списки** — структуры данных, включающие сразу несколько значений. Вот как будет выглядеть такая структура данных для нашего примера.

Во многих языках программирования упорядоченные структуры данных называются массивами, а не списками

```
scores = [60, 50, 60, 58, 54, 54, 58, 50, 52, 54]
```

Мы включили в список числа, но на их месте могут быть любые значения

Это список из 10 значений, записанный в переменную scores

Подобные типы называются структурами данных, поскольку они позволяют упорядочивать хранение данных

Список поддерживает возможность доступа к отдельным его *элементам*. У каждого элемента списка имеется собственный индекс. Компьютерные ученые любят нумеровать числа, начиная с нуля, поэтому первый элемент списка имеет индекс 0. Извлечь элемент списка можно следующим образом.

Для доступа к элементу списка используется следующий синтаксис: после имени переменной списка указывается индекс элемента в квадратных скобках

Индексация ведется с нуля, поэтому первый элемент списка имеет индекс 0, второй — 1 и т.д.

Помните, мы говорили о том, что компьютерщики любят все нумеровать с нуля

```
score = scores[3]
print('Раствор #3 выдал пузырьков:', score)
```

Оболочка Python

```
Раствор #3 выдал пузырьков: 58
>>>
```

Мой раствор номер 3 точно будет лучшим.

Один из исследователей компании "Пузырь-ОК"

Работа со списками

Похоже, нас ждет интересный проект для компании "Пузырь-ОК", но, прежде чем начать, давайте разберемся со списками и узнаем, как вместо числовых оценок работать со строковыми значениями. Причем это будут названия напитков! Как только все прояснится, мы сможем помочь руководству компании.

Итак, мы знаем, что набор значений можно представить в виде списка, сохранив возможность доступа к отдельным его элементам. Как правило, в списках хранятся однотипные элементы, например числовые показатели, названия продуктов, временные ряды или результаты анкетирования с вариантами ответов "да/нет". Рассмотрим еще раз процедуру создания списка, уделив особое внимание синтаксису.

Создание списка

Предположим, необходимо создать список, хранящий названия напитков. Вот как это делается:

Каждый элемент списка отделяется запятой

```
smoothies = ['кокосовый', 'клубничный', 'банановый', 'ананасовый', 'ягодный']
```

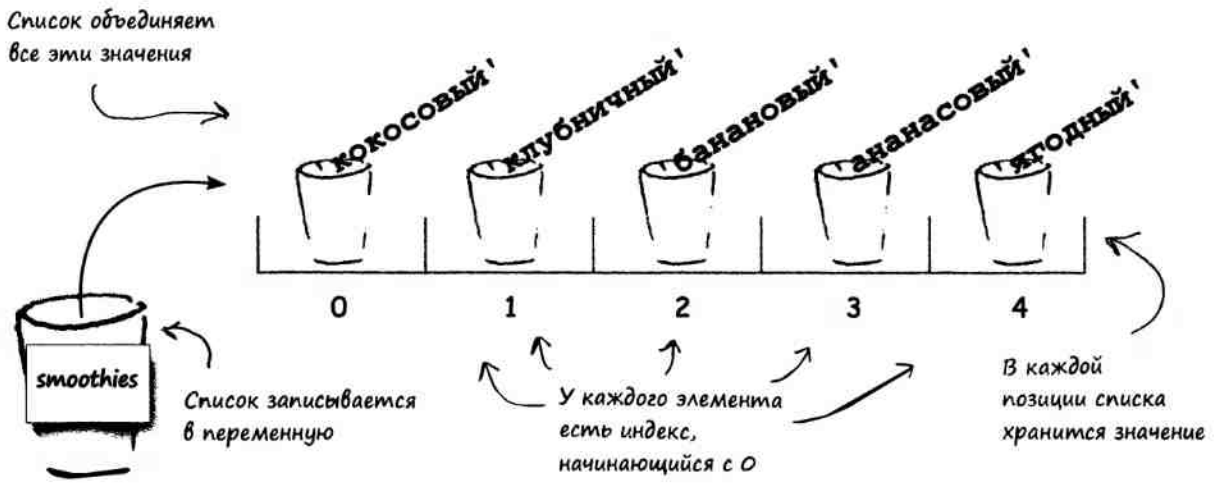
Список записывается в переменную smoothies

Начало списка помечается открывающей квадратной скобкой

Далее перечисляются элементы списка

В конце ставится закрывающая квадратная скобка

Как уже говорилось, положение каждого элемента в списке задается его индексом. В представленном выше списке элементу `'кокосовый'` соответствует индекс `0`, элементу `'клубничный'` — индекс `1` и т.д. Ниже представлена общая структура хранения списка.



Работа с элементами списка

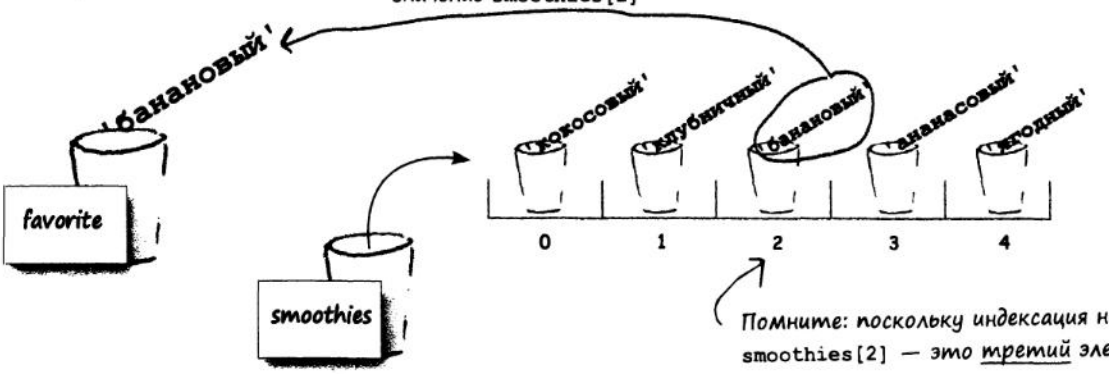
У каждого элемента списка есть индекс, по которому к нему можно получить доступ. Вы уже знаете, что для обращения к элементу списка необходимо указать имя списка, а затем — индекс элемента, заключенный в квадратные скобки. Подобную нотацию можно применять везде, где разрешается указывать имя переменной.

```
favorite = smoothies[2]
```

Данное значение соответствует элементу списка smoothies с индексом 2 ('банановый'); этот элемент записывается в переменную favorite

Чтобы извлечь элемент из списка, необходимо указать переменную списка, а также индекс требуемого элемента

Переменной favorite присваивается значение smoothies[2]



Помните: поскольку индексация начинается с 0, smoothies[2] — это третий элемент списка

Обновление значений в списке

Значение элемента списка можно изменить, подобно обычной переменной.

```
smoothies[3] = 'тропический'
```

В элемент списка с индексом 3 (ранее 'ананасовый') записывается новое значение: 'тропический'

С этого момента список напитков выглядит так:

Значение элемента с индексом 3 изменилось

Возьмите карандаш

Давайте попрактикуемся в обработке элементов списка. Внимательно изучите приведенный ниже код и постарайтесь определить результат его выполнения интерпретатором Python.

```
eighties = ['', 'duran duran', 'B-52s', 'a-ha']
newwave = ['flock of seagulls', 'secret service']

remember = eighties[1]

eighties[1] = 'culture club'

band = newwave[0]

eighties[3] = band

eighties[0] = eighties[2]

eighties[2] = remember

print(eighties)
```

Размер списка

Предположим, в вашем распоряжении оказался очень длинный список. Вы умеете извлекать данные из списка, но совершенно не представляете, насколько он большой. К счастью, в Python есть встроенная функция для подсчета количества элементов списка. Она называется `len()` и имеет следующий синтаксис.

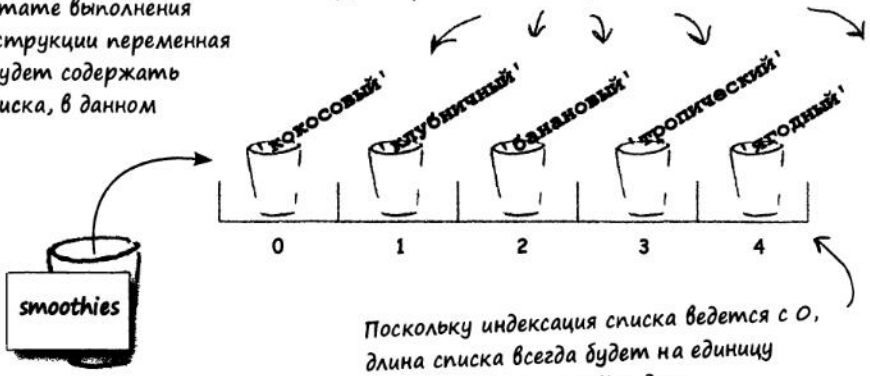
С помощью функции `len()` можно узнать текущее число элементов списка

Просто передайте функции `len()` список, длину которого требуется определить

```
length = len(smoothies)
```

В результате выполнения этой инструкции переменная `length` будет содержать длину списка, в данном случае 5

Длина равна 5, потому что в списке пять элементов



Поскольку индексация списка ведется с 0, длина списка всегда будет на единицу больше, чем последний индекс

Такая особенность характерна для большинства языков программирования

Сила мысли

Как, зная длину списка, получить значение его последнего элемента?

Извлечение последнего элемента списка

Получение последнего элемента списка — распространенная задача в программировании. Предположим, у вас имеется список с результатами соревнований и вы хотите узнать результат, показанный последним участником. Или, допустим, вы фиксируете скорость ветра приближающегося урагана и хотите сообщить его текущую скорость. Думаю, вы поняли идею: данные в списках часто хранятся в упорядоченном виде, когда самое новое (либо самое важное) значение располагается в конце, имея наибольший индекс.

Общепринятый способ решения этой задачи заключается в использовании длины списка в качестве индекса. Но поскольку в Python нумерация элементов списка начинается с 0, индекс последнего элемента будет на единицу меньше длины списка. Соответственно, код получения последнего элемента списка выглядит так.

```
length = len(smoothies)
last = smoothies[length-1]
print(last)
```

Это распространенный прием во многих языках: определить длину списка, а затем отнять единицу, чтобы получить индекс последнего элемента

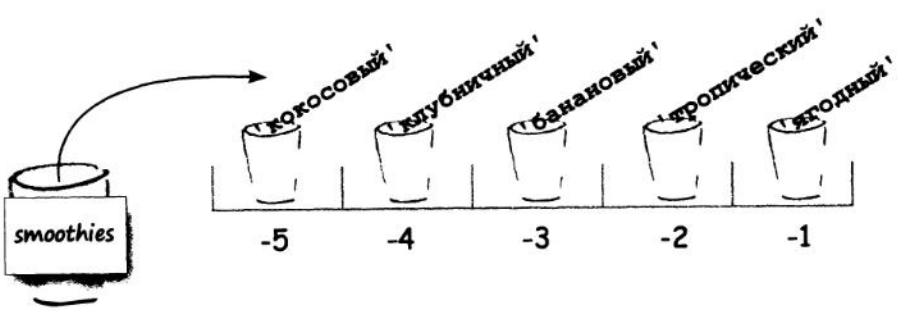
Оболочка Python

```
ягодный
>>>
```

В Python все намного проще

Извлечение последнего элемента списка стало настолько обыденной задачей, что для ее решения в Python предусмотрена специальная методика, которая заключается в индексации элементов списка отрицательными числами: последний элемент списка обозначается индексом –1, предпоследний — индексом –2 и т.д. в обратном порядке до самого первого элемента.

Python поддерживает отрицательные индексы в виде смещения от КОНЦА списка. Соответственно, -1 — индекс последнего элемента, -2 — предпоследнего, -3 — третьего с конца и т.д. Подобной возможности нет в большинстве языков программирования



Вот как выглядят отрицательные индексы при отсчете от конца списка

Отрицательные индексы

Давайте проверим работу отрицательных индексов на следующем простом примере. Предположим, мы хотим получить названия трех последних напитков из нашего списка.

```
last = smoothies[-1]
second_last = smoothies[-2]
third_last = smoothies[-3]
print(last)
print(second_last)
print(third_last)
```

Индекс -1 соответствует последнему элементу списка

Индекс -2 соответствует предпоследнему элементу

Индекс -3 соответствует третьему с конца элементу

Выведем все на экран

Оболочка Python

```
ягодный
тропический
банановый
>>>
```

Супержелезяка

Супержелезяка — это хитроумная штуковина, которая стучит, гремит и даже гудит. Но что она делает? Мы и сами в замешательстве. Программисты утверждают, будто знают, как она работает. А вы сможете разобраться в ее коде?

Это можно рассматривать как упражнение на построение строк. Потратьте какое-то время, чтобы потренировать мозг

```
characters = ['t', 'a', 'c', 'o']

output = ''
length = len(characters)
i = 0
while (i < length):
    output = output + characters[i]
    i = i + 1

length = length * -1
i = -2

while (i >= length):
    output = output + characters[i]
    i = i - 1

print(output)
```

Если считаете, что поняли работу программы, то запишите здесь свой ответ и сверьтесь с решением в конце главы

Попробуйте альтернативные варианты списка символов:

characters = ['a', 'm', 'a', 'n', 'a', 'p', 'l', 'a', 'n', 'a', 'c']

или

characters = ['w', 'a', 's', 'i', 't', 'a', 'r']

Читайте комментарии к коду в конце главы

Не бойтесь задавать вопросы

В: Насколько важен порядок указания элементов списка?

О: Список — упорядоченный тип данных, поэтому порядок элементов в нем обычно важен, но далеко не всегда. В нашем списке для компании "Пузырь-ОК" порядок играет роль, поскольку индекс элемента соответствует номеру раствора в пробирке. К примеру, раствор с номером 0 выдал результат 60. Если перетасовать результаты, данные эксперимента будут разрушены! Тем не менее во многих случаях порядок элементов не важен. Скажем, если составить список продуктов для покупки, то вряд ли для вас важно, в какой последовательности их покупать. Все зависит от того, для чего создается список. Кроме того, в Python имеются неупорядоченные структуры данных — словари и коллекции, о которых мы поговорим в последующих главах.

В: Насколько большим может быть список?

О: Теоретически список может вмещать любое количество элементов. На практике его длина ограничивается объемом оперативной памяти компьютера, ведь каждый элемент списка занимает какое-то место в памяти, и если их добавлять в неограниченном количестве, то память может просто закончиться. Тем не менее, в зависимости от типа значений, список может включать от нескольких тысяч до нескольких сотен тысяч элементов. Если же задача требует включения в список более миллиона значений, имеет смысл поискать другие решения (например, базы данных).

В: Можно ли создать пустой список, лишенный элементов?

О: Помните, мы говорили о пустых строках? Да, пустые списки вполне допустимы, и вы познакомитесь с примерами их использования в этой главе. Пустой список создается так:

```
empty_list = []
```

Помните, что в пустой список всегда можно добавить элементы.

В: Мы видели списки, содержащие числа и строки. Какие еще типы данных можно хранить в них?

В списки можно включать любые поддерживаемые в Python типы данных (в том числе те, с которыми мы еще не успели познакомиться).

Даже другие списки!

В: Должны ли все элементы списка иметь один и тот же тип данных?

О: В Python нет такого ограничения. Списки, содержащие элементы различных типов, называются *неоднородными*. Приведем пример:

```
heterogenous = ['blue', True, 13.5]
```

В: Что произойдет при обращении к несуществующему элементу списка?

О: Если, к примеру, попытаться извлечь 99-й элемент в списке, состоящем из 10 элементов, будет получена ошибка выполнения:

```
IndexError: list index out of range
```

В: А можно ли назначить значение элементу списка с несуществующим индексом?

О: Нет. Если элемента с таким индексом нет в списке, то ему нельзя присвоить значение. Будет получена ошибка выполнения "out of bounds". В некоторых языках такое все же поддерживается, но не в Python. Сначала необходимо добавить новый элемент в список.

Возьмите карандаш

Приведенные ниже названия напитков добавлялись в список в порядке создания. Завершите код программы, определяющей название самого последнего напитка.

```
smoothies = ['кокосовый', 'клубничный', 'банановый', 'ананасовый', 'ягодный']
most_recent = _______________
recent = smoothies[most_recent]
```

Это можно сделать двумя способами: с помощью функции `len()` и без нее. Сможете найти оба?

Продолжим подсчет мыльных пузырьков

Генеральный директор компании "Пузырь-ОК"

```
scores = [60, 50, 60, 58, 54, 54,
          58, 50, 52, 54, 48, 69,
          34, 55, 51, 52, 44, 51,
          69, 64, 66, 55, 52, 61,
          46, 31, 57, 52, 44, 18,
          41, 53, 55, 61, 51, 44]
```

Новые результаты

Что нужно получить

Пузырь-ОК

Мне очень нужен этот отчет, чтобы принимать быстрые решения о том, какой раствор выпускать. Сможете написать?

— генеральный директор компании "Пузырь-ОК"

```
Пузырьковый раствор #0 - результат: 60
Пузырьковый раствор #1 - результат: 50
Пузырьковый раствор #2 - результат: 60
```

Остальные результаты...

```
Пузырьковых тестов: 36
Наибольший результат: 69
Растворы с наибольшим результатом: [11, 18]
```

Попробуем разобраться с требованиями.

Пузырь-ОК

Мне очень нужен этот отчет, чтобы принимать быстрые решения о том, какой раствор выпускать. Сможете написать?

— генеральный директор компании "Пузырь-ОК"

Нам нужно начать со списка растворов и их результатов

```
Пузырьковый раствор #0 - результат: 60
Пузырьковый раствор #1 - результат: 50
Пузырьковый раствор #2 - результат: 60
```

Остальные результаты...

Далее нужно вывести общее число тестов...

```
Пузырьковых тестов: 36
Наибольший результат: 69
Растворы с наибольшим результатом: [11, 18]
```

...а после этого — сообщить наибольший результат и указать, в каких растворах он был достигнут

Сила мысли

Настал ваш черед продемонстрировать навыки составления псевдокода. Напишите псевдокод программы, создающей отчет о результатах пузырьковых тестов. Поразмышляйте над структурой отчета и подумайте, как будет генерироваться каждый его элемент.

Подойдите к выполнению задания предельно ответственно и постарайтесь самостоятельно написать как можно больше строк псевдокода. Далее мы поработаем над составлением отчета вместе.

Офисный диалог

Юлия. Первое, что нужно сделать, – отобразить все результаты с указанием номеров растворов.

Игорь. А номер раствора – это ведь индекс его результата в списке, правильно?

Юлия. Именно так.

Федор. Минуточку! Это значит, что мы должны вывести индекс результата, являющийся номером раствора, а затем отобразить соответствующий ему результат.

Юлия. Все правильно, каждый результат – это отдельный элемент списка.

Игорь. То есть для раствора с порядковым номером 10 его результат хранится в элементе `scores[10]`.

Юлия. Так и есть.

Федор. Но ведь результатов целый список. Как вывести их все?

Юлия. Перебором элементов в цикле.

Федор. Ты имеешь в виду цикл `while`?

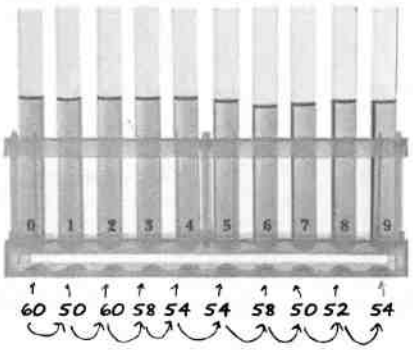
Юлия. Ну да. Мы создадим цикл от 0 до значения длины списка... То есть, я хотела сказать, до значения длины списка минус один, конечно же.

Игорь. Похоже, все прояснилось. Можем приступать к написанию кода.

Юлия. Уже не терпится! Сначала решим эту задачу, а потом вернемся к остальной части отчета.

Просмотр элементов списка

Итак, нашей целью будет получить результат, подобный показанному ниже.




```
Пузырьковый раствор #0 - результат: 60
Пузырьковый раствор #1 - результат: 50
Пузырьковый раствор #2 - результат: 60
.
.
.
Пузырьковый раствор #35 - результат: 44
```

Здесь перечисляются растворы с 3 по 34 (мы решили сберечь парочку деревьев и сократить вывод)

Сначала мы выводим значение, хранящееся в элементе с индексом 0, а затем повторяем эту же операцию для элементов с индексами 1, 2, 3 и т.д. до окончания списка. Вам уже известен цикл `while`, поэтому воспользуемся им.

Вскоре вы узнаете способ получше

```
scores = [60, 50, 60, 58, 54, 54, 58, 50, 52, 54, 48, 69,
          34, 55, 51, 52, 44, 51, 69, 64, 66, 55, 52, 61,
          46, 31, 57, 52, 44, 18, 41, 53, 55, 61, 51, 44]

i = 0

length = len(scores)

while i < length:

  print('Пузырьковый раствор #', i, '- результат:', scores[i])

  i = i + 1
```

Создаем переменную для отслеживания текущего индекса; начинаем с 0

Определяем длину списка результатов

Здесь не нужна запись `length - 1`, поскольку используется оператор <

Обходим в цикле все элементы, пока наш индекс меньше, чем длина списка

Мы используем переменную `i` для указания текущего раствора. Эта переменная также служит индексом в списке результатов

Наконец, увеличиваем индекс `i` на единицу, прежде чем начинать новую итерацию

Быстрый тест-драйв

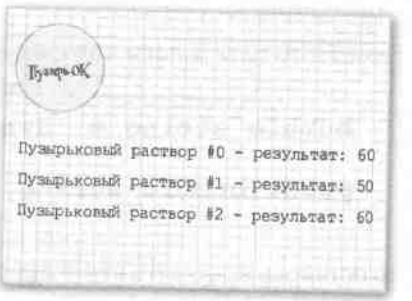
Сохраните код в файле *bubbles.py* и запустите его, чтобы понять, все ли мы сделали правильно.

Вот что мы получили. Неплохо!

```
Оболочка Python
Пузырьковый раствор # 0 - результат: 60
Пузырьковый раствор # 1 - результат: 50
Пузырьковый раствор # 2 - результат: 60
 ...
Пузырьковый раствор # 34 - результат: 51
Пузырьковый раствор # 35 - результат: 44
>>>
```

А вот этот пробел лишний

Но есть одна мелочь. Вы заметили, что после знака решетки стоит лишний пробел? В версии директора его нет



Устранение ошибки вывода

Внимательно изучите следующую инструкцию и постарайтесь определить, откуда берется лишний пробел.

Как вы уже знаете, каждая запятая, разделяющая аргументы функции `print()`, заменяется пробелом

```
print('Пузырьковый раствор #', i, '- результат:', scores[i])
```

Чтобы избавиться от лишних пробелов, разобьем код на две инструкции.

```
bubble_string = 'Пузырьковый раствор #' + i
print(bubble_string, '- результат:', scores[i])
```

Сначала конкатенируем строку `'Пузырьковый раствор #'` с переменной `i`, а затем передаем результат функции `print()`

Поначалу кажется, что простая конкатенация строки `'Пузырьковый раствор #'` и переменной `i` позволит легко избавиться от лишнего пробела, но в реальности этот код не работает. В чем здесь ошибка?

Правильный способ вывода результата

Выяснили причину ошибки? Дело в том, что нельзя выполнять конкатенацию строки и числа. Как же преобразовать целое число в строку? Помните, нам уже приходилось делать обратное — преобразовывать строку в целое число с помощью функции `int()`? Существует и парная к ней функция `str()`, которая получает целое число и возвращает его в строковом представлении.

С учетом этого перепишем код следующим образом.

```
bubble_string = 'Пузырьковый раствор #' + str(i)
print(bubble_string, '- результат:', scores[i])
```

Все, что нам нужно сделать, — это передать целое число i функции str(), чтобы преобразовать его в строку

Теперь осталось включить это в нашу программу, только нужно убрать лишнюю переменную `bubble_string`. Вместо этого мы сделаем код более компактным, вызвав функцию `str()` непосредственно в списке аргументов функции `print()`. Указанные изменения описаны во врезке "Быстрый тест-драйв".

Быстрый тест-драйв

Обновим исходный код, внеся в него рассмотренные выше исправления.

```
scores = [60, 50, 60, 58, 54, 54, 58, 50, 52, 54, 48, 69,
          34, 55, 51, 52, 44, 51, 69, 64, 66, 55, 52, 61,
          46, 31, 57, 52, 44, 18, 41, 53, 55, 61, 51, 44]

i = 0
length = len(scores)
while i < length:
  print('Пузырьковый раствор #' + str(i), '- результат:', scores[i])
  i = i + 1
```

Мы конкатенируем строку 'Пузырьковый раствор #' со значением i, прежде чем передавать ее функции print(). Функция str() нужна, чтобы получить строковое представление числа

Теперь лучше!

```
Оболочка Python
Пузырьковый раствор #0 - результат: 60
Пузырьковый раствор #1 - результат: 50
Пузырьковый раствор #2 - результат: 60
...
Пузырьковый раствор #34 - результат: 51
Пузырьковый раствор #35 - результат: 44
>>>
```

Код на магнитиках

Пора выполнить небольшое упражнение. Мы написали код, чтобы узнать, в каких напитках содержится кокос. Код был записан с помощью набора магнитиков на дверце холодильника, но они упали на пол. Постарайтесь все восстановить. Только будьте внимательны: некоторые магнитики лишние. Сверьтесь с ответом, приведенным в конце главы.

```
while i < len(has_coconut)
```

```
:
```

```
:
```

```
:
```

```
i = i + 2
```

```
i = i + 1
```

```
i = 0
```

```
if has_coconut[i]
```

```
while i > len(has_coconut)
```

```
smoothies = ['кокосовый',
             'клубничный',
             'банановый',
             'тропический',
             'ягодный']
```

```
has_coconut = [True,
               False,
               False,
               True,
               False]
```

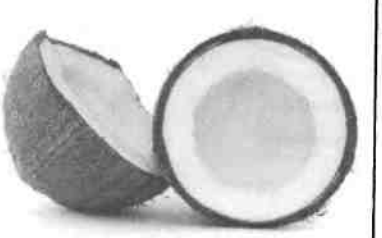
```
print(smoothies[i],'содержит кокос')
```

Расставьте здесь магнитики в нужном порядке

Вот какой результат мы ожидаем

Оболочка Python

```
кокосовый содержит кокос
тропический содержит кокос
>>>
```



Цикл for предпочтителен при работе со списками

В рассмотренных ранее примерах элементы списка поочередно обрабатывались в цикле `while`, однако предпочтительнее применять для этого цикл `for`. Эти циклы во многом похожи и делают примерно одно и то же, только цикл `while` выполняется, пока соблюдается определенное условие, а цикл `for` выполняется до исчерпания элементов в заданном наборе. Возвращаясь к примеру с напитками, давайте рассмотрим, как обработать весь список с помощью цикла `for`, после чего мы сможем вернуться к проекту компании "Пузырь-Ок".

Далее указывается переменная, в которую поочередно записывается каждый элемент списка

По общепринятому соглашению имя списка стоит во множественном числе, а имя переменной цикла — в единственном

Цикл начинается с ключевого слово for

Затем идет ключевое слово in

Это список smoothies, который мы будем последовательно перебирать

Как всегда в Python, перед началом блока кода стоит двоеточие

```
for smoothie in smoothies:
    output = 'Мы подаем ' + smoothie
    print(output)
```

Блок кода выполняется один раз для каждого элемента списка

Переменную smoothie можно использовать в любом месте блока кода, где нужно получить текущий элемент списка

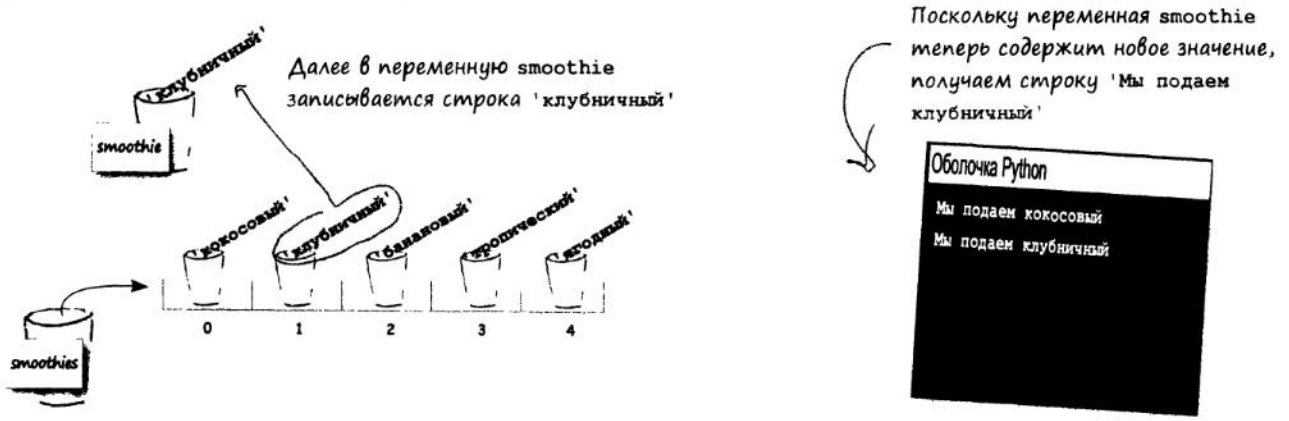
Принцип работы цикла for

Выполним приведенный выше код. На первой итерации цикла переменной `smoothie` присваивается значение первого элемента списка `smoothies`, после чего выполняются инструкции тела цикла.

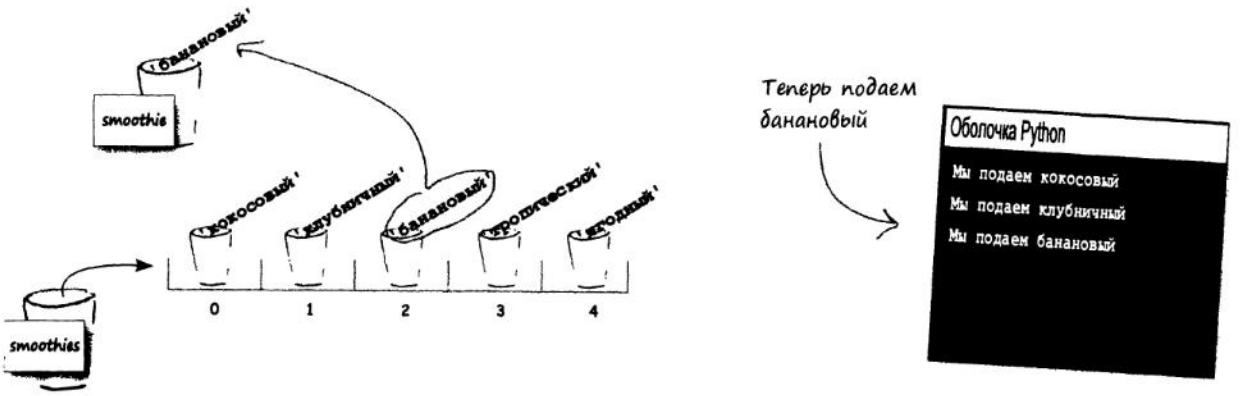
В переменную smoothie (не путать со списком smoothies!) записывается первый элемент списка

После этого выполняется блок кода инструкции for

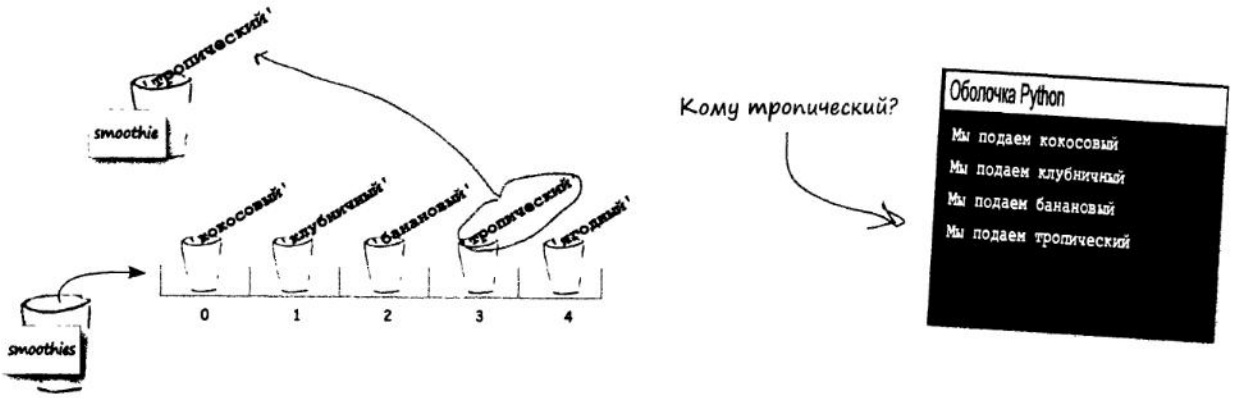
На следующей итерации цикла переменной `smoothie` присваивается значение `'клубничный'`, хранящееся во втором элементе списка `smoothies`. Далее снова выполняется тело цикла.



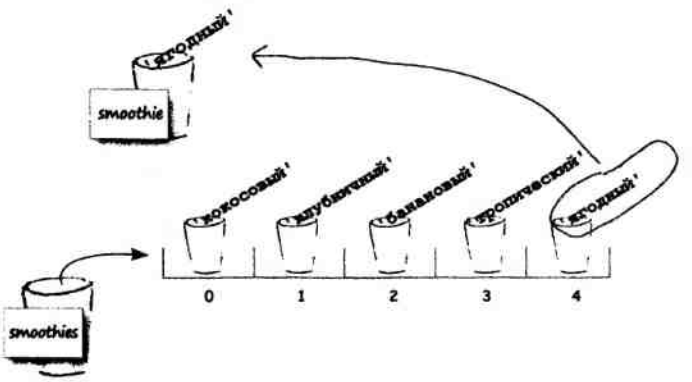
На третьей итерации цикла переменной `smoothie` присваивается строка `'банановый'`, хранящаяся в третьем элементе списка `smoothies`. Затем выполняются инструкции тела цикла.



К четвертой итерации вам должен быть понятен общий принцип — на этом этапе переменная `smoothie` принимает значение `'тропический'`, после чего выполняются инструкции цикла `for`.



В пятый раз, как несложно догадаться, выбирается строка `'ягодный'`, хранящаяся в пятом элементе списка `smoothies`, после чего в последний раз выполняются инструкции тела цикла.



Мы прошли все элементы списка smoothies

Оболочка Python

```
Мы подаем кокосовый
Мы подаем клубничный
Мы подаем банановый
Мы подаем тропический
Мы подаем ягодный
```

Юлия. В случае цикла `while` у нас был счетчик `i`, который служил номером раствора, а также индексом элемента списка.

Федор. Вот именно! А при работе с циклом `for` у нас есть только элементы списка. А где индексы?

Юлия. Хороший вопрос!

Игорь [кричит из другого угла комнаты]. Ребята, я узнал, что есть еще один способ использования цикла `for`! Способ, который вы применяете, хорош для перебора неиндексируемых элементов списка. В нашем случае индексы нужен диапазон индексов, чтобы можно было перебирать элементы списка.

Федор. А вот здесь поподробнее

Игорь. Проще показать...

Обработка диапазона чисел в цикле for

Цикл `for` прекрасно подходит для работы с другим типом последовательностей: *числовым диапазоном*. В Python имеется встроенная функция `range()`, позволяющая генерировать различные последовательности чисел, которые можно последовательно проходить в цикле `for`.

Вот как сгенерировать диапазон от 0 до 4:

```
range(5)
```

Создает последовательность 0, 1, 2, 3, 4

Диапазон начинается с 0 и образует последовательность из 5 чисел

Функцию `range()` можно включить в цикл `for`.

Диапазон образует последовательность 0, 1, 2, 3, 4

```
for i in range(5):
    print('Итерация', i)
```

В переменную i поочередно записывается каждый элемент последовательности

Оболочка Python

```
Итерация 0
Итерация 1
Итерация 2
Итерация 3
Итерация 4
>>>
```

Предположим, нам нужно просмотреть список напитков и вывести индекс каждого из них. Вот как это делается.

Создаем диапазон от 0 до значения длины списка smoothies

```
length = len(smoothies)
for i in range(length):
    print('Смузи #', i, smoothies[i])
```

На каждой итерации мы выводим строку 'Смузи #', индекс и название соответствующего напитка

Оболочка Python

```
Смузи # 0 кокосовый
Смузи # 1 клубничный
Смузи # 2 банановый
Смузи # 3 тропический
Смузи # 4 ягодный
>>>
```

Другие операции с диапазонами

Используя функцию `range()`, можно создавать числовые последовательности, начинающиеся не только с нуля, но и с любого другого числа. Приведем ряд примеров.

Задание начального и конечного значения

Начало диапазона

Конец диапазона (не включающий само число)

```
range(5, 10)
```

Создает последовательность, начинающуюся с 5, т.е. 5, 6, 7, 8, 9

Указание шага

Можно добавить величину шага, с которым будут пропускаться значения

```
range(3, 10, 2)
```

Создает последовательность от 3 до 10, но с шагом 2, т.е. 3, 5, 7, 9

Отрицательный шаг

Можно вести счет в обратном порядке, указав первый аргумент большим, чем второй, и задав отрицательный шаг

```
range(10, 0, -1)
```

Создает последовательность от 10 до 0 с шагом -1, т.е. 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

Как начать с отрицательного числа

Можно начинать отсчет с отрицательного числа

```
range(-10, 2)
```

Создает последовательность от -10 до 2, т.е. -10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1

Не бойтесь задавать вопросы

В: Создает ли функция `range(5)` список `[0, 1, 2, 3, 4]`?

О: Нет, не создает, но ход ваших мыслей понятен. Дело в том, что диапазон — это более эффективная структура данных, чем список. В определенном смысле они похожи, но диапазон нельзя использовать там, где ожидается список. Если требуется создать приведенный выше список на основе диапазона, задайте такую инструкцию:

```
list(range(5))
```

В: Счетчик цикла назывался `i`. Далеко не самое понятное имя, не так ли? Почему бы не назвать его, например, `index` или `smoothie_index`?

О: Хорошее замечание. Вы правы, имя `i` не самое понятное, но исторически сложилось так, что имена типа `i`, `j` и `k` применяются для обозначения счетчиков в циклах. Программисты почти автоматически следуют этой традиции, и более длинные имена будут сбивать их с толку. Так что советуем и вам использовать короткие имена счетчиков. Вы быстро к этому привыкнете.

Кто что делает?

Мы применили функцию `range()` для получения нескольких числовых последовательностей, но случайно перемешали все результаты. Не могли бы вы нам помочь разобраться во всем? Только будьте внимательны: мы не уверены, что каждому вызову соответствует ровно одна последовательность. Один результат мы определили самостоятельно.

`range(8)`	-3, -2, -1, 0
`range(0, 8)`	3, 6
`range(0, 8, 1)`	0, 1, 2, 3, 4, 5, 6, 7
`range(0, 8, 2)`	0, 1, 2, 3, 4, 5, 6
`range(0, 7)`	0, 2, 4, 6
`range(-3, 1)`	3, 4, 5, 6, 7
`range(3, 8)`	3, 5, 7
`range(3, 8, 3)`	1, 2, 3, 4
`range(3, 8, 2)`	

Составление отчета

Воспользуемся знаниями о диапазонах и цикле `for`, чтобы отказаться от применения цикла `while` при выводе списка результатов.

Вот наш список результатов

```
scores = [60, 50, 60, 58, 54, 54, 58, 50, 52, 54, 48, 69,
          34, 55, 51, 52, 44, 51, 69, 64, 66, 55, 52, 61,
          46, 31, 57, 52, 44, 18, 41, 53, 55, 61, 51, 44]
```

Сначала определяем длину списка

Цикл `while` можно удалить

```
length = len(scores)
~~while i < length:~~
for i in range(length):
    print('Пузырьковый раствор #' + str(i), '- результат:', scores[i])
```

Создаем диапазон на основе длины списка `scores`, а затем итерируем по этим значениям от 0 до длины списка минус 1

Создаем выходную строку. Это та же функция `print()`, что и в примере с циклом `while` — ничего не поменялось!

Тестирование отчета

Введите показанный выше код и сохраните его в файле *bubbles.py*, после чего запустите программу. Убедитесь в том, что полученный отчет соответствует нашим целям.

Как и просил директор

Приятно видеть все результаты в отчете, но нам по-прежнему сложно найти наибольший среди них. Нужно поработать над другими требованиями к отчету, чтобы было проще определять победителя

Оболочка Python

```
Пузырьковый раствор #0 - результат: 60
Пузырьковый раствор #1 - результат: 50
Пузырьковый раствор #2 - результат: 60
Пузырьковый раствор #3 - результат: 58
Пузырьковый раствор #4 - результат: 54
Пузырьковый раствор #5 - результат: 54
Пузырьковый раствор #6 - результат: 58
Пузырьковый раствор #7 - результат: 50
Пузырьковый раствор #8 - результат: 52
Пузырьковый раствор #9 - результат: 54
Пузырьковый раствор #10 - результат: 48
Пузырьковый раствор #11 - результат: 69
Пузырьковый раствор #12 - результат: 34
Пузырьковый раствор #13 - результат: 55
Пузырьковый раствор #14 - результат: 51
Пузырьковый раствор #15 - результат: 52
Пузырьковый раствор #16 - результат: 44
Пузырьковый раствор #17 - результат: 51
Пузырьковый раствор #18 - результат: 69
Пузырьковый раствор #19 - результат: 64
Пузырьковый раствор #20 - результат: 66
Пузырьковый раствор #21 - результат: 55
Пузырьковый раствор #22 - результат: 52
Пузырьковый раствор #23 - результат: 61
Пузырьковый раствор #24 - результат: 46
Пузырьковый раствор #25 - результат: 31
Пузырьковый раствор #26 - результат: 57
Пузырьковый раствор #27 - результат: 52
Пузырьковый раствор #28 - результат: 44
Пузырьковый раствор #29 - результат: 18
Пузырьковый раствор #30 - результат: 41
Пузырьковый раствор #31 - результат: 53
Пузырьковый раствор #32 - результат: 55
Пузырьковый раствор #33 - результат: 61
Пузырьковый раствор #34 - результат: 51
Пузырьковый раствор #35 - результат: 44
```

Возьмите карандаш

Вам предстоит выполнить еще одно короткое упражнение. Помните код на магнитиках, рассмотренный несколькими страницами ранее? Обновите его, заменив цикл `while` циклом `for`. В качестве подсказки сверьтесь с обновленным кодом проекта для компании "Пузырь-ОК".

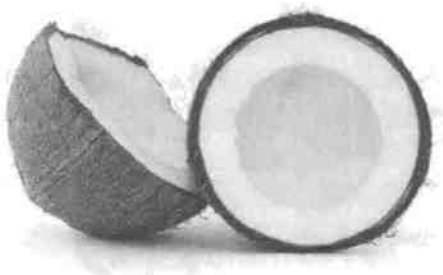
```
smoothies = ['кокосовый',
             'клубничный',
             'банановый',
             'тропический',
             'ягодный']
```

Вам не нужно листать страницы назад, потому что мы воспроизвели здесь решение

```
has_coconut = [True,
               False,
               False,
               True,
               False]
```

```
i = 0
while i < len(has_coconut)  :
    if has_coconut[i]  :
        print(smoothies[i],
              'содержит кокос')
    i = i + 1
```

Вставьте здесь свой код



Посиделки

Тема дискуссии: **какой цикл, WHILE или FOR, важнее?**

Цикл WHILE

Да вы шутите?! О чем мы спорим? Я универсальный цикл в Python. Мне не нужны ни списки, ни диапазоны, я могу работать с любыми типами условий. Кто-то вообще заметил, что меня первым рассматривали в книге?

И к тому же с чувством юмора у цикла `for` совсем плохо. Не удивительно для того, кто целыми днями только и делает, что перемалывает числовые последовательности.

Что-то с трудом в этом верится.

Между прочим, в книге было сказано, что циклы `for` и `while` делают примерно одно и то же. Так в чем проблема?

Цикл FOR

Попрошу не обобщать.

Остроумно. Особенно если учесть, что в 9 случаях из 10 программисты используют именно цикл `for`, а не `while`.

Не говоря уже о том, что обработка списка, содержащего фиксированное число элементов, с помощью цикла `while` считается плохой практикой программирования.

Ага, значит, о заявленном превосходстве речь уже не идет!

Понятно, почему...

При использовании цикла `while` приходится писать отдельные инструкции для объявления и приращения счетчика. Если в процессе редактирования кода случайно изменить или удалить одну из этих инструкций, последствия будут печальными. В случае цикла `for` такое в принципе невозможно — указанные операции реализуются в самой инструкции `for`.

Цикл WHILE

Как это все мило! В большинстве моих итераций вообще нет счетчиков. Вот, с чем я работаю:

```
while (input != ''):
```

Ну-ка, потянешь такое?

Я не только сильнее, но и симпатичнее.

Я, кстати, тоже умею работать с последовательностями.

А с чем еще?

Уверен, я с ними тоже справлюсь.

Да не такой уж ты и крутой! Смотри не надорвись, когда придется выполнять итерации, пока условие истинно.

Цикл FOR

И это все, в чем ты сильнее? Только с условиями и работаешь?

Ой, простите, не знал, что у нас конкурс красоты! Могу поспорить, что людям намного чаще приходится обрабатывать последовательности, чем писать циклы, основанные на условных конструкциях.

Мы уже обсуждали это. Да, ты умеешь, но это совсем не так симпатично, как кажется. И не забывай, я тоже достаточно универсален и могу работать не только со списками.

В Python много всяких последовательностей. Мы уже видели списки, диапазоны и строки, а есть еще файлы и более сложные структуры данных, не рассматривавшиеся в книге.

Возможно, но это будет вовсе не так симпатично. Признай: когда дело доходит до рутинных итераций, я просто-таки рожден для этого.

И тебе не захворать, когда придется обрабатывать числовые последовательности.

Офисный диалог продолжается...

Юлия. Первое, что нужно сделать, — подсчитать общее число пузырьковых тестов. Это легко — просто определяем длину списка результатов.

Игорь. Да, но еще нужно определить наибольший результат и узнать, в каких растворах он достигнут.

Юлия. Последнее будет самым сложным, поэтому начнем с нахождения наибольшего результата.

Игорь. Да, так проще.

Юлия. Думаю, для этого нам понадобится переменная, которая будет хранить текущий наибольший результат, пока мы будем проходить по списку в цикле. Вот как выглядит соответствующий псевдокод.

```
DECLARE переменная high_score и присвоить ей 0
FOR i in range(length)
    PRINT i и результат раствора scores[i]
    IF scores[i] > high_score:
        high_score = scores[i];
PRINT high_score
```

Добавляем переменную для хранения наибольшего результата

На каждой итерации цикла проверяем, превышен ли наибольший результат, и если да, то сохраняем новое значение

Если наибольший результат превышен, то сохраняем значение в переменной high_score

По завершении цикла выводим наибольший результат

Игорь. Прекрасно! Получилось всего несколько строк кода.

Юлия. На каждой итерации цикла мы проверяем, превышает ли текущий результат переменную `high_score`, и если да, то заменяем ее значение. По завершении цикла просто отображаем на экране значение `high_score`.

Возьмите карандаш

Попробуйте реализовать псевдокод, приведенный на предыдущей странице, заполнив пустые участки в коде. Вам необходимо определить наибольший из полученных результатов тестов. Сохраните код в файле *bubbles.py* и запустите программу. Проверьте, какие значения будут выведены вместо подчеркиваний в окне оболочки Python. Как всегда, сверьтесь с ответом в конце главы, прежде чем двигаться дальше.

```
scores = [60, 50, 60, 58, 54, 54,
          58, 50, 52, 54, 48, 69,
          34, 55, 51, 52, 44, 51,
          69, 64, 66, 55, 52, 61,
          46, 31, 57, 52, 44, 18,
          41, 53, 55, 61, 51, 44]

high_score = ______

length = len(scores)
for i in range(length):
    print('Пузырьковый раствор #' + str(i), '- результат:', scores[i])
    if _____________ > high_score:
        _____________ = scores[i]

print('Пузырьковых тестов:', _____________)
print('Наибольший результат:', _____________)
```

Заполните прочерки, чтобы завершить код...

...а затем заполните прочерки здесь, показав, каким будет результат на консоли

Оболочка Python

```
Пузырьковый раствор #0 - результат: 60
Пузырьковый раствор #1 - результат: 50
Пузырьковый раствор #2 - результат: 60
...
Пузырьковый раствор #34 - результат: 51
Пузырьковый раствор #35 - результат: 44
Пузырьковых тестов: _____
Наибольший результат: _____
```

Несколько? А какую структуру мы используем для хранения сразу нескольких значений? Список, конечно же! Значит, мы можем пройти по списку результатов, найти значения, соответствующие максимальному результату, и добавить их в новый список, который впоследствии отобразим в отчете? Именно так. Но сначала нужно узнать, как создать пустой список и добавить в него новые элементы.

Вот что нам еще осталось сделать

Пузырь-ОК

Мне очень нужен этот отчет, чтобы принимать быстрые решения о том, какой раствор выпускать. Сможете написать?

— генеральный директор "Пузырь-ОК"

```
Пузырьковый раствор #0 - результат: 60
Пузырьковый раствор #1 - результат: 50
Пузырьковый раствор #2 - результат: 60

Пузырьковых тестов: 36
Наибольший результат: 69
Растворы с наибольшим результатом: [11, 18]
```

Остальные результаты...

Построение списка с нуля

Прежде чем мы сможем завершить отчет, давайте разберемся, как создать новый список и заполнить его элементами. Мы уже знаем, как объявляется список с фиксированными значениями:

```
menu = ['Пицца', 'Паста', 'Суп', 'Салат']
```

Однако указывать элементы при создании списка совсем не обязательно; в таком случае создается пустой список.

```
menu = []
```

Новый пустой список без элементов и с нулевой длиной

Для добавления элементов в список предназначена функция `append()`.

```
menu = []
menu.append('Бургер')
menu.append('Суши')

print(menu)
```

Создаем новый список

Добавляем в него строку 'Бургер'

А затем добавляем второй элемент — строку 'Суши'

```
Оболочка Python
['Бургер', 'Суши']
>>>
```

В Python пустой список можно также создать с помощью функции `list()`, чем мы обязательно воспользуемся позже. А пока просто примите к сведению такую возможность

Возьмите карандаш

Что, по-вашему, делает приведенный ниже код? Обязательно введите его в оболочке Python, чтобы узнать это.

```
mystery = ['секрет'] * 5
```

Умножение числа на список? И что же мы получим в результате?

```
mystery = 'секрет' * 5
```

А чем отличается такой вариант?

Другие операции со списками

Со списками можно много чего делать: добавлять элементы, удалять элементы, объединять списки, находить в них элементы и т.п. Ниже приведено несколько примеров таких операций.

Удаление элемента списка

Хотите сократить список? В Python для этого имеется встроенная функция `del()`. Вот как она работает:

Удаляем элемент с индексом 0

```
del menu[0]
print(menu)
```

Элемент с индексом 0 удален, и остался только один элемент, 'Суши', который теперь имеет индекс 0

Оболочка Python

```
['Суши']
>>>
```

После удаления элемента из списка все элементы, расположенные после него, смещаются на одну позицию вниз. Например, если вы удалили элемент с индексом 2, то элемент, ранее имевший индекс 3, занимает его место. Элемент с индексом 4 становится элементом с индексом 3 и т.д.

Включение списка в другой список

Предположим, вы хотите включить в имеющийся у вас список элементы другого списка. В Python это несложно сделать.

Расширяем наше меню элементами из другого списка

Наш текущий список, menu, дополнен элементами из предоставленного списка

```
menu.extend(['Барбекю', 'Тако'])
print(menu)
```

Оболочка Python

```
['Суши', 'Барбекю', 'Тако']
>>>
```

Сила мысли

Чем отличаются функции `append()` и `extend()` применительно к спискам?

Есть и другой способ объединения списков: с помощью оператора +, как показано ниже.

```
menu = menu + ['Барбекю', 'Тако']
```

Если выполнить этот код вместо функции extend(), получим тот же результат

Оболочка Python

```
['Суши', 'Барбекю', 'Тако']
>>>
```

Примечание: функция extend() расширяет существующий список. При использовании оператора + получаем совершенно новый список, включающий элементы обоих списков

Вставка элементов в список

Предположим, вы хотите вставить новый элемент в указанное место списка. Для этого предназначена функция `insert()`.

Это индекс, куда мы хотим вставить элемент

А вот сам элемент

```
menu.insert(1, 'Рагу')
print(menu)
```

Оболочка Python

```
['Суши', 'Рагу', 'Барбекю', 'Тако']
>>>
```

Функция insert() вставляет новый элемент по указанному индексу, в данном случае 1

В последующих главах вы познакомитесь с другими операциями, выполняемыми над списками, а пока что вам будет достаточно рассмотренных операций.

Не бойтесь задавать вопросы

В: Что произойдет, если попытаться вставить в список элемент с несуществующим индексом?

О: Если вы попробуете вставить в список элемент, индекс которого больше, чем длина списка, то он будет просто добавлен в конец списка.

В: Что означает синтаксис *мой_список*.append(*значение*)? Он напоминает синтаксис команды `random.randint(0, 2)` из предыдущей главы.

О: Да, их сходство не случайно. Вы поймете это в последующих главах, где вас ждет знакомство с объектами и применяемыми к ним функциями (там вы узнаете и новую терминологию). Мы не будем слишком забегать наперед. Вам достаточно знать, что структуры данных, в частности списки, могут поддерживать собственные операции, такие как добавление элементов. Функция `append()` — пример такой операции, реализуемой списком. Сейчас примите этот синтаксис как данность — время объектов еще не настало.

В: А почему тогда при работе со списками применяются команды `menu.append()` и `menu.insert()`, но `del menu[0]`? Разве Python не поддерживает синтаксис `menu.delete(0)` или подобный ему? Странновато для столь последовательного языка, как Python.

О: Хороший вопрос! Оказывается, создатели Python посчитали, что привычные операции, такие как `len` и `del`, заслуживают особой трактовки. Кроме того, по их мнению, синтаксис `len(menu)` лучше воспринимается, чем `menu.length()`. Дебаты долго не утихали, и в итоге имеем то, что имеем. Опять-таки, все это станет для вас более понятным, когда мы познакомимся с объектами.

Юлия. Точно! Мы начнем с пустого списка, в который будем добавлять номера растворов с наибольшим результатом по мере их нахождения в исходном списке.

Федор. Отлично, можем начинать!

Юлия. Погоди-ка... Кажется, нам понадобится еще один цикл.

Федор. Разве? Наверняка есть способ сделать это в существующем цикле.

Юлия. Боюсь, что нет. Дело в том, что мы должны *заранее* знать максимальный результат, прежде чем начинать искать соответствующие растворы. Так что нам нужны два цикла: один — для поиска максимального значения в списке результатов (он у нас уже есть), а второй — для поиска всех растворов с таким результатом.

Федор. Ты права. Во втором цикле мы будет сравнивать каждый из элементов исходного списка с максимальным результатом. В случае совпадения индекс найденного результата будет заноситься в отдельный, заранее созданный пустой список.

Юлия. Именно так! Теперь можно начинать.

Возьмите карандаш

Помогите написать цикл для нахождения *всех* результатов, равных максимальному. На данный момент у нас есть приведенный ниже код. Заполните оставшиеся прочерки и только после этого сверьтесь с ответом в конце главы.

```
scores = [60, 50, 60, 58, 54, 54,
          58, 50, 52, 54, 48, 69,
          34, 55, 51, 52, 44, 51,
          69, 64, 66, 55, 52, 61,
          46, 31, 57, 52, 44, 18,
          41, 53, 55, 61, 51, 44]

high_score = 0

length = len(scores)
for i in range(length):
    print('Пузырьковый раствор #' + str(i), ' - результат:', scores[i])
    if scores[i] > high_score:
        high_score = scores[i]

print('Пузырьковых тестов:',  length)
print('Наибольший результат:',  high_score)
best_solutions = []
 ____________________
    ____________________
    ____________________
```

Это наш код на текущий момент

Это новый список, в котором будут храниться номера растворов с наибольшим результатом

Запишите здесь свой код, используя больше строк, если понадобится

Помните о том, что переменная high_score содержит наибольший результат

Тест-драйв готового отчета

Добавьте в файл *bubbles.py* код нахождения растворов с максимальным результатом и проверьте работу программы. Текст программы приведен ниже.

```
scores = [60, 50, 60, 58, 54, 54,
          58, 50, 52, 54, 48, 69,
          34, 55, 51, 52, 44, 51,
          69, 64, 66, 55, 52, 61,
          46, 31, 57, 52, 44, 18,
          41, 53, 55, 61, 51, 44]

high_score = 0

length = len(scores)
for i in range(length):
    print('Пузырьковый раствор #' + str(i), '- результат:', scores[i])
    if scores[i] > high_score:
        high_score = scores[i]

print('Пузырьковых тестов:', length)
print('Наибольший результат:', high_score)

best_solutions = []
for i in range(length):
    if high_score == scores[i]:
        best_solutions.append(i)

print('Растворы с наибольшим результатом:', best_solutions)
```

Если этот код кажется вам незнакомым, значит, вы не изучили решение предыдущего упражнения "Возьмите карандаш". Сделайте это сейчас

И победителями становятся...

Растворы #11 и #18 показали наилучший результат: 69 пузырьков. Они и становятся победителями!

Оболочка Python

```
Пузырьковый раствор #0 - результат: 60
Пузырьковый раствор #1 - результат: 50
 ...
Пузырьковый раствор #34 - результат: 51
Пузырьковых тестов: 36
Наибольший результат: 69
Растворы с наибольшим результатом: [11, 18]
```

Прекрасная работа! Но есть еще кое-что... Не могли бы вы определить наиболее экономически выгодный раствор? Имея такие данные, мы сможем захватить весь рынок пузырьковых растворов! Вот список со стоимостями растворов.

В этом списке приведены стоимости растворов. Растворы в нем указаны в той же последовательности, что и в списке результатов

```
costs = [.25, .27, .25, .25, .25, .25,
         .33, .31, .25, .29, .27, .22,
         .31, .25, .25, .33, .21, .25,
         .25, .25, .28, .25, .24, .22,
         .20, .25, .30, .25, .24, .25,
         .25, .25, .27, .25, .26, .29]
```

Ну и в чем тут загвоздка? Задача состоит в выборе наиболее дешевого раствора среди тех, что показали наилучший результат в тестах. Теперь в нашем распоряжении имеются сразу два списка: `costs` и `scores`. Обратите внимание на то, что элементы с одинаковыми индексами содержат данные для одних и тех же растворов. В частности, элементу с индексом `0` списка `scores` соответствует элемент с таким же индексом в списке `cost`, определяющий стоимость этого раствора (`.25`). Стоимость второго раствора (`.27`) хранится в элементе `costs[1]` и т.д. Такие списки иногда называют параллельными.

Списки scores и costs параллельны, поскольку для каждого результата есть соответствующая стоимость с тем же индексом

```
costs = [.25, .27, .25, .25, .25, .25, .33, .31, .25, .29, .27, .22, ..., .29]

scores = [60,  50,  60,  58,  54,  54,  58,  50,  52,  54,  48,  69,  ..., 44]
```

Стоимость с индексом 0 — это стоимость раствора с индексом 0

То же самое справедливо для всех остальных элементов обоих списков

Юлия. Нам уже известны растворы с наибольшим результатом.

Федор. Да, но как нам это поможет? Как нам использовать данные из обоих списков?

Юлия. Любой из нас легко напишет простой цикл `for` для просмотра списка `scores` и определения номеров растворов с наибольшим результатом.

Федор. Я точно смогу. Ну и что?

Юлия. Всякий раз, когда находится раствор с наибольшим результатом, нужно определить его стоимость и проверить, не является ли она наименьшей.

Федор. Ясно, нам потребуется переменная, в которой будет храниться индекс раствора с наименьшей стоимостью. Пара пустяков!

Юлия. Согласна. После просмотра всего списка эта переменная будет содержать номер раствора, который не только показал наилучший результат, но и имеет самую низкую стоимость.

Федор. А что если у двух растворов окажется одинаковая стоимость?

Юлия. Нужно решить, как быть в этой ситуации. Можно объявить победителем первый из найденных растворов. Есть и более сложные решения, но пока остановимся на этом, если только генеральный директор не потребует чего-то другого.

Федор. Придется составить псевдокод, прежде чем приступать к программированию.

Юлия. Однозначно. Когда приходится обрабатывать сразу несколько списков, все быстро усложняется. Лучше все заранее распланировать, чтобы потом не попасть впросак.

Федор. Это точно. Пойдем привычным путем.

Уверен, это правильный псевдокод. Изучите его, и затем мы перейдем к написанию программного кода.

Создаем переменную, которая будет хранить стоимость самого выгодного раствора. Мы задаем ее большей любого элемента списка `costs` *и делаем ее числом с плавающей точкой, чтобы она соответствовала типу элементов списка*

Создаем переменную для хранения индекса самого выгодного раствора

DECLARE ***переменная*** cost *и присвоить ей 100.0*

DECLARE ***переменная*** most_effective

Просматриваем все растворы, и если раствор имеет наибольший результат...

FOR i in range(length):

IF *результат раствора scores[i] равен high_score* **AND** *стоимость раствора costs[i] меньше, чем cost:*

...а также меньшую стоимость, чем предыдущие растворы...

SET most_effective *равно i*

SET cost *равно стоимости раствора*

...то записываем индекс и стоимость текущего раствора

В конце цикла переменная `most_effective` *будет содержать индекс раствора с наибольшим результатом и наименьшей стоимостью, а переменная* `cost` *будет содержать стоимость этого раствора. Если таких растворов несколько, то программа всегда будет выбирать первый из них*

Сила мысли

Как и порекомендовала Юлия, при нахождении двух растворов с наибольшим результатом и одинаково низкой стоимостью победа присуждается первому из найденных. Но почему так происходит? Какая часть кода отвечает за подобное поведение программы? И как быть, если победу нужно присудить последнему из растворов? Что нужно поменять?

Ответ: в показателе сравнение стоимостей растворов реализуется в виде операции "меньше, чем". Соответственно, раствор с равной стоимостью не пройдет эту проверку. Если нужно, чтобы победителем становился последний из найденных растворов, используйте операцию "меньше или равно" (<=).

Тестирование самого выгодного раствора

Наконец-то мы закодировали весь отчет, как того хотел директор компании "Пузырь-ОК". Внимательно изучите код программы и сравните его с приведенным выше псевдокодом, после чего добавьте его в файл *bubbles.py* и запустите программу для проверки. Узнайте номер раствора-победителя, после чего переверните страницу и сравните его с нашим результатом.

```
scores = [60, 50, 60, 58, 54, 54,
          58, 50, 52, 54, 48, 69,
          34, 55, 51, 52, 44, 51,
          69, 64, 66, 55, 52, 61,
          46, 31, 57, 52, 44, 18,
          41, 53, 55, 61, 51, 44]

costs = [.25, .27, .25, .25, .25, .25,
         .33, .31, .25, .29, .27, .22,
         .31, .25, .25, .33, .21, .25,
         .25, .25, .28, .25, .24, .22,
         .20, .25, .30, .25, .24, .25,
         .25, .25, .27, .25, .26, .29]
```

Не забудьте о новом списке стоимостей

```
high_score = 0

length = len(scores)
for i in range(length):
    print('Пузырьковый раствор #' + str(i), '- результат:', scores[i])
    if scores[i] > high_score:
        high_score = scores[i]

print('Пузырьковых тестов:', length)
print('Наибольший результат:', high_score)

best_solutions = []
for i in range(length):
    if high_score == scores[i]:
        best_solutions.append(i)

print('Растворы с наибольшим результатом:', best_solutions)

cost = 100.0
most_effective = 0
for i in range(length):
    if scores[i] == high_score and costs[i] < cost:
        most_effective = i
        cost = costs[i]
print('Раствор', most_effective,
      'самый выгодный. Его цена -', costs[most_effective])
```

Мы перевели псевдокод непосредственно в код Python

Мы также добавили инструкцию вывода, чтобы сообщить о том, какой раствор самый выгодный

ПОБЕДИТЕЛЬ: РАСТВОР #11

Последний фрагмент программы позволил нам определить истинного победителя, т.е. раствор, который выдает наибольшее количество пузырьков и при этом стоит меньше всего. Поздравляем! Вам удалось обработать большой массив данных и предоставить компании "Пузырь-ОК" ценные сведения, которые помогут ей с продвижением своей продукции на рынке.

Наверняка вам, как и нам, не терпится узнать, что же собой представляет раствор под номером 11. К счастью, директор компании "Пузырь-ОК" решил отблагодарить вас за безвозмездный труд и поделиться коммерческим рецептом.

Рецепт раствора #11 приведен ниже. Изучите его, а затем попробуйте приготовить раствор и выдуть несколько пузырьков, после чего можете переходить к следующей главе. Только не забудьте прочитать резюме главы и решить кроссворд.

Пузырьковый раствор #11

2/3 чашки средства для мытья посуды

4 литра воды

2-3 столовые ложки глицерина (продается в аптеках или магазинах бытовой химии)

ИНСТРУКЦИИ: смешайте ингредиенты в большой банке и наслаждайтесь!

ПОПЫТАЙТЕСЬ повторить это ДОМА!

Хочу спросить: у нас уже есть список best_solutions, в котором представлены растворы с наибольшим результатом, так зачем нам заново просматривать КАЖДЫЙ результат?

Вы правы: в этом нет необходимости.

Самый выгодный раствор можно выбрать из списка best_solutions, поскольку он содержит все нужные для этого элементы. Мы пошли другим путем только потому, что для первой попытки так было проще.

Кто-то спросит: а в чем разница? Кто нас за это обругает? Работает ведь! Все дело в эффективности кода. Какой объем вычислений выполняет программа? Для такого маленького списка, как наш, разницы действительно нет, но если бы мы имели дело с *огромным* списком, то стремились бы избежать повторного итерирования по нему при наличии более эффективного пути.

Все, что нам нужно сделать для определения наиболее выгодного раствора, — поискать его в списке best_solutions. Код получается чуть более сложным, но не намного.

Мы переписали код для вычисления самого выгодного раствора

На этот раз мы просматриваем список наилучших растворов, а не список результатов

Каждый элемент списка best_solutions служит индексом в списке costs

Таким образом, значения из списка best_solutions используются как индексы

```
cost = 100.0
most_effective = 0

for i in range(len(best_solutions)):
    index = best_solutions[i]
    if cost > costs[index]:
        most_effective = index
        cost = costs[index]

print('Раствор', most_effective,
      'самый выгодный. Его цена -',
      costs[most_effective])
```

Мы проверяем стоимость каждого раствора в списке best_solutions и ищем наименьшую

В конце выводим результаты

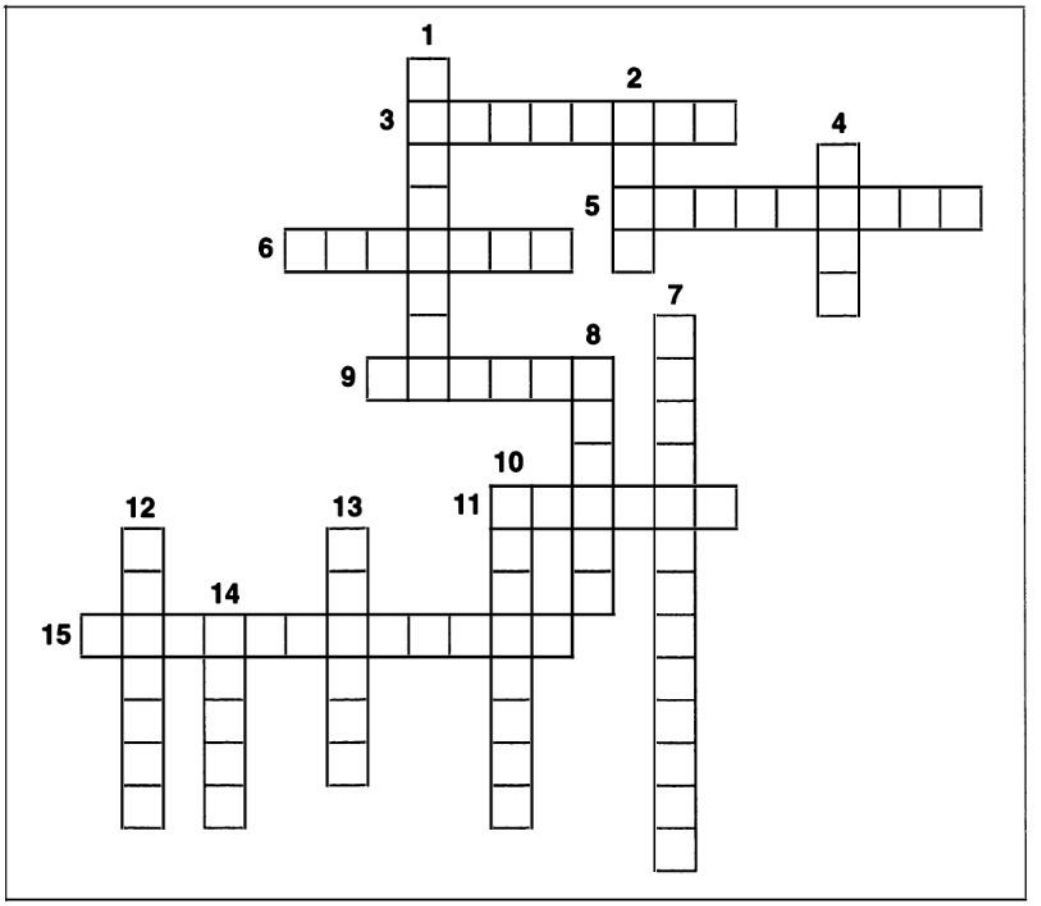
Сравните этот код с предыдущим вариантом решения. Заметили разницу? А теперь представьте, как оба варианта кода будут выполняться компьютером. Не правда ли, итоговое решение требует намного меньше времени на выполнение? Разница настолько большая, что стоит потраченного на ее изучение времени.

САМОЕ ГЛАВНОЕ

- Списки — это *структуры* для хранения упорядоченных данных.
- Список состоит из набора элементов, снабженных *индексами*.
- Индексация элементов списка начинается с нуля. Первый элемент имеет индекс 0.
- Количество элементов в списке (его длину) можно подсчитать с помощью функции `len()`.
- К элементу списка можно обратиться по индексу. К примеру, выражение `my_list[1]` возвращает второй элемент списка `my_list`.
- Допускается использовать отрицательные индексы, обозначающие нумерацию с конца списка.
- Попытка обратиться к элементу, индекс которого больше длины списка, приводит к ошибке выполнения.
- Значение существующего элемента списка можно изменить с помощью операции присваивания.
- Попытка присвоить значение несуществующему элементу списка приводит к ошибке выполнения "out of bounds".
- В списках можно хранить произвольные значения.
- Список может содержать значения сразу нескольких типов.
- Списки, хранящие данные разных типов, называются неоднородными.
- Пустой список создается инструкцией `my_list = []`.
- Для добавления новых элементов в список предназначена функция `append()`.
- Список можно дополнить элементами другого списка с помощью функции `extend()`.
- Можно создать новый список путем объединения двух существующих списков с помощью оператора `+`.
- Функция `insert()` позволяет добавить в указанную позицию списка новый элемент.
- Для просмотра последовательностей типа списков обычно применяется цикл `for`.
- Цикл `while` применяют, когда число итераций не известно заранее, а ограничивается определенным условием. Цикл `for`, наоборот, лучше всего использовать при фиксированном количестве итераций.
- Функция `range()` создает диапазон целых чисел.
- Для последовательного прохода по диапазону применяется цикл `for`.
- Функция `str()` преобразует число в строку.

Кроссворд

Чтобы лучше усвоить тему списков, попробуйте решить кроссворд.



По горизонтали

3. Отдельный шаг цикла
5. Один из напитков, который мы тестировали в этой главе
6. Добавление элемента в список
9. Порядковый номер элемента списка
11. Список без элементов
15. Список, содержащий элементы разных типов

По вертикали

1. Последовательный числовой интервал
2. Управляющая конструкция, предназначенная для многократного повторения блока инструкций
4. Число, с которого начинается индексация списка
7. Добавление списка в другой список
8. Набор элементов, хранящихся под одним именем
10. Подсчитываются при тестировании растворов
12. Составная часть списка
13. Текст, хранящийся в переменной
14. Число элементов списка

Возьмите карандаш
Решение

Давайте попрактикуемся в обработке элементов списка. Внимательно изучите приведенный ниже код и постарайтесь определить результат его выполнения интерпретатором Python.

Попробуем отследить изменение списка **eighties** *и других переменных программы. Список* **newwave** *никогда не меняется.*

```
eighties = ['', 'duran duran', 'B-52s', 'a-ha']
newwave = ['flock of seagulls', 'secret service']
```

	eighties	remember	band
`remember = eighties[1]`	['', 'duran duran', 'B-52s', 'a-ha']	'duran duran'	
`eighties[1] = 'culture club'`	['', 'culture club', 'B-52s', 'a-ha']	'duran duran'	
`band = newwave[0]`	['', 'culture club', 'B-52s', 'a-ha']	'duran duran'	'flock of seagulls'
`eighties[3] = band`	['', 'culture club', 'B-52s', 'flock of seagulls']	'duran duran'	'flock of seagulls'
`eighties[0] = eighties[2]`	['B-52s', 'culture club', 'B-52s', 'flock of seagulls']	'duran duran'	'flock of seagulls'
`eighties[2] = remember`	['B-52s', 'culture club', 'duran duran', 'flock of seagulls']	'duran duran'	'flock of seagulls'

```
print(eighties)
```

Оболочка Python

```
['B-52s', 'culture club', 'duran duran', 'flock of seagulls']
>>>
```

Итоговый результат

Супержелезяка

Супержелезяка — это хитроумная штуковина, которая стучит, гремит и даже гудит. Но что она делает? Мы и сами в замешательстве. Программисты утверждают, будто знают, как она работает. А вы сможете разобраться в ее коде?

Что же делает супержелезяка? Она берет последовательность символов и создает из нее палиндром. Как известно, палиндром — это слово или фраза, которые читаются одинаково в прямом и обратном направлении, как, например, "топот". Таким образом, если передать программе последовательность символов `'t'`, `'a'`, `'c'`, `'o'`, то она выведет `'tacocat'`. Результат не слишком впечатляющий, но иногда удается получить по-настоящему поразительный палиндром. К примеру, набор символов `'a'`, `'m'`, `'a'`, `'n'`, `'a'`, `'p'`, `'l'`, `'a'`, `'n'`, `'a'`, `'c'` превращается в `'manaplanacanalpanama'` (или "a man a plan a canal panama").

Самое важное в программе — то, как она берет половину палиндрома и "замыкает" его. Проанализируем код.

```
characters = ['t', 'a', 'c', 'o']

output = ''
length = len(characters)
i = 0
while (i < length):
    output = output + characters[i]
    i = i + 1

length = length * -1
i = -2

while (i >= length):
    output = output + characters[i]
    i = i - 1

print(output)
```

Вначале переменная output содержит пустую строку

Определяем длину списка символов

Задаем переменную i равной 0

Начинаем с позиции 0 и проходим каждый элемент списка символов, добавляя его в строку output

Теперь идем в обратном направлении, инвертируя значение length (например, значение 8 превратится в -8)

Устанавливаем переменную i равной -2 (сейчас увидите зачем)

Почему в обратном порядке? Потому что индексы теперь отрицательные

Мы проходим список символов в обратном порядке, пропуская самый последний элемент, чтобы он не повторялся посреди строки

Выводим результат

Изучайте пример, пока не поймете, как он работает! Пройдите каждую итерацию цикла и запишите значения переменных, которые меняются

Другие примеры палиндромов будут приведены в главе 8

Возьмите карандаш
Решение

Приведенные ниже названия напитков добавлялись в список в порядке создания. Завершите код программы, определяющей название самого последнего напитка.

```
smoothies = ['кокосовый', 'клубничный', 'банановый', 'ананасовый', 'ягодный']
most_recent = ____-1____
recent = smoothies[most_recent]
```

В Python для доступа к последнему элементу списка можно использовать отрицательный индекс -1

```
smoothies = ['кокосовый', 'клубничный', 'банановый', 'ананасовый', 'ягодный']
most_recent = len(smoothies) - 1
recent = smoothies[most_recent]
```

Или же можно вычесть 1 из значения длины списка для получения индекса последнего элемента

Код можно дополнительно улучшить

Когда задача разбивается на пошаговые действия, это позволяет сделать код более наглядным. Однако в случае простых операций чем короче код, тем он понятнее. Давайте попробуем сделать приведенный выше код более понятным.

```
smoothies = ['кокосовый', 'клубничный', 'банановый', 'ананасовый', 'ягодный']
~~most_recent = -1~~
recent = smoothies[-1]
```

Можно избавиться от переменной `most_recent` и напрямую использовать индекс -1

Аналогичным образом можно убрать лишнюю инструкцию во втором примере

```
smoothies = ['кокосовый', 'клубничный', 'банановый', 'ананасовый', 'ягодный']
~~most_recent = len(smoothies) - 1~~
recent = smoothies[len(smoothies)-1]
```

Мы избавились от промежуточной переменной `most_recent` и переместили выражение `len(smoothies)-1` в инструкцию определения текущего элемента списка. Возможно, это не делает код более читабельным, но для опытного программиста так понятнее. Следует подчеркнуть, что написание понятного кода — это больше искусство, чем наука. Выбирайте стиль, который проще и понятнее для вас, только помните, что со временем восприятие кода может измениться.

Код на магнитиках: решение

Пора выполнить небольшое упражнение. Мы написали код, чтобы узнать, в каких напитках содержится кокос. Код был записан с помощью набора магнитиков на дверце холодильника, но они упали на пол. Постарайтесь все восстановить. Только будьте внимательны: некоторые магнитики лишние. Сверьтесь с ответом, приведенным в конце главы.

```
i = i + 2
```

```
:
```

```
while i > len(has_coconut)
```

```
smoothies = ['кокосовый',
             'клубничный',
             'банановый',
             'тропический',
             'ягодный']
```

```
has_coconut = [True,
               False,
               False,
               True,
               False]
```

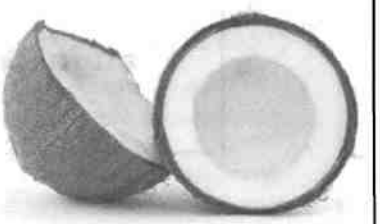
```
i = 0
```

```
while i < len(has_coconut)   :
    if has_coconut[i]   :
        print(smoothies[i],'содержит кокос')
    i = i + 1
```

Расставьте здесь магнитики в нужном порядке

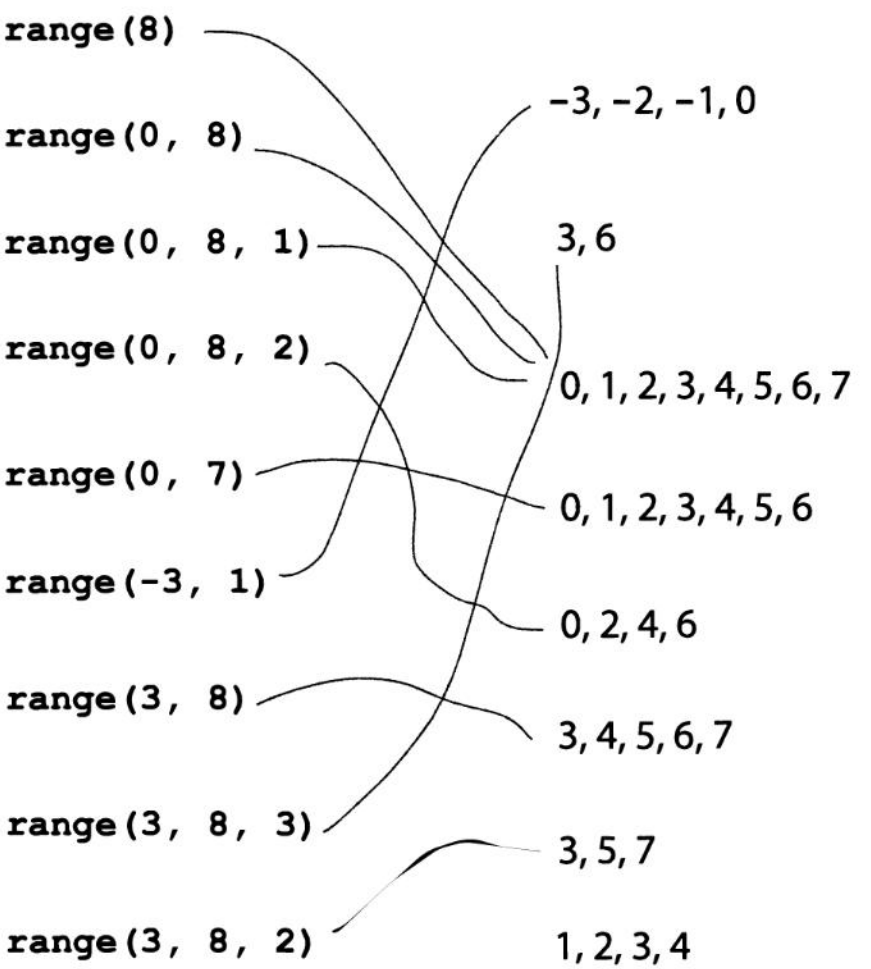
Вот какой результат мы ожидаем

```
Оболочка Python
кокосовый содержит кокос
тропический содержит кокос
>>>
```



Кто что делает? Решение

Мы применили функцию `range()` для получения нескольких числовых последовательностей, но случайно перемешали все результаты. Не могли бы вы нам помочь разобраться во всем? Только будьте внимательны: мы не уверены, что каждому вызову соответствует ровно одна последовательность. Один результат мы определили самостоятельно.



Возьмите карандаш
Решение

Вам предстоит выполнить еще одно короткое упражнение. Помните код на магнитиках, рассмотренный несколькими страницами ранее? Обновите его, заменив цикл `while` циклом `for`. В качестве подсказки сверьтесь с обновленным кодом проекта для компании "Пузырь-ОК".

```
smoothies = ['кокосовый',
             'клубничный',
             'банановый',
             'тропический',
             'ягодный']
```

```
has_coconut = [True,
               False,
               False,
               True,
               False]
```

```
i = 0
while i < len(has_coconut) :
    if has_coconut[i] :
        print(smoothies[i],
              'содержит кокос')
    i = i + 1
```

Вставьте здесь свой код

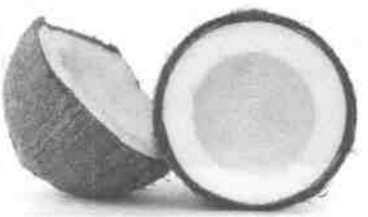
```
smoothies = ['кокосовый',
             'клубничный',
             'банановый',
             'тропический',
             'ягодный']

has_coconut = [True,
               False,
               False,
               True,
               False]

length = len(has_coconut)

for i in range(length):
    if has_coconut[i]:
        print(smoothies[i], 'содержит кокос')
```

Определяем длину списка

Итерируем от 0 до значения длины списка минус 1

Проверяем элемент списка `has_coconut` *с индексом* `i`*, и если он содержит кокос, то выводим его название из списка* `smoothies`

Можно было бы записать `if has_coconut[i] == True`*, но поскольку* `has_coconut[i]` *содержит булево значение, в этом нет необходимости*



Возьмите карандаш
Решение

Попробуйте реализовать псевдокод, приведенный на предыдущей странице, заполнив пустые участки в коде. Вам необходимо определить наибольший из полученных результатов тестов. Сохраните код в файле *bubbles.py* и запустите программу. Проверьте, какие значения будут выведены вместо подчеркиваний в окне оболочки Python.

```
scores = [60, 50, 60, 58, 54, 54,
          58, 50, 52, 54, 48, 69,
          34, 55, 51, 52, 44, 51,
          69, 64, 66, 55, 52, 61,
          46, 31, 57, 52, 44, 18,
          41, 53, 55, 61, 51, 44]

high_score = __0__

length = len(scores)
for i in range(length):
    print('Пузырьковый раствор #' + str(i), '- результат:', scores[i])
    if __scores[i]__ > high_score:
        __high_score__ = scores[i]

print('Пузырьковых тестов:', __length__)
print('Наибольший результат:', __high_score__)
```

Заполните прочерки, чтобы завершить код...

Вот что мы получили

Оболочка Python

```
Пузырьковый раствор #0 - результат: 60
Пузырьковый раствор #1 - результат: 50
Пузырьковый раствор #2 - результат: 60
 ...
Пузырьковый раствор #34 - результат: 51
Пузырьковый раствор #35 - результат: 44
Пузырьковых тестов: 36
Наибольший результат: 69
```

Возьмите карандаш
Решение

Что, по-вашему, делает приведенный ниже код? Обязательно введите его в оболочке Python, чтобы узнать это.

```
mystery = ['секрет'] * 5
```

Данная инструкция создает список, в котором элемент 'секрет' повторяется 5 раз: ['секрет', 'секрет', 'секрет', 'секрет', 'секрет']. Это удобная особенность Python, которую не встретишь в других языках программирования. Она часто оказывается полезной (мы воспользуемся ею в главе 11)

```
mystery = 'секрет' * 5
```

Умножение строки? Мы уже сталкивались с этим в примере, когда возраст Тузика "12" повторялся 7 раз. Если умножить число на строку, то вы получите новую строку, в которой соответствующее число раз повторяется исходная строка

Возьмите карандаш
Решение

Помогите написать цикл для нахождения *всех* результатов, равных максимальному. На данный момент у нас есть приведенный ниже код. Заполните оставшиеся прочерки и только после этого сверьтесь с ответом в конце главы.

Начинаем с создания нового списка, который будет хранить номера растворов, соответствующих наибольшему результату

Далее просматриваем весь список результатов в поисках элементов с наибольшим значением

Здесь показан только новый код. Вы ведь в курсе, что нужно беречь деревья...

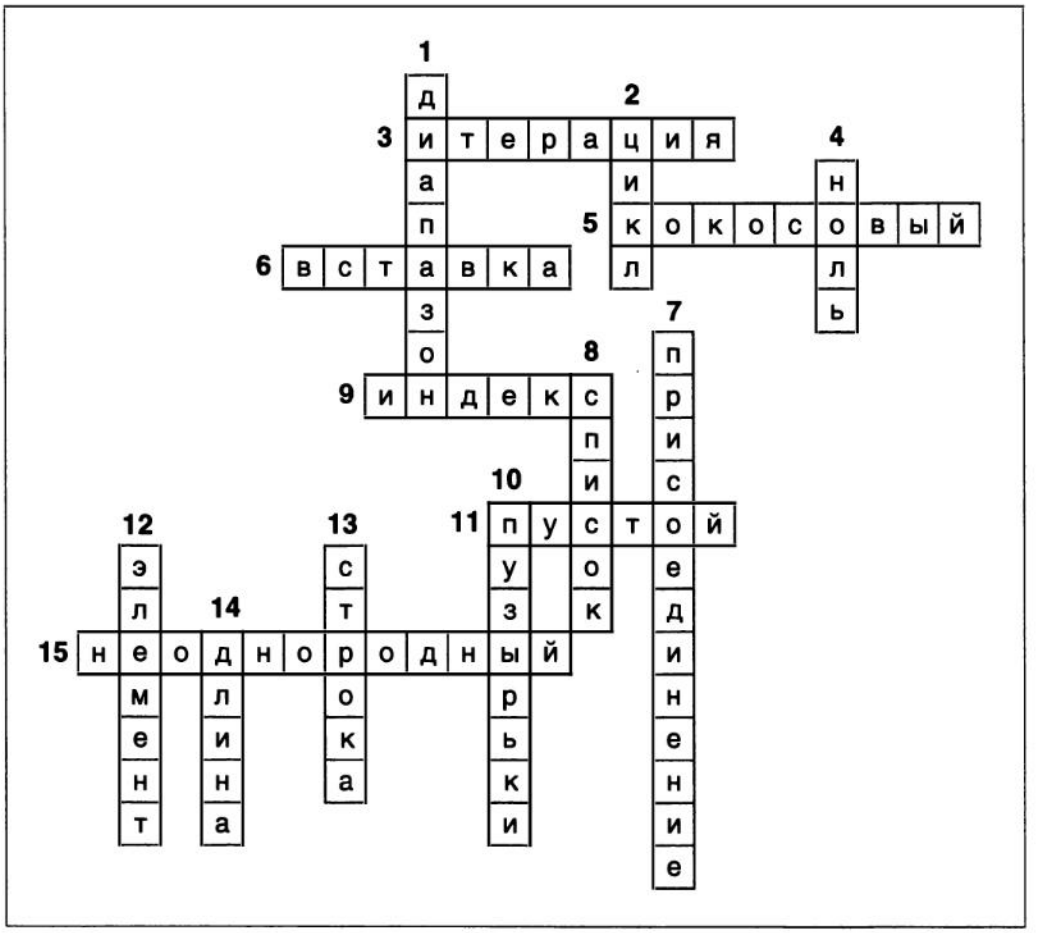
```
best_solutions = []
for i in range(length):
    if high_score == scores[i]:
        best_solutions.append(i)
```

На каждой итерации цикла мы сравниваем результат, находящийся по индексу i, с переменной high_score, и если они равны, то добавляем индекс раствора в список best_solutions с помощью функции append()

```
print('Растворы с наибольшим результатом:', best_solutions)
```

Наконец, выводим номера растворов с наибольшим результатом. Обратите внимание: мы используем функцию print() для вывода списка. Можно было бы создать новый цикл и выводить элементы списка один за другим, но, к счастью, функция print() делает это за нас (взглянув на результат, вы также увидите, что функция разделяет элементы запятыми, как нам и нужно)

Кроссворд: решение



5 Функции и абстракции

Функциональный код

Вы уже многое знаете. Переменные и типы данных, условные конструкции и циклы — этих инструментов вполне достаточно, чтобы написать почти любую программу. Но зачем останавливаться на достигнутом? Следующий шаг — научиться **создавать абстракции** в коде. Звучит пугающе, хотя ничего страшного в этом нет. Абстракции — ваш спасательный круг. Они упрощают процесс разработки, давая возможность создавать более эффективные и функциональные программы. Абстракции позволяют упаковать код в небольшие блоки, удобные для повторного использования. С ними вы перестанете концентрироваться на низкоуровневых деталях и сможете программировать на высоком уровне.

Возьмите карандаш

Проанализируйте приведенный ниже код. Что вы можете о нем сказать? Выберите любое число пунктов в списке либо дайте свою характеристику.

```
dog_name = "Тузик";
dog_weight = 40
if dog_weight > 20:
   print(dog_name, 'говорит ГАВ-ГАВ')
else:
print(dog_name, 'говорит гав-гав')

dog_name = "Смайли"
dog_weight = 9
if dog_weight > 20:
   print(dog_name, 'говорит ГАВ-ГАВ')
else:
   print(dog_name, 'говорит гав-гав')

dog_name = "Джексон"
dog_weight = 12
if dog_weight > 20:
   print(dog_name, 'говорит ГАВ-ГАВ')
else:
   print(dog_name, 'говорит гав-гав')

dog_name = "Филя"
dog_weight = 65
   print(dog_name, 'говорит ГАВ-ГАВ')
else:
   print(dog_name, 'говорит гав-гав')
```

- ☐ А. Все время один и тот же код, выглядит избыточно.
- ☐ Б. Утомительно все это набирать.
- ☐ В. Столько кода и так мало пользы.
- ☐ Г. Не самый читабельный код, который доводилось видеть.
- ☐ Д. Если мы захотим поменять то, как лает собака, придется вносить много изменений!
- ☐ Е. _____

Что не так с кодом?

Легко заметить, что одни и те же конструкции повторяются *снова* и *снова*. И что с того, спросите вы? Да, в общем-то, ничего. Программа ведь работает правильно! Давайте присмотримся повнимательнее к ней.

```
dog_name = "Тузик"
dog_weight = 40
if dog_weight > 20:
    print(dog_name, 'говорит ГАВ-ГАВ')
else:
    print(dog_name, 'говорит гав-гав')
```

Мы здесь проверяем вес собаки. Если он больше, чем 20, то выводим прописью "ГАВ-ГАВ". Если же он меньше или равен 20, то выводим строчными "гав-гав"

```
dog_name = "Смайли"
dog_weight = 9
if dog_weight > 20:
    print(dog_name, 'говорит ГАВ-ГАВ')
else:
    print(dog_name, 'говорит гав-гав')
```

А здесь мы... погодите-ка! Это ведь ТО ЖЕ самое! И в остальной части кода мы снова и снова повторяем одно и то же

```
dog_name = "Джексон"
dog_weight = 12
if dog_weight > 20:
    print(dog_name, 'говорит ГАВ-ГАВ')
else:
    print(dog_name, 'говорит гав-гав')

dog_name = "Филя"
dog_weight = 65
if dog_weight > 20:
    print(dog_name, 'говорит ГАВ-ГАВ')
else:
    print(dog_name, 'говорит гав-гав')
```

Отличаются только данные собаки

Другая собака, а код

Код кажется совершенно бесхитростным, и в то же время его утомительно писать, трудно читать и проблематично редактировать, если понадобится что-то изменить в будущем. Последнее утверждение станет вам понятным по мере приобретения опыта в программировании. В любой код рано или поздно приходится вносить изменения, а редактирование приведенной выше программы превратится в кошмар, поскольку одни и те же конструкции встречаются в ней многократно.

Что, если нужно добавить новое условие для маленьких собак весом в килограмм, которые лают по-другому ("тяв-тяв")? Сколько строк придется изменить в исходном коде для этого?

Что же нам делать?

Сила мысли

Как можно улучшить данную программу? Уделите несколько минут обдумыванию возможных вариантов.

Ах, если бы существовал способ повторного использования кода, чтобы всякий раз, когда он мне нужен, я могла бы просто вызвать его, а не вводить повторно! И чтобы можно было дать ему легко запоминающееся имя. И чтобы исправления можно было вносить в одном месте. Как было бы здорово! Знаю, это лишь мои мечты...

Превращение блока кода в функцию

Что, если я скажу вам, что блоку кода можно присвоить имя, позволяющее повторно вызывать его в любом месте программы? Скорее всего, вы воскликнете: "Что же вы раньше молчали?!"

В Python такая операция называется *объявлением функции*. Вам уже доводилось применять функции, например `print()`, `str()`, `int()` и `range()`. Давайте познакомимся с ними поближе.

Функции можно объявлять во всех языках программирования

Мы вызываем функцию, указывая ее имя в коде

После имени ставится скобка

Далее указывается ноль или более аргументов, которые передаются функции

```
print("Я функция")
```

В конце ставится закрывающая скобка

Вы наверняка заметили, что функции иногда возвращают значение, как, например, функция `str()`, которая возвращает строковое представление целого числа

При вызове функции предполагается, что она выполнит некое действие или решит какую-то задачу (например, выведет на экран передаваемые ей данные), а затем завершит свою работу, после чего программа продолжит выполняться со следующей строки кода. Так поступают все функции.

Вам не обязательно довольствоваться одними только встроенными функциями. Всегда можно создать свою собственную. Вот как это делается:

Чтобы создать функцию, начните с ключевого слова def, после которого укажите имя функции

Функция может иметь ноль или более параметров. Их можно рассматривать как переменные, которые хранят значения, передаваемые функции

В конце ставится двоеточие, которое, как вы знаете, начинает блок кода

```
def bark(name, weight):
    if weight > 20:
        print(name, 'говорит ГАВ-ГАВ')
    else:
        print(name, 'говорит гав-гав')
```

Вот блок кода, который будет использоваться повторно

В Python, как и в других языках программирования, этот блок кода называется телом функции

Когда интерпретатор Python видит этот код, он создает определение функции и сохраняет его для будущего использования. Тело функции в данный момент не выполняется. Код будет выполнен только при вызове функции

Теперь, когда вы умеете объявлять функции, давайте рассмотрим, как они используются.

Функция создана — можно использовать

Итак, вы создали функцию: выделили блок инструкций, дали ему имя и определили передаваемые параметры (имя собаки и ее вес). Пора поработать с ней.

Имея опыт работы с функциями `print()`, `str()`, `random()` и `range()`, вы знаете, что нужно делать дальше. Давайте вызовем функцию bark() несколько раз и посмотрим, что из этого выйдет.

```
bark('Тузик', 40)
bark('Смайли', 9)
bark('Джексон', 12)
bark('Филя', 65)
```

Давайте протестируем функцию `bark()` на всех известных нам собаках

Отлично, именно то, что мы и ожидали!

Оболочка Python

```
Тузик говорит ГАВ-ГАВ
Смайли говорит гав-гав
Джексон говорит гав-гав
Филя говорит ГАВ-ГАВ
>>>
```

Как работает функция

Мы только что создали функцию, успешно вызвали ее и получили ожидаемый результат, но что происходит "за кулисами"? Как на самом деле все это работает? Давайте проследим за выполняемыми действиями в пошаговом режиме.

Начинаем с функции `print()`, чтобы подготовиться

```
print('Подготовьте собак')

def bark(name, weight):
    if weight > 20:
        print(name, 'говорит ГАВ-ГАВ')
    else:
        print(name, 'говорит гав-гав')

bark('Тузик', 40)

print("Отлично, все готово")
```

Далее идет определение функции `bark()`

Теперь вызываем функцию `bark()` с аргументами `'Тузик'` и `40`

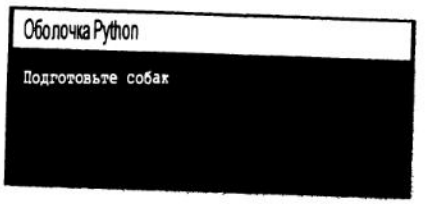
В конце вызываем функцию `print()`, чтобы сообщить об удачном завершении

СИЛА МЫСЛИ

Представьте себя в роли интерпретатора Python. Начните с первой строки и мысленно выполните каждую инструкцию программы. Есть ли в ней какие-то действия, которые вам непонятны?

Как и любой другой интерпретатор, начнем анализ кода с самого начала. В первой строке кода содержится вызов функции `print()`, которая выводит сообщение `'Подготовьте собак'` в оболочке Python.

За Кадром



Пока что мы лишь вызвали функцию `print()`, которая выводит сообщение на консоль

Начинаем отсюда

```
print('Подготовьте собак')

def bark(name, weight):
    if weight > 20:
        print(name, 'говорит ГАВ-ГАВ')
    else:
        print(name, 'говорит гав-гав')

bark('Тузик', 40)

print("Отлично, все готово")
```

Далее идет определение функции.

После выполнения функции `print()` интерпретатор Python переходит к объявлению функции `bark()`. На данном этапе интерпретатор не выполняет код функции, а всего лишь присваивает блоку кода имя `bark` и выполняет синтаксический анализ, после чего сохраняет параметры и тело функции для дальнейшего использования. После обработки интерпретатором функцию `bark()` можно вызвать в любом месте программы.

```
print('Подготовьте собак')

def bark(name, weight):
    if weight > 20:
        print(name, 'говорит ГАВ-ГАВ')
    else:
        print(name, 'говорит гав-гав')

bark('Тузик', 40)

print("Отлично, все готово")
```

Смысл этого фрагмента кода заключается в том, чтобы создать определение функции, а не вызвать ее. После его обработки функцию можно будет вызвать в любой момент по имени `bark()`

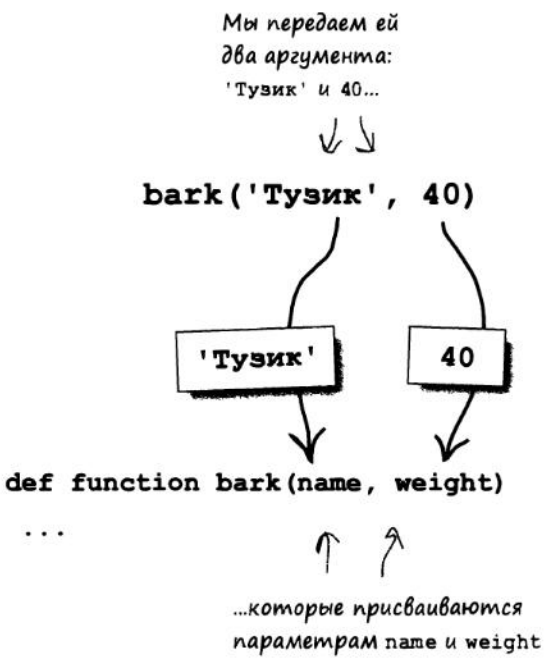
Затем мы вызываем функцию `bark()`.
В следующей строке программы интерпретатор сталкивается с вызовом функции `bark()`, которой передаются два аргумента: строка `'Тузик'` и число `40`.

Интерпретатор загружает определение функции `bark()` и присваивает значения `'Тузик'` и `40` ее параметрам `name` и `weight` соответственно.

За Кадром

Пора вызвать нашу функцию

```
print('Подготовьте собак')

def bark(name, weight):
    if weight > 20:
        print(name, 'говорит ГАВ-ГАВ')
    else:
        print(name, 'говорит гав-гав')

bark('Тузик', 40)

print("Отлично, все готово")
```



Параметры функции, `name` и `weight`, можно рассматривать как переменные, которые существуют, только пока функция выполняется. Они хранят значения аргументов, переданных при вызове функции.

Далее интерпретатор загружает код функции `bark()` и начинает ее выполнять.

Закрепим

Мы **вызываем** функцию.

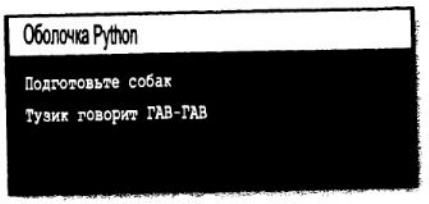
Мы передаем **аргументы** вызовам функций.

Функция имеет нуль или более **параметров**, получающих значения из аргументов ее вызова.

Функция начинает выполняться.
Самый важный момент здесь заключается в переходе от вызова функции `bark()` к выполнению ее тела. В этом месте линейный ход программы нарушается: мы временно покидаем основную последовательность инструкций и переходим в тело функции. Помните об этом.

В функции сначала проверяется, больше ли параметр `weight` числа 20. Поскольку функции передан аргумент 40, условие истинно, следовательно, выполняется первая ветвь условной инструкции.

Соответствующая функция `print()` выводит значение параметра `name` (в нашем случае он содержит строку `'Тузик'`), а также фразу `'говорит ГАВ-ГАВ'`.



На этом работа функции `bark()` завершается. Что же происходит дальше? А дальше управление передается туда, откуда была вызвана функция. Интерпретатор возвращается к выполнению последующих инструкций.

Помните: когда выполнение функции завершается, программа возвращается туда, откуда была вызвана функция, и интерпретатор начинает выполнять последующие инструкции.

После того как значения аргументов были присвоены каждому из параметров функции, настало время выполнить ее тело

```
print('Подготовьте собак')

def bark(name, weight):
    if weight > 20:
        print(name, 'говорит ГАВ-ГАВ')
    else:
        print(name, 'говорит гав-гав')

bark('Тузик', 40)

print("Отлично, все готово")
```

```
print('Подготовьте собак')

def bark(name, weight):
    if weight > 20:
        print(name, 'говорит ГАВ-ГАВ')
    else:
        print(name, 'говорит гав-гав')

bark('Тузик', 40)

print("Отлично, все готово")
```

После того как вызов функции `bark()` завершен, интерпретатор возвращается к выполнению последующих инструкций

Теперь выполняется инструкция, идущая после вызова функции. Итак, вызов функции `bark()` завершен, и единственное, что осталось, — функция `print()`, которая выводит сообщение `"Отлично, все готово"` в оболочке Python.

Оболочка Python

```
Подготовьте собак
Тузик говорит ГАВ-ГАВ
Отлично, все готово
```

За Кадром

```
print('Подготовьте собак')

def bark(name, weight):
    if weight > 20:
        print(name, 'говорит ГАВ-ГАВ')
    else:
        print(name, 'говорит гав-гав')

bark('Тузик', 40)

print("Отлично, все готово")
```

Мы наконец достигли конца программы

Тест-драйв

Вам пришлось многое узнать и выучить. Сохраните приведенный ниже код в файле *bark.py* и выполните команду **Run > Run Module**, чтобы убедиться в его работоспособности.

Добавьте весь этот код в файл

```
def bark(name, weight):
    if weight > 20:
        print(name, 'говорит ГАВ-ГАВ')
    else:
        print(name, 'говорит гав-гав')

bark('Тузик', 40)
bark('Смайли', 9)
bark('Джексон', 12)
bark('Филя', 65)
```

Посмотрите, насколько короче и понятнее он стал!

Вот что мы получим при тестировании кода

Оболочка Python

```
Тузик говорит ГАВ-ГАВ
Смайли говорит гав-гав
Джексон говорит гав-гав
Филя говорит ГАВ-ГАВ
>>>
```

Возьмите карандаш

В этом упражнении вам предстоит соединить точки. Начните с п. 1 и нарисуйте стрелку к каждому следующему пункту в той очередности, в которой будет выполняться программа. Можете писать комментарии, поясняющие, что происходит в программе. Мы уже нарисовали пару стрелок за вас.

Начинаем здесь

Вызываем функцию `print()` *и переходим к следующей строке*

```
1  print('Подготовьте собак')

2  def bark(name, weight):
a      if weight > 20:
б          print(name, 'говорит ГАВ-ГАВ')
       else:
           print(name, 'говорит гав-гав')

3  bark('Тузик', 40)

4  print("Отлично, все готово")
```

Возьмите карандаш

Ниже приведено несколько вызовов функции `bark()`. Запишите рядом с каждым из них, каким, по вашему мнению, будет результат или же программа должна будет выдать ошибку. Прежде чем продолжить, сверьтесь с ответом в конце главы.

Напишите, что, по-вашему, будет выведено на консоль

`bark('Снуппи', 20)` _____

`bark('Бакстер', -1)` _____

`bark('Скотти', 0, 0)` _____

Хм! Есть мысли по этому поводу?

`bark('Лори', "20")` _____

`bark('Сэми', 10)` _____

`bark('Рокки', 21)` _____

Функции делают код абстрактным.
Вернемся к первому примеру с собакой. В нем было много кода, и, взглянув на такой листинг, трудно понять, что делает эта программа. Потребовалось время, чтобы понять очевидное: мы хотим изучить лай каждой собаки. Причем лай определенным образом связан с размером животного: крупные собаки лают "ГАВ-ГАВ", а те, что поменьше, — "гав-гав".

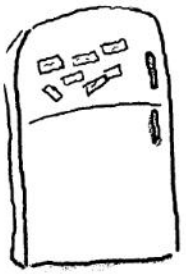
В результате мы решили обособить код, в котором описывается лай, и представить его в виде функции. Это позволило нам писать простые инструкции следующего вида:

```
bark('Тузик', 40)
```

Нам теперь не нужно думать о том, как реализуется лай в коде (в отличие от самой первой версии программы), — мы просто вызываем функцию `bark()`. Если, например, спустя два месяца вам понадобится добавить еще нескольких собак, можно будет воспользоваться этой функцией, не задумываясь над тем, как она работает. Такой подход позволяет сфокусироваться на целях программы, а не на низкоуровневых деталях ее работы.

Или вашим коллегам

Итак, мы берем фрагмент кода и оформляем его в виде функции, после чего используем ее в программе.



Код на магнитиках

У нас есть код, который записан с помощью набора магнитиков на дверце холодильника. Помогите составить программу, которая выдает показанный ниже результат. Только будьте внимательны: некоторые магнитики лишние. Сверьтесь с ответом, приведенным в конце главы.

```
,
,
,
:
}
}
:
}
(
def
)
{
how_should_I_get_there(800.3)
else:
    print('Пешком')
how_should_I_get_there
elif miles >= 2.0:
    print('Машиной')
miles
kilometer
if miles > 120.0:
    print('Самолетом')
how_should_I_get_there(.5)
how_should_I_get_there(2.0)
```

```
Оболочка Python
Самолетом
Машиной
Пешком
```

Не бойтесь задавать вопросы

В: Обязательно ли объявлять функцию до того, как она будет вызвана в коде?

О: Конечно, функцию нужно сначала объявить, чтобы иметь возможность вызвать ее в программе. Рассмотрим более сложный случай: в программе используются две функции, `f1` и `f2`, причем функция `f2` вызывается из функции `f1`. В таком случае допускается объявить функцию `f2` после функции `f1`, при условии, что функция `f1` не вызывается до объявления функции `f2`. Все дело в том, что при объявлении функции `f1` функция `f2` сама по себе не вызывается — это происходит только при вызове функции `f1` в основной программе. Впрочем, во избежание путаницы рекомендуется размещать объявления функций в начале файла.

В: Какие значения можно передавать функции?

О: В Python функциям разрешается передавать данные любых уже известных вам (а также тех, которые только предстоит выучить) типов: булевы значения, строки, числа и списки. Хотите верьте, хотите нет, но аргументом функции может даже быть другая функция. О том, зачем это может понадобиться, будет рассказано в приложении.

В: Я никак не могу понять, в чем разница между параметрами и аргументами?

О: Не стоит ломать голову над этим. Оба термина в сущности означают одно и то же. Аргументом называют значение, передаваемое функции при ее вызове в основной программе. А параметр является частью определения функции — он инициализируется значением аргумента, передаваемого функции.

В: Что произойдет, если перепутать порядок аргументов, передаваемых функции?

О: Ничего хорошего! Либо возникнет ошибка выполнения, либо программа будет работать неправильно. Внимательно изучите определение функции, чтобы понимать, какие аргументы она ожидает и в каком порядке.

К слову, существуют и другие способы передачи аргументов функциям, с которыми мы познакомимся далее.

В: Существуют ли какие-то правила именования функций?

О: Правила те же самые, что и в случае имен переменных (см. главу 2). Имя функции должно начинаться со знака подчеркивания или буквы и содержать буквы, числа и знаки подчеркивания. Разработчики Python придерживаются соглашения записывать имена функций в нижнем регистре, разделяя слова знаками подчеркивания, например `get_name` или `fire_cannon`.

В: Может ли функция вызывать другие функции?

О: Конечно! Это происходит регулярно. Обратите внимание на вызов функции `print()` из функции `bark()`. При написании собственных функций вы можете поступать точно так же.

Функции могут возвращать значения

Пока что мы имели дело только с функциями, которым *передавались аргументы*. Однако функция может и *возвращать значение* с помощью инструкции `return`.

Это новая функция `get_bark()`, которая возвращает описание лая в зависимости от веса собаки

```
def get_bark(weight):
    if weight > 20:
        return 'ГАВ-ГАВ'
    else:
        return 'гав-гав'
```

Если вес больше, чем 20, возвращается строка `'ГАВ-ГАВ'`

В противном случае возвращается строка `'гав-гав'`

Но выполняется только одна из них, поскольку при запуске инструкции `return` функция сразу же завершает свою работу

Функция может иметь ноль, одну или несколько инструкций `return`

Вызов функции, возвращающей значение

Итак, у нас есть функция `get_bark()`, которая получает вес собаки и возвращает описание ее лая. Рассмотрим, как это работает.

Функция вызывается так же, как и любая другая функция, только на этот раз она возвращает значение, поэтому мы записываем его в переменную `tuzik_bark`

```
tuzik_bark = get_bark(40)
print("Тузик лает", tuzik_bark)
```

Оболочка Python

```
Тузик лает ГАВ-ГАВ
>>>
```

Возьмите карандаш

Давайте попрактикуемся в определении значений, возвращаемых функциями. Запишите результат каждого вызова.

```
def make_greeting(name):
    return 'Привет, ' + name + '!'
```

```
def compute(x, y):
    total = x + y
    if (total > 10):
        total = 10
    return total
```

```
def allow_access(person):
    if person == 'Доктор Зло':
        answer = True
    else:
        answer = False

    return answer
```

Определена на предыдущей странице

`get_bark(20)` _____

`make_greeting('Снуппи')` _____

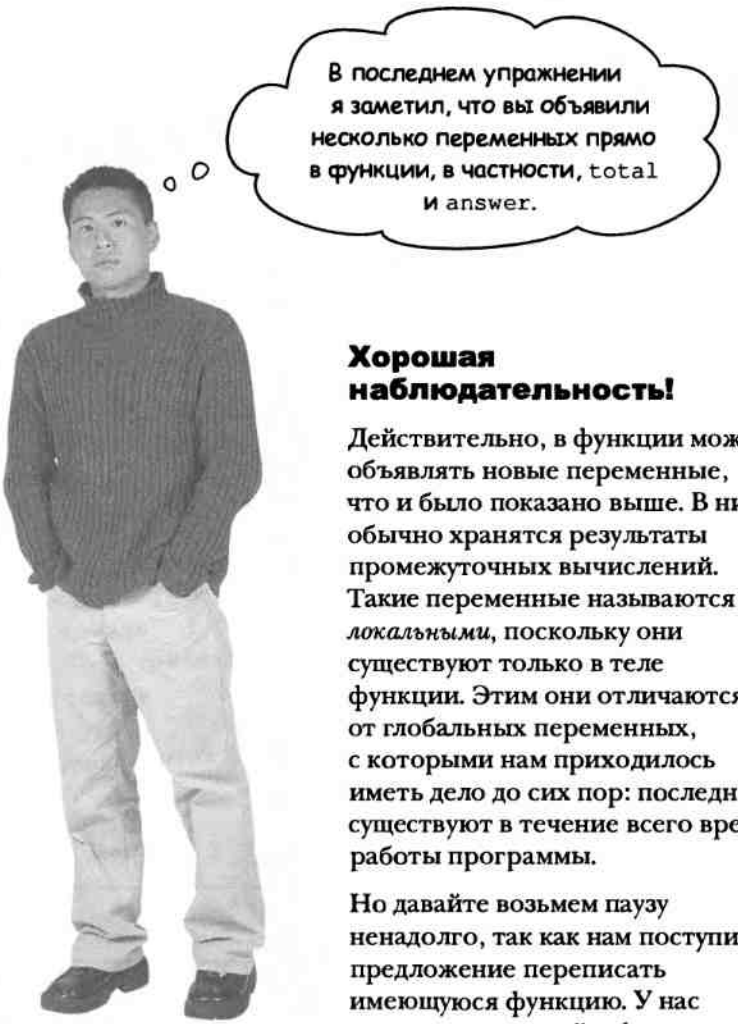
`compute(2, 3)` _____

`compute(11, 3)` _____

`allow_access('Тузик')` _____

`allow_access('Доктор Зло')` _____

Напишите, что возвращает каждая функция



Хорошая наблюдательность!

Действительно, в функции можно объявлять новые переменные, что и было показано выше. В них обычно хранятся результаты промежуточных вычислений. Такие переменные называются *локальными*, поскольку они существуют только в теле функции. Этим они отличаются от глобальных переменных, с которыми нам приходилось иметь дело до сих пор: последние существуют в течение всего времени работы программы.

Но давайте возьмем паузу ненадолго, так как нам поступило предложение переписать имеющуюся функцию. У нас достаточно знаний о функциях, их параметрах и способах вызова, чтобы справиться с этим заданием, после чего мы вернемся к разговору о переменных (локальных, глобальных и любых других).

Улучшение имеющегося кода

Расположенная по соседству интернет-компания разрабатывает программу, позволяющую пользователю выбрать аватарку, т.е. графическое представление для аккаунта в социальной сети. Проект пока находится на начальной стадии. Программа просто запрашивает у пользователя предпочтения: цвет волос, цвет глаз, пол и т.п. Предполагается, что на основе этих предпочтений программа сгенерирует аватарку.

Все бы ничего, вот только код программы получился слишком громоздким для решения столь простой задачи. Полный листинг приведен ниже. Обратите внимание: они попытались упростить пользователю задачу, задав значения по умолчанию, которые можно не вводить. Если просто нажать <Enter>, будет принято значение, уже заложенное в программе.

Для каждого атрибута аватарки мы выдаем запрос пользователю, указывая выбор по умолчанию, например темные волосы

```
hair = input("Цвет волос [темные]? ")
if hair == '':
    hair = 'темные'
print('Вы выбрали', hair)

hair_length = input("Длина волос [короткие]? ")
if hair_length == '':
    hair_length = 'короткие'
print('Вы выбрали', hair_length)

eyes = input("Цвет глаз [голубые]? ")
if eyes == '':
    eyes = 'голубые'
print('Вы выбрали', eyes)

gender = input("Пол [женский]? ")
if gender == '':
    gender = 'женский'
print('Вы выбрали', gender)

has_glasses = input("Носит очки [нет]? ")
if has_glasses == '':
    has_glasses = 'нет'
print('Вы выбрали', has_glasses)

has_beard = input("Носит бороду [нет]? ")
if has_beard = '':
    has_beard = 'нет'
print('Вы выбрали', has_beard)
```

Если пользователь просто нажимает клавишу <Enter>, мы присваиваем переменной значение по умолчанию

Мы также сообщаем о выборе пользователя

Повторяем для каждого атрибута

Выполнение кода

Давайте выполним имеющуюся программу, чтобы лучше понять, как она работает. Только предварительно изучите ее исходный код.

Оболочка Python

```
Цвет волос [темные]? светлые
Вы выбрали светлые
Длина волос [короткие]?
Вы выбрали короткие
Цвет глаз [голубые]? карие
Вы выбрали карие
Пол [женский]? мужской
Вы выбрали мужской
Носит очки [нет]?
Вы выбрали нет
Носит бороду [нет]? да
Вы выбрали да
```

Программа выдает запрос по каждому атрибуту. Мы либо вводим нужный атрибут, либо принимаем значение по умолчанию (например, короткие волосы и не носит очки), нажимая <Enter>

Для каждого атрибута программа подтверждает выбор пользователя, выводя его на консоль

Сила мысли

Очевидно, что код программы нуждается в абстракциях. Запишите здесь свои соображения насчет того, какую функцию (или функции) следует создать и как это должно выглядеть. В следующих разделах мы займемся этим вместе, но сначала проведите самостоятельную работу!

Создание абстракций в коде Выбора аватарки

Давайте выявим повторяющиеся места в программе, которые можно преобразовать в функцию.

Для каждого атрибута мы выводим запрос и ожидаем ответ от пользователя

Каждый раз мы задаем разные вопросы

И всякий раз у нас разные значения по умолчанию

```
hair = input("Цвет волос [темные]? ")
if hair == '':
    hair = 'темные'
print('Вы выбрали', hair)

hair_length = input("Длина волос [короткие]? ")
if hair_length == '':
    hair_length = 'короткие'
print('Вы выбрали', hair_length)

eyes = input("Цвет глаз [голубые]? ")
if eyes == '':
    eyes = 'голубые'
print('Вы выбрали', eyes)

.
.
.
```

Для каждого атрибута мы делаем одно и то же

Мы показываем код лишь для трех атрибутов, чтобы сэкономить бумагу

Мы сообщаем пользователю выбранное значение атрибута

Каждый атрибут присваивается собственной переменной, например `hair` или `eyes`

Представленные фрагменты кода отличаются только двумя аспектами: вопросом и значением по умолчанию. Они прекрасно подходят на роль параметров функции, поскольку при каждом вызове они будут отличаться. Начнем с объявления функции:

Назовем функцию `get_attribute()`

```
def get_attribute(question, default):
```

Мы используем два параметра: `question` для задаваемого вопроса, например "Цвет волос", и `default` для значения по умолчанию, например "темные"

Тело функции get_attribute()

Теперь можно приступить к написанию тела нашей функции. В первую очередь нужно сформировать строку вопроса и вывести приглашение на экран.

```
def get_attribute(query, default):
    question = query + ' [' + default + ']? '
    answer = input(question)
```

Составляем запрос к пользователю, конкатенируя параметры query и default с необходимыми символами

Выводим запрос к пользователю и получаем от него значение. Результат присваивается переменной answer

После этого, как и в исходной версии программы, нужно проверить, не выбрал ли пользователь значение по умолчанию, нажав клавишу <Enter> (в этом случае переменная answer будет содержать пустую строку). Затем нужно сообщить о выборе пользователя.

```
def get_attribute(query, default):
    question = query + ' [' + default + ']? '
    answer = input(question)
    if (answer == ''):
        answer = default
    print('Вы выбрали', answer)
```

Сравниваем переменную answer с пустой строкой. Если пользователь ничего не ввел, то записываем в переменную answer значение параметра default

Нам осталось решить всего одну задачу: вернуть значение answer программе, из которой была вызвана функция get_attribute(). Для этого мы воспользуемся инструкцией return.

```
def get_attribute(query, default):
    question = query + ' [' + default + ']? '
    answer = input(question)
    if (answer == ''):
        answer = default
    print('Вы выбрали', answer)
    return answer
```

Мы определили выбор пользователя, осталось только сообщить его вызывающей программе

Вызов функции get_attribute()

Итак, нам осталось включить в программу вызовы функции `get_attribute()` и получить от пользователя все необходимые характеристики аватарки.

```
def get_attribute(query, default):
    question = query + ' [' + default + ']? '
    answer = input(question)
    if (answer == ''):
        answer = default
    print('Вы выбрали', answer)
    return answer

hair = get_attribute('Цвет волос', 'темные')
hair_length = get_attribute('Длина волос', 'короткие')
eye = get_attribute('Цвет глаз', 'голубые')
gender = get_attribute('Пол', 'женский')
glasses = get_attribute('Носит очки', 'нет')
beard = get_attribute('Носит бороду', 'нет')
```

Для каждого из атрибутов мы создали отдельный вызов функции `get_attribute()`

Тест-драйв

Сохраните приведенный выше код в файле *avatar.py* и протестируйте его, выполнив команду **Run > Run Module**.

Еще раз взгляните на приведенный выше код. Обратите внимание на то, насколько он стал короче и понятнее! Не говоря уже о том, что такой код легче сопровождать, если в будущем потребуется вносить какие-либо изменения

Все именно так, как и было раньше

Оболочка Python

```
Цвет волос [темные]? светлые
Вы выбрали светлые
Длина волос [короткие]?
Вы выбрали короткие
Цвет глаз [голубые]? карие
Вы выбрали карие
Пол [женский]? мужской
Вы выбрали мужской
Носит очки [нет]?
Вы выбрали нет
Носит бороду [нет]? да
Вы выбрали да
```

По-серьезному

Переписывание кода программы с целью сделать его более структурированным, компактным и понятным, — привычная практика для каждого программиста. Этот процесс называется *рефакторинг*.

Функции в эфире

Интервью недели: "Посиделки с локальной переменной"

[Редакция Head First] Добро пожаловать в студию! Спасибо, что согласились прийти к нам.

[Локальная переменная] Рада побывать у вас в гостях.

В общем-то, мы мало что о вас знаем. Не могли бы вы вкратце рассказать о себе?

Если вам приходилось объявлять переменную внутри функции, то это была я.

Чем это отличается от объявления других переменных?

Я работаю только внутри функции. Это единственное место, где я доступна.

Что вы имеете в виду?

Представьте, что вы пишете функцию, в которой промежуточные результаты вычислений нужно сохранить в переменной. Вы объявляете эту переменную внутри функции и можете свободно использовать ее в теле данной функции. Переменная не существует до тех пор, пока функция не будет вызвана, и как только функция завершится, переменная исчезнет.

А что произойдет при следующем вызове функции?

Будет создана новая локальная переменная с тем же именем, которая просуществует ровно до завершения работы функции, после чего тоже исчезнет.

Какая от вас польза, если вы существуете только короткое время и все время исчезаете, как только функция завершается.

Что вы, я очень полезна! С моей помощью можно хранить временные результаты вычислений, выполняемых в функции. Без меня вам не обойтись. А когда функция завершается, я беру на себя задачу очистки. Вам не придется беспокоиться об удалении переменных, ставших ненужными.

Однажды нам довелось брать интервью у параметров функции. Какие у вас отношения?

Они мне как братья! Это тоже локальные переменные, но особого рода: их значения устанавливаются непосредственно при вызове функции. Каждый из параметров получает свое значение от соответствующего аргумента. Как и локальные переменные, параметры доступны только внутри функции и прекращают свое существование по завершении ее работы.

Не знали, что вы, оказывается, настолько близки. Тогда нельзя не спросить о другом известном персонаже: глобальной переменной.

Если хотите знать мое мнение, банальная выскочка. С ней куча проблем.

Серьезно?

Для тех, кто не в курсе: глобальная переменная объявляется вне функции. У нее глобальная область видимости.

Что, простите?

Область видимости — это та часть программы, в пределах которой можно обратиться к переменной, т.е. прочитать или изменить ее значение. Глобальная переменная доступна в пределах всей программы.

А у вас какая область видимости?

Как я уже говорила, она ограничена телом функции, как и у параметров.

И что же плохого в том, что глобальная переменная видна в пределах всей программы?

Это считается не лучшим решением с точки зрения дизайна программы.

Но почему, ведь такие переменные достаточно удобны для хранения данных?

Думаю, ваши читатели вскоре об этом узнают. Просто в больших программах это может вызывать проблемы.

Наверное, в следующий раз стоит пригласить вас обеих на посиделки.

Уж там-то мы подискутируем!

Сила мысли

Каким будет результат работы этой программы? Вы уверены? Обязательно протестируйте программу. Почему получен именно такой результат? Соответствует ли он вашим ожиданиям?

```
def drink_me(param):
    msg = 'Выпиваем ' + param + ' стакан'
    print(msg)
    param = 'пустой'

glass = 'полный'
drink_me(glass)
print('Стакан', glass)
```

Не вздумайте пропустить это упражнение!

И снова о переменных

Оказывается, при работе с переменными нужно знать не только о том, как их объявлять и как задавать их значения. После знакомства с функциями выяснилось, что бывают локальные и глобальные переменные. А еще у функций есть параметры, которые действуют как локальные переменные, только инициализируются автоматически при вызове функции. И, ко всему прочему, существует концепция области видимости переменной.

Возьмите карандаш

Опишите, к какой из категорий относится каждая переменная в следующей программе. Отметьте локальные, глобальные переменные и параметры. Сверьтесь с ответом в конце главы.

```
def drink_me(param):
    msg = 'Выпиваем ' + param + ' стакан'
    print(msg)
    param = 'пустой'

glass = 'полный'
drink_me(glass)
print('Стакан', glass)
```

Область видимости переменной

Область видимости переменной определяет, в какой части программы будет доступна эта переменная. Правила просты.

- **Глобальная переменная.** Доступна в любом месте программы, за одним исключением, о котором вы вскоре узнаете.
- **Локальная переменная.** Доступна только в теле функции, в которой она объявлена.
- **Параметр.** Доступен только в теле функции, в которой он объявлен.

Локальную переменную можно объявить только в функции

Рассмотрим концепцию области видимости переменных на уже знакомом вам примере.

Локальные переменные question и answer имеют локальную область видимости внутри функции get_attribute()

Параметры query и default тоже имеют локальную область видимости внутри функции get_attribute()

```
def get_attribute(query, default):
    question = query + ' [' + default + ']? '
    answer = input(question)
    if (answer == ''):
        answer = default
    print('Вы выбрали', answer)
    return answer

hair = get_attribute('Цвет волос', 'темные')
hair_length = get_attribute('Длина волос', 'короткие')
eye = get_attribute('Цвет глаз', 'голубые')
gender = get_attribute('Пол', 'женский')
glasses = get_attribute('Носит очки', 'нет')
beard = get_attribute('Носит бороду', 'нет')
```

Заметьте, что мы возвращаем значение через локальную переменную answer

Переменные hair, hair_length, eye, gender, glasses и beard являются глобальными и видимы в пределах всей программы

Не бойтесь задавать вопросы

В: Как можно вернуть локальную переменную из функции, если такого рода переменная исчезает после завершения функции?

О: Возвращая локальную переменную, вы возвращаете не ее саму, а ее значение. Тут уместна аналогия с эстафетой: вы передаете эстафетную палочку следующему участнику забега, и дальше бежит он, а не вы. Таким образом, по завершении работы функции значение переменной возвращается, но сама она удаляется.

Все то же самое справедливо и для локальных переменных

В: Что произойдет, если имя параметра совпадет с именем глобальной переменной? Такое вообще допускается?

О: Да, это допускается. А произойдет то, что внутри функции все обращения к глобальной переменной будут заменены ссылками на одноименную локальную переменную (параметр). Фактически глобальная переменная станет невидимой внутри функции. В подобных случаях говорят о *перекрытии* переменной (параметр перекрывает глобальную переменную). Такие ситуации довольно часто возникают даже в хорошо структурированных программах; это не то, чего следует избегать. Если совпадающее имя оправдано для параметра, смело оставляйте его, при условии, что функция не требует обращения к глобальной переменной.

В: Я понимаю, зачем нужны локальные переменные в функциях: в них хранятся результаты промежуточных вычислений. Но зачем нужны параметры? Почему нельзя ссылаться на значения глобальных переменных?

О: Технически так можно делать, но это ведет к появлению ошибок в программах. Используя параметры, можно писать универсальные функции, не зависящие от глобальных переменных. Сама программа определяет, какие именно аргументы следует передать в функцию.

Для примера возьмем созданную нами функцию `bark()`. Если бы в ней использовались глобальные переменные, то как она возвращала бы разные результаты для разных собак?

Передача переменных в функцию

Все еще размышляете над тем, какие результаты выводит программа, содержащая функцию `drink_me()`? Или пытаетесь понять, почему стакан не оказался пустым? Вы не одиноки. Все дело в трактовке значений и переменных, передаваемых в функцию. Попробуем разобраться.

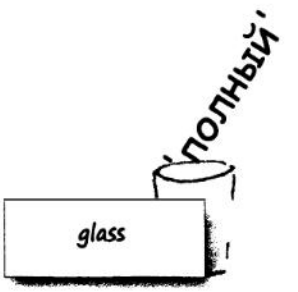
Забегая наперед: вот каким будет результат

```
Оболочка Python
Выпиваем полный стакан
Стакан полный
>>>
```

Это снова функция `drink_me()`. Пройдем программу пошагово, чтобы понять, почему стакан остается полным после вызова функции

```
def drink_me(param):
    msg = 'Выпиваем ' + param + ' стакан'
    print(msg)
    param = 'пустой'

glass = 'полный'
drink_me(glass)
print('Стакан', glass)
```

В этой инструкции, стоящей после определения функции, мы присваиваем переменной `glass` строку `'полный'`



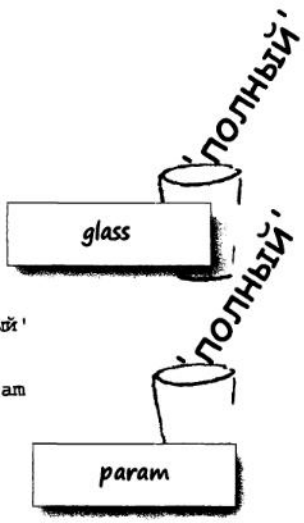
Вызов функции drink_me()

Давайте проанализируем, что происходит с переменными и параметрами при вызове функции `drink_me()`.

При вызове функции `drink_me()` *вычисляется значение аргумента* `glass`. *Полученное значение* `'полный'` *присваивается параметру* `param`

```
def drink_me(param):
    msg = 'Выпиваем ' + param + ' стакан'
    print(msg)
    param = 'пустой'

glass = 'полный'
drink_me(glass)
print('Стакан', glass)
```

Значение `'полный'` *присваивается параметру* `param`



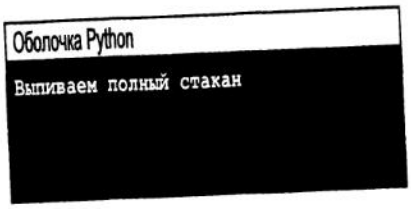
Вспомните из главы 2: когда мы присваиваем значение переменной (а `param` *можно считать переменной), мы выделяем участок в памяти для хранения значения (стакан на рисунке) и помечаем его именем переменной*

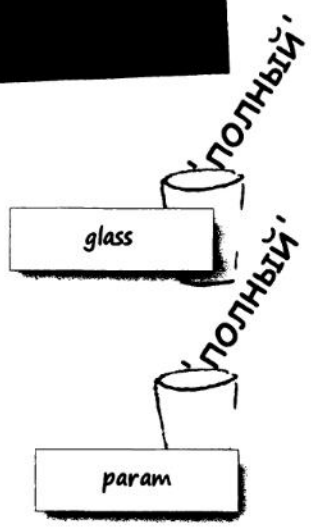
ПРИМЕЧАНИЕ: способ передачи аргументов в Python немного сложнее, чем описано здесь, особенно когда речь идет об объектах, но пока что это хорошая рабочая модель

Далее создаем строку `msg`, *используя значение параметра* `param`

```
def drink_me(param):
    msg = 'Выпиваем ' + param + ' стакан'
    print(msg)
    param = 'пустой'

glass = 'полный'
drink_me(glass)
print('Стакан', glass)
```

Выводим строку



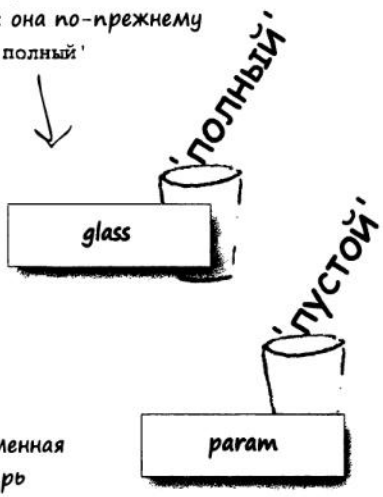


В данный момент переменная `param` *содержит* `'полный'`

А теперь самое важное: мы присваиваем переменной param значение 'пустой'

```
def drink_me(param):
    msg = 'Выпиваем ' + param + ' стакан'
    print(msg)
    param = 'пустой'

glass = 'полный'
drink_me(glass)
print('Стакан', glass)
```

Переменная glass не поменялась: она по-прежнему содержит 'полный'



Зато переменная param теперь содержит 'пустой'

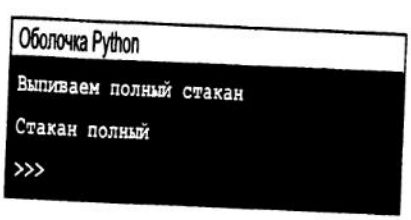
Наконец, мы возвращаемся из функции drink_me() и выполняем функцию print()

```
def drink_me(param):
    msg = 'Выпиваем ' + param + ' стакан'
    print(msg)
    param = 'пустой'

glass = 'полный'
drink_me(glass)
print('Стакан', glass)
```

Наш параметр param больше не существует, потому что вызов функции завершился

Опять-таки, для переменной glass ничего не поменялось: она по-прежнему содержит значение 'полный'

Функция print() выводит значение переменной glass, т.е. 'полный'



То есть, когда мы передаем переменную в функцию, мы фактически передаем не саму переменную, а ее значение?

Вот именно! Программисты часто говорят: "Когда я передаю переменную x в функцию `do_it()`..." Просто им так удобнее. В реальности имеется в виду "Когда я передаю *значение переменной* x в функцию `do_it()`..." Это ключевой момент – при вызове функции каждый из аргументов вычисляется *до того*, как он передается в функцию. Рассмотрим следующий код.

```
x = 10
do_it('секрет', 2.31, x)
```

Первый аргумент передается в функцию `do_it()` как строка `'секрет'`, второй аргумент – как число с плавающей точкой `2.31`, а вместо переменной x подставляется значение `10`. Таким образом, функция вообще ничего не знает о существовании переменной x. Она сразу получает число `10` и записывает его в соответствующий параметр.

Вскоре вы узнаете, что ситуация еще более усложняется при работе с объектами, но в любом случае следует помнить, что в функцию всегда передаются значения, а не сами переменные.

Использование глобальных переменных в функциях

Глобальные переменные не просто так ведь называются глобальными, правильно? Значит, они должны быть доступны и внутри функции. В Python, чтобы к глобальной переменной можно было обратиться внутри функции, ее сначала нужно объявить как глобальную, воспользовавшись специальным ключевым словом `global`.

```
greeting = 'Приветствую,'

def greet(name, message):
    global greeting
    print(greeting, name + '!', message)

greet('Джун', 'Увидимся!')
```

Создаем глобальную переменную `greeting`

Сообщаем функции о том, что собираемся использовать глобальную переменную

Обращаемся к глобальной переменной

```
Оболочка Python
Приветствую, Джун! Увидимся!
>>>
```

Конечно, при необходимости глобальную переменную можно изменить в самой функции.

```
greeting = 'Приветствую,'

def greet(name, message):
    global greeting
    greeting = 'Привет,'
    print(greeting, name + '!', message)

greet('Джун', 'Увидимся!')
print(greeting)
```

Меняем глобальную переменную на `'Привет'`

Выводим значение переменной `greeting` после вызова функции

```
Оболочка Python
Привет, Джун! Увидимся!
Привет,
>>>
```

Мы поменяли значение глобальной переменной `greeting` внутри функции на `'Привет'`

Не забывайте использовать ключевое слово `global`

Как видите, все достаточно просто: если требуется использовать глобальную переменную в функции, объявите ее внутри функции с ключевым словом `global`, чтобы сообщить Python о своих намерениях. Только будьте осторожны: многие до вас уже сталкивались с подобными проблемами. Если не указать ключевое слово `global`, глобальная переменная по-прежнему будет доступна в функции, но лишь для чтения. При попытке изменить ее значение произойдет одно из двух. Если это первое обращение к переменной внутри функции, Python посчитает, что имеет дело с локальной переменной. Если же значение переменной уже считывалось, то будет выдано сообщение об ошибке `UnboundLocalError`. Получив такую ошибку, проверьте, не происходит ли где-либо непреднамеренного совмещения локальных и глобальных переменных.

Посиделки

Тема дискуссии: **локальная и глобальная переменные спорят о том, кто из них важнее**

Локальная переменная

Согласна, мои задачи не слишком глобальны, зато я востребована. Всякий раз, когда нужно сохранить какое-то значение внутри функции, я тут как тут. Но как только функция завершается, меня и след простыл.

То, что ты везде, не значит, что программисты должны обращаться с тобой по-особому.

Я имею в виду, что программисты не должны натыкаться на глобальную переменную, когда она им не нужна.

Не буду спорить насчет простых программ, но что касается сложного кода, то советую почаще бывать на форуме stackoverflow.com. Там разработчики только и говорят: "Избегайте глобальных переменных!"

Проблема в том, что когда глобальных переменных становится много, в программе рано или поздно повстречаются две переменные с одинаковыми именами. А представь, каково другому программисту изучать такую программу. Без детального анализа трудно понять, какие глобальные переменные используются в программе.

Глобальная переменная

Моя область видимости глобальна. Я везде. Какие еще нужны аргументы?

Я ни в чем особом не нуждаюсь. Просто объявляешь переменную за пределами функции — и вот она я собственной персоной!

А кому, интересно, я мешаю? В простых программах это самый простой и понятный способ хранения нужных значений.

Ты серьезно? Я являюсь неотъемлемой частью практически всех языков программирования. Думаешь, их создатели просто так воспользовались моими услугами?

Ну конечно, программист налажал, а виновата я! Кого еще обвиним — цикл `for`?

Локальная переменная

Другая проблема заключается в том, что, просматривая фрагмент кода, содержащий глобальную переменную, невозможно определить, будет ли ее значение изменяться или применяться где-то еще, особенно когда речь идет о большой программе. В случае локальной переменной весь код перед глазами, ведь это одна функция.

Предположим, разрабатывается программа для управления шоколадной фабрикой.

В ней есть глобальная булева переменная, контролирующая запирание сливного крана в чане с кипящим шоколадом. Прежде чем заливать шоколад в чан, такую переменную обязательно нужно установить равной `True`.

Да, но тут появляется новый сотрудник, который где-то в дебрях программы управления шоколадной фабрикой устанавливает глобальную переменную равной `False`, не проверив перед этим, есть ли в чане шоколад. Ты понимаешь, насколько это опасно?

Вообще-то, хорошим решением станут объекты. Но об этом читатели узнают гораздо позже.

Не поверишь, но в объектах тоже используются локальные переменные.

Смотри, как бы не пришлось искать новую работу. Как только наши читатели наберутся опыта в программировании, они перестанут в тебе нуждаться.

Глобальная переменная

Ну и что тут такого? Не вижу никаких проблем.

Вот тут поподробнее.

Вот видишь, как все просто? Есть одна глобальная переменная, берешь и проверяешь, равна ли она `True` или `False`. Меня легко найти.

Здесь проблема в низкой квалификации персонала. Если переменная будет локальной, это что-то изменит?

Объекты? Что-то знакомое.

Смотри не лопни от счастья.

Ой, да ладно, без работы я не останусь! Я всегда нахожу себе применение.

Размышления о параметрах: значения по умолчанию и ключевые слова

Ранее мы уже говорили о том, как важно соблюдать порядок аргументов. Если передать не те аргументы и не в том порядке, работа функции будет нарушена. Представьте, что вы перепутали аргументы, задающие высоту и скорость самолета! Ничего хорошего из этого не выйдет.

Разогнаться до 10 000 км/ч на высоте 1000 метров? Это невозможно! Проверьте порядок передачи аргументов.

Подобные проблемы существуют во всех языках программирования. В Python предусмотрен другой, более гибкий способ передачи аргументов: параметры могут снабжаться *значениями по умолчанию* и *ключевыми словами*, что позволяет самому задавать порядок передачи аргументов в функцию. Значения по умолчанию и ключевые слова применяются во многих библиотеках и модулях Python, и вы вольны использовать их в собственных приложениях.

Вы познакомитесь с ними, когда мы перейдем к изучению модулей Python

Параметры по умолчанию

Параметры функции могут иметь значения по умолчанию. Ниже приведена упрощенная версия функции `greet()`, лишенная глобальной переменной.

Это обычный параметр, ожидающий получения аргумента

А этому параметру не важно, будет передан аргумент или нет, потому что у него есть значение по умолчанию

```
def greet(name, message='Все пучком!'):
    print('Привет,', name + '!', message)
```

Это стандартный текст сообщения, выводимый, если вызывающий код не передает аргумент для параметра `message`

Снабдив параметр значением по умолчанию, давайте посмотрим, как это работает.

Если мы вызываем функцию `greet()` *без аргумента для параметра* `message`*, то функция использует значение по умолчанию*

```
greet('Джон')
greet('Дженнифер', 'Как дела?')
```

А если мы передаем аргумент для параметра `message`*, то функция* `greet()` *использует значение аргумента*

Оболочка Python

```
Привет, Джон! Все пучком!
Привет, Дженнифер! Как дела?
>>>
```

Первыми всегда указываются обязательные параметры

Если значения по умолчанию задаются только для некоторых параметров, то первыми в объявлении функции должны быть указаны *обязательные параметры*. Что такое обязательные параметры? Любые параметры, не снабжаемые значениями по умолчанию, требуют *обязательной* передачи аргументов при вызове функции. Предположим, мы хотим расширить определение функции `greet()`.

Это новый параметр, являющийся обязательным, потому что для него не задано значение по умолчанию

```
def greet(name, message='Все пучком!', emoticon):
    print('Привет,', name + '!', message, emoticon)
```

Так поступать не следует!

Такое определение функции недопустимо. Python жалуется на то, что обязательный аргумент (non-default в терминологии Python) стоит после аргумента, имеющего значение по умолчанию

```
Оболочка Python
File "defaults.py", line 1
    def greet(name, message='Все
пучком!', emoticon):
                ^
SyntaxError: non-default argument
follows default argument
```

Если вы прочитали определение того, что такое аргумент и что такое параметр, то наверняка спросите, почему интерпретатор называет параметры аргументами. Мы задаем тот же вопрос, но с интерпретатором Python не поспоришь

И что тут неправильного? Почему так нельзя поступать? Дело в том, что в сложных ситуациях интерпретатор не сможет определить, какой аргумент какому параметру соответствует. Сейчас мы не будем рассматривать подобные ситуации, вы просто должны запомнить, что все обязательные параметры (`non-default` в терминологии Python) должны указываться перед параметрами, снабженными значениями по умолчанию. Прислушаемся к этому требованию и исправим код функции.

Теперь обязательные параметры идут первыми...

...а за ними идут необязательные

```
def greet(name, emoticon, message='Все пучком!'):
    print('Привет,', name + '!', message, emoticon)
```

Аргументы с ключевыми словами

До сих пор при вызове функций мы использовали *позиционные* аргументы. Это означает, что первый аргумент сопоставлялся с первым параметром функции, второй аргумент — со вторым и т.д. Но если назначить параметрам ключевые слова, то появится возможность указывать аргументы в произвольном порядке.

Чтобы понять, как это работает, попробуем вызвать нашу функцию `greet()` с использованием ключевых слов.

```
Оболочка Python
Привет, Джилл! Где ты была? отлично
>>>
```

```
greet(message='Где ты была?', name='Джилл', emoticon='отлично')
```

Чтобы использовать ключевое слово, укажите имя параметра, затем знак = и значение аргумента

Используя ключевые слова, мы можем указывать аргументы в произвольном порядке и даже опускать их, если у них есть значение по умолчанию. Главное, чтобы аргументы без ключевых слов шли в начале

Позиционную систему передачи аргументов и ключевые слова можно совмещать в одном вызове.

```
greet('Бетти', message='Салют!', emoticon=':)')
```

```
Оболочка Python
Привет, Бетти! Салют! :)
>>>
```

Здесь мы задаем позиционный параметр name, а для остальных параметров указываем ключевые слова

Как правильно использовать все доступные возможности

Параметры по умолчанию и ключевые слова являются характерными особенностями Python. Они удобны, когда у функции много аргументов, часть из которых может быть представлена стандартными значениями. В дальнейших примерах мы в основном не будем их применять, но при более углубленном изучении методик программирования на Python вы не раз с ними столкнетесь. Кроме того, знание этих возможностей понадобится вам при рассмотрении программных модулей в последующих главах.

УПРАЖНЕНИЕ

Закрепим полученные знания об аргументах функций, изучив следующий код. Попытайтесь предугадать возвращаемые им результаты и запишите их в приведенном ниже окне оболочки Python.

```
def make_sundae(ice_cream='ванильное', sauce='шоколадный', nuts=True,
                banana=True, brownies=False, whipped_cream=True):
    recipe = ice_cream + ' мороженое и ' + sauce + ' соус, '
    if nuts:
        recipe = recipe + 'с орехами, '
    if banana:
        recipe = recipe + 'с бананом, '
    if brownies:
        recipe = recipe + 'с брауни,'
    if not whipped_cream:
        recipe = recipe + 'не '
    recipe = recipe + 'содержит взбитые сливки.'
    return recipe

sundae = make_sundae()
print('Один десерт. Состав:', sundae)

sundae = make_sundae('шоколадное')
print('Один десерт. Состав:, sundae)

sundae = make_sundae(sauce='карамельный', whipped_cream=False, banana=False)
print('Один десерт. Состав:', sundae)

sundae = make_sundae(whipped_cream=False, banana=True,
                     brownies=True, ice_cream='фисташковое')
print('Один десерт. Состав:', sundae)
```

Оболочка Python

Запишите здесь результаты

Знаю, глава подходит к концу, но у меня остался еще один вопрос по поводу функций. Если в функции отсутствует инструкция `return`, то возвращает ли она что-либо?

Такая функция возвращает значение `None`. Это непростой вопрос, и, пожалуй, мы должны были рассмотреть его ранее.

Если функция не возвращает значение в явном виде с помощью инструкции `return`, значит, она возвращает значение `None` (не строку `'None'`, а именно *значение* `None`). Что это за значение такое, спросите вы? В программировании нередко используются специальные вычислительные сущности, такие как пустая строка, пустой список или булевы значения `True` и `False`. В нашем случае `None` означает отсутствие значения или ситуацию, когда значение не определено.

Какой же тип имеет это странное значение? Оно относится к типу данных `NoneType`. Навряд ли вам это о чем-то говорит, так что не будем слишком углубляться в дебри. Значение `None` существует только для того, чтобы помечать неопределенные или отсутствующие значения. В дальнейшем мы будем время от времени с ним сталкиваться, а пока подытожим все то, что нужно знать о типе данных `NoneType`.

Просто зафиксируйте это в голове, чтобы в будущем знать, с чем мы имеем дело.

`NoneType` — *очень странный тип*

Во многих языках программирования есть похожие значения: `NULL`, `null` *и* `nil`

Попытка ограбления, не нуждающаяся в расследовании

Закончив разговор с Лестрейдом, шефом полиции Скотленд-Ярда, Шерлок Холмс положил трубку, уселся в кресло напротив камина и взял в руки утреннюю газету. Доктор Ватсон вопросительно посмотрел на него.

– В чем дело, Ватсон? – невозмутимо произнес Холмс, не отрываясь от газеты.

– Так что же сказал Лестрейд? – спросил Ватсон.

– Они нашли вредоносный программный код в банковской системе, где наблюдалась подозрительная активность.

– И? – Ватсон сгорал от любопытства.

– Лестрейд переслал мне этот код, я просмотрел его и заверил инспектора, что беспокоиться совершенно не о чем. Преступник допустил глупую ошибку и никогда не сможет украсть деньги, – спокойно ответил Холмс.

– Но как вы догадались, Холмс? – на лице доктора Ватсона застыло недоумение.

– Элементарно, Ватсон! Нужно просто знать, на что смотреть! – воскликнул Шерлок Холмс. – И хватит донимать меня вопросами, я еще не дочитал газету.

Пока Холмс изучал утреннюю прессу, доктор Ватсон тайком взял его смартфон и открыл письмо от Лестрейда, чтобы просмотреть присланный программный код.

```
balance = 10500
camera_on = True

def steal(balance, amount):
   global camera_on
   camera_on = False
   if (amount < balance):
      balance = balance - amount

   return amount
   camera_on = True

proceeds = steal(balance, 1250)
print('Преступник: вы украли', proceeds)
```

Это сумма на банковском счете

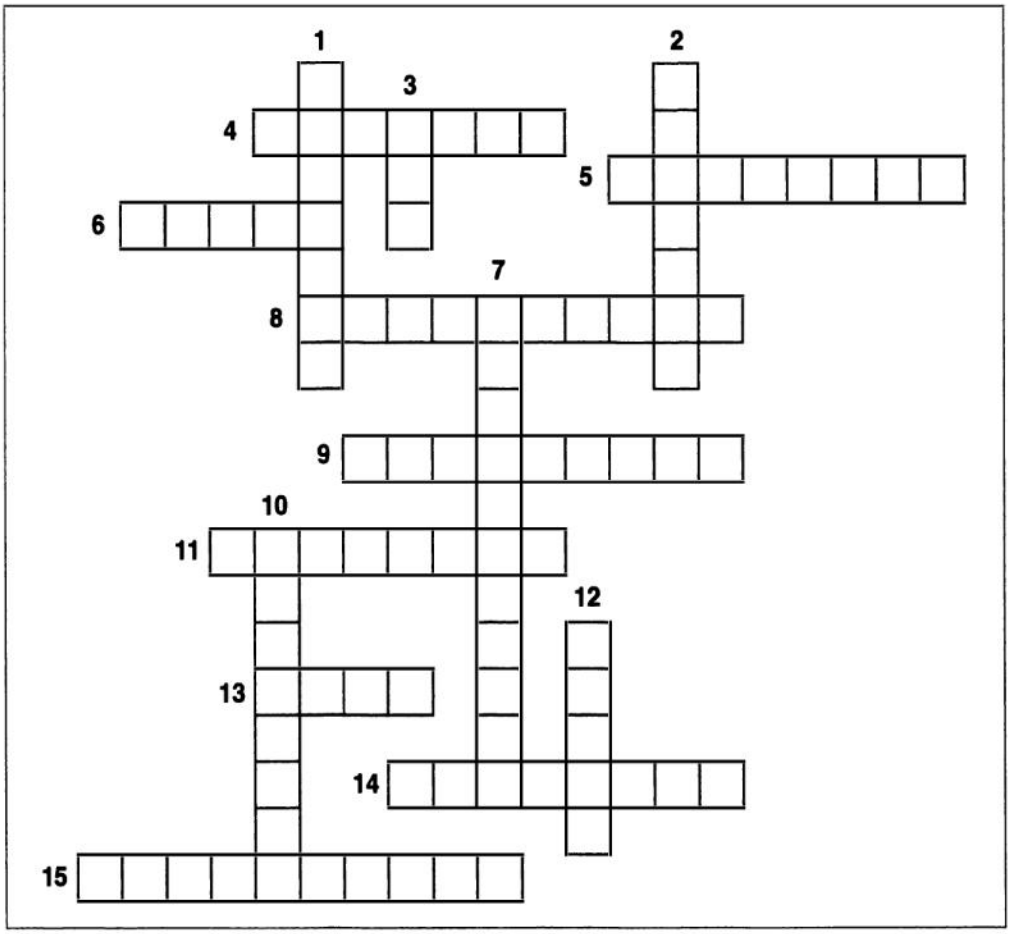
Почему Шерлок Холмс решил, что дело не стоит расследования? Как он узнал, что с помощью приведенного выше кода преступник не сможет украсть деньги со счета? В программе одна логическая ошибка или несколько?

САМОЕ ГЛАВНОЕ

- Функции позволяют обособлять блоки кода, делая возможным их повторное использование.
- У функции есть имя, параметры (не обязательно) и тело (блок инструкций).
- При вызове функции могут передаваться аргументы.
- В функцию можно передавать любые допустимые значения.
- Количество и порядок аргументов функции должны в точности соответствовать количеству и порядку параметров в ее объявлении. Но если аргументы задаются с помощью ключевых слов, то их порядок может быть произвольным.
- При вызове функции значения аргументов записываются в переменные-параметры, после чего начинает выполняться тело функции.
- Функции могут возвращать значения с помощью инструкции `return`.
- Чтобы получить значение, возвращаемое функцией, сохраните результат ее вызова в переменной.
- В функциях можно вызывать другие функции, как встроенные, так и пользовательские.
- Функции можно объявлять в любой очередности, при условии, что они вызываются после объявления.
- В функции можно создавать локальные переменные.
- Локальные переменные существуют, только пока функция выполняется.
- Областью видимости переменной называют ту часть программы, в которой эта переменная доступна.
- Переменные, объявленные вне функций, называются глобальными.
- Параметры функции трактуются как локальные переменные в ее теле.
- Если параметр функции называется так же, как и глобальная переменная, то он перекрывает ее в теле функции.
- Если необходимо обратиться к глобальной переменной в теле функции, следует объявить ее в функции с ключевым словом `global`.
- Функции делают программу более понятной и структурированной, упрощая ее последующее сопровождение.
- Функциональные абстракции позволяют сконцентрироваться на высокоуровневых аспектах работы программы и не заниматься низкоуровневыми деталями реализации функций.
- Процесс переработки кода называется рефакторингом.
- Для параметров можно задавать значения по умолчанию. Они будут задействованы, если при вызове функции соответствующие аргументы не указаны.
- Имена параметров можно использовать в качестве ключевых слов при вызове функции.

Кроссворд

Чтобы лучше усвоить тему функций, попробуйте решить кроссворд.



По горизонтали

4. Порядковый номер параметра функции
5. То, что хранится в переменной
6. Обращение к функции
8. Способ представления кода в виде функций
9. Переменная, объявленная внутри функции
11. Переменная, значение которой функция получает при вызове
13. Блок инструкций внутри функции
14. Значение, передаваемое в функцию
15. Переменная, объявленная вне функции

По вертикали

1. Передача функцией значения в вызывающую программу
2. Абстракция в коде
3. Название функции
7. Редактирование чужого кода
10. Графическое представление пользователя в социальной сети
12. Фамилия знаменитого сыщика

Возьмите карандаш
Решение

Проанализируйте приведенный ниже код. Что вы можете о нем сказать? Выберите любое число пунктов в списке либо дайте свою характеристику.

- ☑ А. Все время один и тот же код, выглядит избыточно.
- ☑ Б. Утомительно все это набирать.
- ☑ В. Столько кода и так мало пользы.
- ☑ Г. Не самый читабельный код, который доводилось видеть.
- ☑ Д. Если мы захотим поменять то, как лает собака, придется вносить много изменений!
- ☑ Е. *Стоило ли использовать одни и те же переменные снова и снова для разных собак?*

Возьмите карандаш
Решение

В этом упражнении вам предстоит соединить точки. Начните с п. 1 и нарисуйте стрелку к каждому следующему пункту в той очередности, в которой будет выполняться программа. Можете писать комментарии, поясняющие, что происходит в программе. Мы уже нарисовали пару стрелок за вас.

Начинаем здесь

Вызываем функцию print() и переходим к следующей строке

Мы обрабатываем определение функции, но не выполняем ее

```
1   print('Подготовьте собак')

2   def bark(name, weight):
a       if weight > 20:
б           print(name, 'говорит ГАВ-ГАВ')
        else:
            print(name, 'говорит гав-гав')

3   bark('Тузик', 40)

4   print("Отлично, все готово")
```

ПРОДОЛЖЕНИЕ НА СЛЕДУЮЩЕЙ СТРАНИЦЕ...

При вызове функции bark() мы возвращаемся к п. 2

Далее выполняется тело функции (пп. 2, а и 2, б)...

```
1  print('Подготовьте собак')

2  def bark(name, weight):
а      if weight > 20:
б          print(name, 'говорит ГАВ-ГАВ')
       else:
           print(name, 'говорит гав-гав')

3  bark('Тузик', 40)

4  print("Отлично, все готово")
```

...после чего мы возвращаемся из функции в п. 3 и переходим к п. 4

```
1  print('Подготовьте собак')

2  def bark(name, weight):
а      if weight > 20:
б          print(name, 'говорит ГАВ-ГАВ')
       else:
           print(name, 'говорит гав-гав')

3  bark('Тузик', 40)

4  print("Отлично, все готово")
```

Возьмите карандаш
Решение

Ниже приведено несколько вызовов функции `bark()`. Запишите рядом с каждым из них, каким, по вашему мнению, будет результат или же программа должна будет выдать ошибку.

`bark('Снуппи', 20)` Снуппи говорит гав-гав

`bark("Бакстер", -1)` Бакстер говорит гав-гав

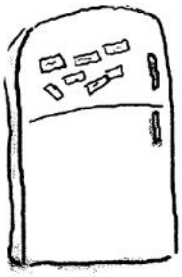
Наша функция `bark()` не проверяет, является ли вес собаки положительным. Поэтому все работает, ведь -1 меньше, чем 20

`bark('Скотти', 0, 0)` ERROR, bark() takes 2 positional arguments but 3 were given.

`bark('Лори', "20")` ERROR, '>' not supported between instances of 'str' and 'int'

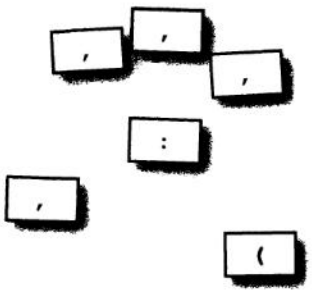
`bark('Сэми', 10)` Сэми говорит гав-гав

`bark('Рокки', 21)` Рокки говорит ГАВ-ГАВ



Код на магнитиках: решение

У нас есть код, который записан с помощью набора магнитиков на дверце холодильника. Помогите составить программу, которая выдает показанный ниже результат. Только будьте внимательны: некоторые магнитики лишние.




```
def how_should_I_get_there ( miles ) :
    if miles > 120.0:
        print('Самолетом')
    elif miles >= 2.0:
        print('Машиной')
    else:
        print('Пешком')

how_should_I_get_there(800.3)
how_should_I_get_there(2.0)
how_should_I_get_there(.5)
```

Функция получает `miles` *в качестве аргумента и...*

...проверяет расстояние, чтобы определить оптимальный способ, как добраться до места назначения: самолетом, машиной или пешком

`kilometer`

```
Оболочка Python
Самолетом
Машиной
Пешком
```

Возьмите карандаш

Решение

Давайте попрактикуемся в определении значений, возвращаемых функциями. Запишите результат каждого вызова.

`get_bark(20)`	'гав-гав'
`make_greeting('Скупши')`	'Привет, Снупи!'
`compute(2, 3)`	5
`compute(11, 3)`	10
`allow_access('Тузик')`	False
`allow_access('Доктор Зло')`	True

Возьмите карандаш

Решение

Опишите, к какой из категорий относится каждая переменная в следующей программе. Отметьте локальные, глобальные переменные и параметры. Сверьтесь с ответом в конце главы.

```
def drink_me(param):
    msg = 'Выпиваем ' + param + ' стакан'
    print(msg)
    param = 'пустой'

glass = 'полный'
drink_me(glass)
print('Стакан', glass)
```

параметр

локальная переменная

параметр

локальная переменная

глобальная переменная

глобальная переменная

УПРАЖНЕНИЕ [РЕШЕНИЕ]

Закрепим полученные знания об аргументах функций, изучив следующий код. Попытайтесь предугадать возвращаемые им результаты и запишите их в приведенном ниже окне оболочки Python.

```
def make_sundae(ice_cream='ванильное', sauce='шоколадный', nuts=True,
                banana=True, brownies=False, whipped_cream=True):
    recipe = ice_cream + ' мороженое и ' + sauce + ' соус, '
    if nuts:
        recipe = recipe + 'с орехами, '
    if banana:
        recipe = recipe + 'с бананом, '
    if brownies:
        recipe = recipe + 'с брауни, '
    if not whipped_cream:
        recipe = recipe + 'не '
    recipe = recipe + 'содержит взбитые сливки.'
    return recipe

sundae = make_sundae()
print('Один десерт. Состав:', sundae)

sundae = make_sundae('шоколадное')
print('Один десерт. Состав:', sundae)

sundae = make_sundae(sauce='карамельный', whipped_cream=False, banana=False)
print('Один десерт. Состав:', sundae)

sundae = make_sundae(whipped_cream=False, banana=True,
                     brownies=True, ice_cream='фисташковое')
print('Один десерт. Состав:', sundae)
```

Оболочка Python

```
Один десерт. Состав: ванильное мороженое и шоколадный соус, с орехами, с бананом, содержит
взбитые сливки.
Один десерт. Состав: шоколадное мороженое и шоколадный соус, с орехами, с бананом, содержит
взбитые сливки.
Один десерт. Состав: ванильное мороженое и карамельный соус, с орехами, не содержит взбитые сливки.
Один десерт. Состав: фисташковое мороженое и шоколадный соус, с орехами, с бананом, с брауни,
не содержит взбитые сливки.
>>>
```

Вот наши результаты

Загадка на пять минут: решение

```
balance = 10500
camera_on = True

def steal(balance, amount):
    global camera_on
    camera_on = False
    if (amount < balance):
        balance = balance - amount

    return amount
    camera_on = True

proceeds = steal(balance, 1250)
print('Преступник: вы украли', proceeds)
```

balance — это глобальная переменная...

...но она перекрывается данным параметром

Поэтому, когда вы меняете баланс в функции steal(), вы не меняете фактический банковский баланс!

Мы возвращаем величину украденного...

...но не используем ее для обновления реального баланса счета. В итоге баланс остается тем же, каким был изначально

Преступник думает, будто украл деньги, но это не так!

Мало того, что преступник не украл никаких денег, так он еще и забыл включить обратно камеру, что немедленно наведет полицию на след преступления. Помните: после выполнения инструкции `return` функция перестает выполняться, поэтому все инструкции после `return` игнорируются!

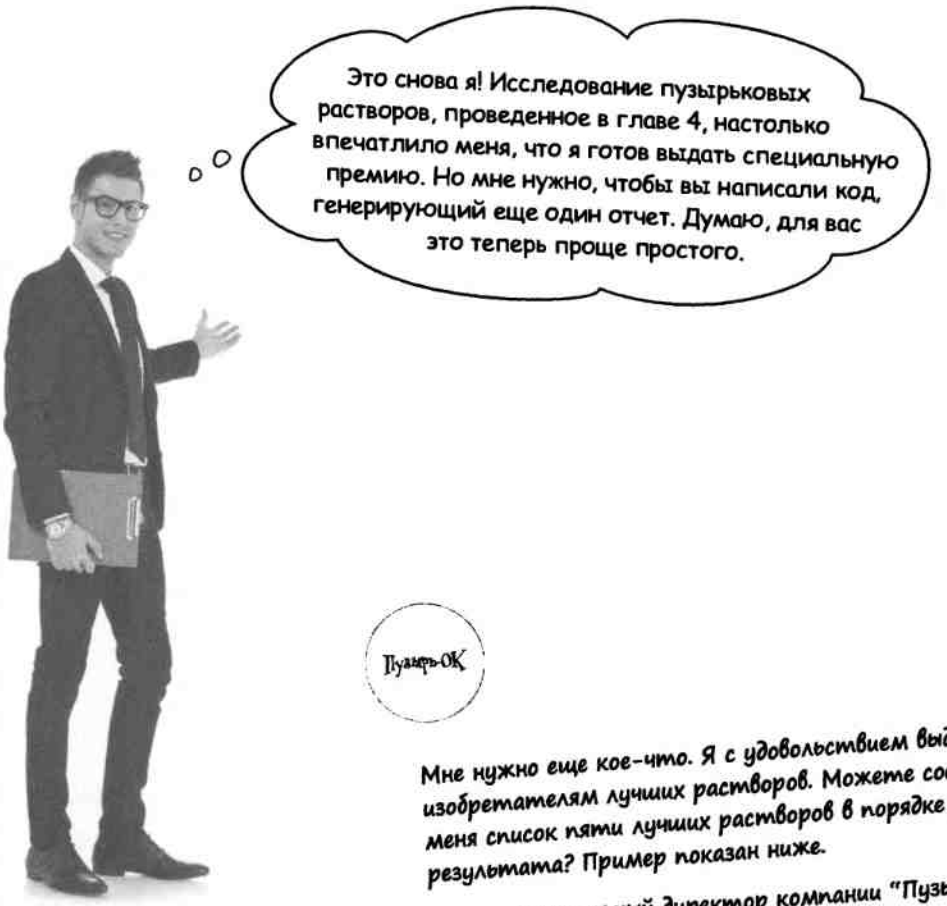
			1								2				
			в		3						ф				
		4	п	о	з	и	ц	и	я		у				
			з		м					5	з	н	а	ч	е	н	и	е
6	в	ы	з	о	в		я				к				
			р					7			ц				
		8	а	б	с	т	р	а	к	ц	и	я			
			т				е				я				
							ф								
		9	л	о	к	а	л	ь	н	а	я				
		10					к								
	11	п	а	р	а	м	е	т	р						
			в				о			12					
			а				р				х				
		13	т	е	л	о		и			о				
			а					н			л				
			р			14	а	р	г	у	м	е	н	т	
			к								с				
15	г	л	о	б	а	л	ь	н	а	я					

4 (продолжение) сортировка и вложенные циклы

вернемся к спискам и добавим ряд суперспособностей

Наведение порядка в данных

Иногда стандартный порядок сортировки данных нас не устраивает. Например, у вас есть список лучших результатов в аркадных играх, и вы хотите упорядочить его по названиям игр. Или же это может быть список сотрудников, регулярно опаздывающих на работу, и вы хотите узнать, кто из них проштрафился чаще остальных. Для решения подобных задач нужно знать, как сортировать данные и как самостоятельно настраивать порядок сортировки. В этой главе мы также изучим вложенные циклы и выясним, насколько эффективен написанный нами код. Нам предстоит существенно улучшить навыки вычислительного мышления!



Пузырь-ОК

Мне нужно еще кое-что. Я с удовольствием выдам премии изобретателям лучших растворов. Можете составить для меня список пяти лучших растворов в порядке убывания результата? Пример показан ниже.

— генеральный директор компании "Пузырь-ОК"

```
Лучшие пузырьковые растворы
1. Пузырьковый раствор #10 - результат: 68
2. Пузырьковый раствор #12 - результат: 60
3. Пузырьковый раствор #2 - результат: 57
4. Пузырьковый раствор #31 - результат: 50
5. Пузырьковый раствор #3 - результат: 34
```

Учтите, что это не настоящие результаты. Это лишь иллюстрация того, что мне нужно.

Спасибо!

Офисный диалог

Федор. Мы обязаны справиться, но как все это реализовать?

Юлия. Мы уже столько алгоритмов придумали, должны и сортировку побороть.

Игорь. Некоторые ученые всю жизнь посвятили разработке алгоритмов сортировки, а ты говоришь "побороть". Придется изучить имеющиеся алгоритмы и выбрать самый походящий для нашей задачи.

Федор. Ну, это надолго! Можно я сначала перекушу?

Игорь. Давай не будем терять время. Я уже все разузнал.

Федор. Что же ты сразу не сказал? И какой метод сортировки мы применим?

Игорь. Пузырьковый.

Юлия [гневно]. Очень смешно! Прямо открыл нам Америку. Мы и так знаем, что у нас пузырьковые тесты. А с сортировкой то что?

Игорь. Я, вообще-то, совершенно серьезен. Мы будем использовать пузырьковый метод сортировки. Он не самый эффективный, зато один из самых простых для понимания.

Федор. Что за странное название? Кто его придумал?

Игорь. Его так назвали, потому что в процессе сортировки элементы перемещаются к началу списка, или, как говорят, "всплывают", словно пузырьки. Ты сам все поймешь, когда мы приступим к реализации.

Юлия. Похоже, ты хорошо подготовился. Что ж, бери на себя руководство проектом.

Игорь. Я готов, давайте начинать.

Юлия. Чуть не забыла, я попросила Григория присоединиться к проекту. Я ведь не знала, что ты уже все продумал. Придется сказать ему, что мы все сделали без него. Напомните мне, если забуду, хорошо?

Пузырьковый метод сортировки

К написанию псевдокода пузырькового метода сортировки мы перейдем чуть позже, а пока давайте познакомимся с самим алгоритмом. Для этого попробуем отсортировать следующий список чисел:

```
[6, 2, 5, 3, 9]
```

Не отсортированный список

В общем случае, говоря о сортировке, подразумевают упорядочение чисел по возрастанию. Таким образом, ожидается, что после сортировки список будет приведен к такому виду:

```
[2, 3, 5, 6, 9]
```

Этот же список, отсортированный по возрастанию

Сразу подчеркнем, что пузырьковый метод сортировки подразумевает упорядочение элементов списка в несколько проходов. Как вы вскоре увидите, если текущий проход завершается перестановкой значений, то нужно совершить еще один проход. Если же на текущем проходе перестановок не было, значит, сортировка завершена.

Начало: первый проход

Мы начинаем со сравнения первого и второго элементов (имеющих индексы 0 и 1). Если первый элемент больше второго, меняем их местами.

```
[6, 2, 5, 3, 9]
```

Сравниваем числа. Первое больше, поэтому их нужно поменять местами

Далее сравниваем элементы с индексами 1 и 2. Если первый из них (6) больше второго (5), тоже меняем их местами.

```
[2, 6, 5, 3, 9]
```

Переставляем

Переходим к сравнению элементов с индексами 2 и 3. И здесь первый из элементов оказывается больше второго. Переставляем их.

```
[2, 5, 6, 3, 9]
```

Переставляем

Теперь нужно сравнить элемент номер 3 с элементом номер 4. В данном случае большим оказывается второй элемент, поэтому порядок элементов не меняется.

```
[2, 5, 3, 6, 9]
```

Не меняем

Заметьте, как число 6, которое было первым в списке, постепенно перемещается в конец списка

```
[2, 5, 3, 6, 9]
```

На этом первый проход завершен, но мы выполнили несколько перестановок, а значит, необходим еще один проход. Переходим ко второму этапу.

Второй проход

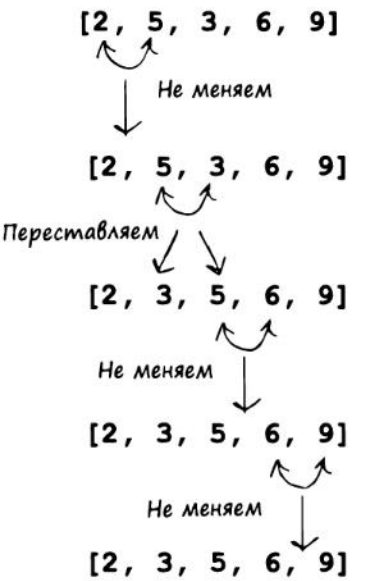
На втором проходе список просматривается повторно с самого начала. Вначале сравниваем элементы с индексами 0 и 1. Второй из них больше, поэтому перестановка не происходит.

Далее сравниваем элементы с индексами 1 и 2. В данном случае элемент с индексом 2 больше элемента с индексом 1, поэтому переставляем их местами.

Вам уже должно быть понятно, что будет дальше. Сравниваем элементы с индексами 2 и 3 — поскольку 5 меньше, чем 6, оставляем элементы в исходных позициях.

Продолжаем, сравнивая элементы с индексами 3 и 4. Значение элемента с индексом 4 больше, следовательно, перестановка не происходит.

Второй проход завершен, но поскольку имела место перестановка (элементы с индексами 1 и 2 менялись местами), нужно выполнить еще один проход.



Третий проход

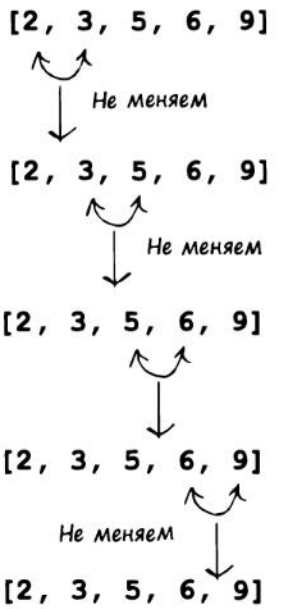
Как и ранее, повторно просматриваем список, начиная с первого элемента. Вначале сравним элементы с индексами 0 и 1. Первое значение меньше второго, поэтому элементы не переставляются.

Далее сравниваем элементы с индексами 1 и 2. В данном случае первый из них меньше второго, поэтому и тут переставлять элементы не нужно.

Переходим к сравнению элементов с индексами 2 и 3. Второе значение больше — оставляем оба элемента в исходных позициях.

Аналогичным образом сравним элементы с индексами 3 и 4. Второе значение больше, поэтому элементы остаются на прежних местах.

На третьем проходе мы не поменяли местами ни одну пару элементов, следовательно, сортировка списка завершена!



Возьмите карандаш

Теперь, когда вы получили представление о пузырьковом методе сортировки, давайте применим его для выполнения практического задания, чтобы закрепить приобретенные навыки. Отсортируйте приведенный ниже список так, как это делалось на двух предыдущих страницах. Если на каком-то этапе запутаетесь, сверьтесь с ответом в конце главы.

Мы не просим вас писать код. Мы лишь просим применить пузырьковый метод для сортировки этого списка

Вот наш список. Не пугайтесь строк, просто сравнивайте их в алфавитном порядке

```
['клубничный', 'банановый', 'ананасовый', 'ягодный']
```

Проход 1

Начните здесь первый проход

Псевдокод алгоритма пузырьковой сортировки

Познакомившись с пузырьковым методом сортировки, давайте опишем алгоритм его работы в виде псевдокода. Вы уже знаете все, что для этого нужно, поскольку мы задействуем лишь простые булевы операции и циклы. Однако циклы применяются здесь ранее не встречавшимся способом – один внутри другого. Такая конструкция называется *вложенным циклом*.

При первом знакомстве вложенные циклы могут несколько обескуражить, но не стоит отчаиваться, ведь вам уже доводилось иметь с ними дело при сортировке списка чисел и выполнении предыдущего упражнения. Внешний цикл представляет отдельные проходы пузырькового метода. А во внутреннем цикле выполняется сравнение (и, если нужно, перестановка) соседних элементов списка на каждом проходе. Принимая это во внимание, алгоритм пузырькового метода можно описать следующим образом.

Вспомним все, что мы узнали из главы 5 о функциях!

Мы хотим написать функцию `bubble_sort()`, которая получает список в качестве параметра

```
DEFINE функция bubble_sort(list):
    DECLARE переменная swapped и присвоить ей True

    WHILE swapped:
        SET swapped равно False
        FOR переменная i in range(0, len(list)-1)
            IF list[i] > list[i+1]:
                DECLARE переменная temp и присвоить ей list[i]
                SET list[i] равно list[i+1]
                SET list[i+1] равно temp
                SET swapped равно True
```

Нам нужна переменная для отслеживания перестановок на текущем проходе. Изначально устанавливаем ее равной True, чтобы сделать первый проход

Мы используем цикл `while`, который выполняется, пока переменная `swapped` равна True

На каждой итерации цикла `while` выполняется новый проход сортировки

Первое, что мы делаем в цикле, — задаем переменную `swapped` равной False

В цикле `for` мы проходим каждый элемент списка (кроме последнего) и выполняем сравнение. Если он больше следующего элемента, то меняем их местами

Псевдокод бывает разным. Здесь мы имитируем код, но это все равно не код, по крайней мере не Python

Если были перестановки, задаем переменную `swapped` равной True. Это означает, что по завершении цикла `for` будет выполнен еще один проход

Сила мысли

В приведенном выше псевдокоде список сортируется по возрастанию. Что нужно поменять в случае сортировки по убыванию?

Офисный диалог продолжается...

Юлия. Думаю, да, если я правильно понимаю работу обоих циклов.

Игорь. Тут все просто: имеем цикл `while` и вложенный в него цикл `for`.

Юлия. Внешний цикл `while` выполняется до тех пор, пока переменная `swapped` равна `True`.

Игорь. Да, каждая его итерация соответствует одному проходу по элементам сортируемого списка.

Юлия. По-моему, во всех наших примерах это всегда происходило по три раза.

Игорь. Это чистое совпадение: алгоритм допускает произвольное количество проходов. В худшем случае оно будет равно числу элементов списка.

Юлия. То есть, если список состоит из 100 элементов, то его сортировка может длиться 100 проходов?

Игорь. Да, именно так!

Юлия. Не многовато ли?

Игорь. Ну, это же худший случай, когда исходный список полностью отсортирован по убыванию.

Юлия. Понятно, а что насчет внутреннего цикла `for`? Что происходит в нем?

Игорь. В этом цикле каждый элемент списка сравнивается со следующим элементом. Если текущий элемент больше следующего, то они меняются местами.

Юлия. То есть в этом цикле число итераций равно количеству элементов в списке, причем не в худшем случае, а всегда?

Игорь. Технически число итераций равно количеству элементов списка минус один, а в остальном ты права.

Юлия. Для больших списков получаем что-то совсем уж много итераций.

Игорь. Ну да. Если список содержит 100 элементов, то в худшем случае получаем 100 проходов по 100 сравнений в каждом, итого 100 * 100 = 10 000 сравнений.

Юлия. Ого!

Игорь. Что поделать, пузырьковый метод славится своей простотой, но не эффективностью. Почему, по-твоему, столько людей до сих пор занимаются разработкой быстрых алгоритмов сортировки? К счастью, наши списки очень короткие, поэтому пузырьковый метод вполне для них подходит.

Юлия. В цикле `for` происходит еще кое-что: если имела место перестановка, переменная `swapped` устанавливается равной `True`, что означает необходимость еще одного прохода.

Игорь. Абсолютно верно!

Учимся понимать

Возьмите на себя роль интерпретатора Python и мысленно выполните приведенные ниже фрагменты кода, попытавшись предугадать их результаты. Завершив упражнение, сверьтесь с ответом в конце главы.

```
for i in range(0, 4):
    for j in range(0, 4):
        print(i * j)
```

```
for word in ['як', 'кот', 'пума', 'ягуар', 'собака']:
    for i in range(2, 7):
        letters = len(word)
        if (letters % i) == 0:
            print(i, word)
```

Вспомните, что операция деления по модулю означает нахождение остатка от деления. Например, 4 % 2 = 0, а 4 % 3 = 1

```
full = False

donations = []
full_load = 45

toys = ['робот', 'кукла', 'мяч', 'волчок']

while not full:
    for toy in toys:
        donations.append(toy)
        size = len(donations)
        if (size >= full_load):
            full = True

print('Полон', len(donations), 'игрушек')
print(donations)
```

Реализация пузырьковой сортировки на Python

Наш псевдокод близок к реальному коду, поэтому перевести его на Python не составляет труда. Вот что у нас получилось.

Это наша функция, аргументом которой служит список

```
def bubble_sort(scores):
    swapped = True
```

Мы устанавливаем переменную swapped равной True, чтобы сделать первый проход

```
    while swapped:
        swapped = False
        for i in range(0, len(scores)-1):
            if scores[i] > scores[i+1]:
                temp = scores[i]
                scores[i] = scores[i+1]
                scores[i+1] = temp
                swapped = True
```

Вложение циклов: цикл for внутри цикла while

Цикл while выполняется, пока переменная swapped равна True. Мы просматриваем весь список, при необходимости делая перестановки

Именно так мы выполняли перестановку переменных в главе 2!

Функция ничего не возвращает, потому что мы выполнили перестановку фактических значений списка. Другими словами, мы отсортировали оригинальный список

Тест-драйв

Создайте новый файл *sort.py*, скопируйте в него приведенный выше код и добавьте в конец следующие инструкции для тестирования.

```
scores = [60, 50, 60, 58, 54, 54,
          58, 50, 52, 54, 48, 69,
          34, 55, 51, 52, 44, 51,
          69, 64, 66, 55, 52, 61,
          46, 31, 57, 52, 44, 18,
          41, 53, 55, 61, 51, 44]

bubble_sort(scores)
print(scores)

smoothies = ['кокосовый', 'клубничный', 'банановый', 'ананасовый']
bubble_sort(smoothies)
print(smoothies)
```

Оператор > в Python работает и со строками, что позволяет отсортировать список строк

Оболочка Python

```
[18, 31, 34, 41, 44, 44, 44, 46, 48, 50,
50, 51, 51, 51, 52, 52, 52, 52, 53, 54,
54, 54, 55, 55, 55, 57, 58, 58, 60, 60,
61, 61, 64, 66, 69, 69]
['ананасовый', 'банановый', 'клубничный',
'ягодный']
>>>
```

Все отсортировано!

Нам необходимо отсортировать список результатов так, чтобы вначале шли растворы с наилучшими результатами. Другими словами, список должен быть отсортирован по убыванию, а не по возрастанию. Для этого достаточно поменять оператор сравнения в функции сортировки с > на <. Внесите изменение в файл и проверьте получаемый результат.

```
def bubble_sort(scores):
    swapped = True

    while swapped:
        swapped = False
        for i in range(0, len(scores)-1):
            if scores[i] < scores[i+1]:
                temp = scores[i]
                scores[i] = scores[i+1]
                scores[i+1] = temp
                swapped = True
```

Это все, что нужно изменить для сортировки списка по убыванию

Вот результаты: списки отсортированы по убыванию

```
Оболочка Python
[69, 69, 66, 64, 61, 61, 60, 60, 58, 58,
57, 55, 55, 55, 54, 54, 54, 53, 52, 52,
52, 52, 51, 51, 51, 50, 50, 48, 46, 44,
44, 44, 41, 34, 31, 18]
['ягодный', 'клубничный', 'банановый',
'ананасовый']
>>>
```

Мы почти справились. Осталось сгенерировать отчет, содержащий 5 лучших растворов с номерами и результатами.

Федор. Игорь молодец, написал весь код сортировки! Но кое-чего не хватает. Мы сортируем список результатов, однако не запоминаем номера исходных растворов. Как же мы узнаем номера растворов-победителей? Их ведь нужно включить в отчет.

Юлия. Согласна. Что же нам делать?

Федор. Игорь, конечно, крутой программист, но я могу улучшить его код. Что, если мы создадим еще один, параллельный, список `solutions_numbers`, элементы которого равны своим индексам, например `[0, 1, 2, 3, ..., 35]`? Тогда при сортировке списка результатов мы аналогичным образом отсортируем и список индексов. В итоге каждый индекс будет находиться в той же позиции, что и соответствующий ему результат.

Юлия. Но как создать список, элементы которого равны своим индексам?

Федор. Вспомни, для этого есть функции `range()` и `list()`.

```
number_of_scores = len(scores)
solution_numbers = list(range(number_of_scores))
```

Определяем длину списка

Создаем диапазон от 0 до длины списка (минус 1) и используем функцию `list()` для преобразования диапазона в список [0, 1, 2, ...]

Юлия. Интересно! Я, кажется, начинаю понимать ход твоих мыслей.

Федор. Некоторые вещи проще показать, чем объяснить. Давай взглянем на готовый код...

Правильное определение номеров растворов

Федор оказался прав. Мы сортируем список результатов, но при этом теряем информацию об их исходных позициях, по которым мы сопоставляем результаты с растворами. Решение заключается в том, чтобы создать второй список, который будет содержать только номера растворов. При сортировке списка результатов мы будем синхронно сортировать список номеров в той же последовательности. Рассмотрим программный код.

По-серьезному

Какой результат возвращает приведенное ниже выражение? Анализируйте его от внутренних скобок к внешним.

```
list(range(number_of_scores))
```

Равно целочисленной длине списка результатов: 36

Возвращает диапазон от 0 до 35

Возвращает список, содержащий числа от 0 до 35

Теперь мы передаем функции `bubble_sort()` два списка: список результатов и список соответствующих номеров растворов

```
def bubble_sort(scores, numbers):
    swapped = True

    while swapped:
        swapped = False
        for i in range(0, len(scores)-1):
            if scores[i] < scores[i+1]:
                temp = scores[i]
                scores[i] = scores[i+1]
                scores[i+1] = temp
                temp = numbers[i]
                numbers[i] = numbers[i+1]
                numbers[i+1] = temp
                swapped = True

scores = [60, 50, 60, 58, 54, 54,
          58, 50, 52, 54, 48, 69,
          34, 55, 51, 52, 44, 51,
          69, 64, 66, 55, 52, 61,
          46, 31, 57, 52, 44, 18,
          41, 53, 55, 61, 51, 44]

number_of_scores = len(scores)
solution_numbers = list(range(number_of_scores))

bubble_sort(scores, solution_numbers)
```

Все остальное работает так же, как и раньше...

...за исключением того, что после перестановки значений в списке результатов мы также переставляем аналогичные значения в списке номеров

Если вы считаете это избыточным кодом, то вы правы. В следующей главе вы узнаете, как убрать дублирующийся код

Здесь мы создаем список `solution_numbers`, хранящий номера всех растворов (они соответствуют исходным индексам в списке результатов)

Передаем оба списка в функцию сортировки

Возьмите карандаш

Теперь в программе создаются сразу два списка: один хранит результаты исследований, упорядоченные по убыванию, а второй — соответствующие им номера растворов. Напишите код вывода отчета на основе этих двух списков.

Пузырь-ОК

```
Лучшие пузырьковые растворы
1. Пузырьковый раствор #10 - результат: 68
2. Пузырьковый раствор #12 - результат: 60
3. Пузырьковый раствор #2 - результат: 57
4. Пузырьковый раствор #31 - результат: 50
5. Пузырьковый раствор #3 - результат: 34
```

Наш отчет должен выглядеть так. И не забывайте о том, что это не фактические данные, выдаваемые отчетом, а всего лишь пример от директора

Тест-драйв

Добавьте код из предыдущего упражнения (приведен ниже) в файл *sort.py*, заменив ненужные инструкции тестирования. Проверьте работу программы.

```
print('Лучшие пузырьковые растворы')
for i in range(0, 5):
    print(str(i+1) + '.',
        'Пузырьковый раствор #' + str(solution_numbers[i]),
         '- результат:', scores[i])
```

Именно так, как и хотел генеральный директор. Он будет счастлив!

Оболочка Python

```
Лучшие пузырьковые растворы
1. Пузырьковый раствор #11 - результат: 69
2. Пузырьковый раствор #18 - результат: 69
3. Пузырьковый раствор #20 - результат: 66
4. Пузырьковый раствор #19 - результат: 64
5. Пузырьковый раствор #23 - результат: 61
>>>
```

Отличная работа!

Это была короткая, но непростая глава. Вам пришлось изучить немало новых концепций, на осмысление которых потребуется время. А пока наслаждайтесь заслуженным призом. Вы хорошо потрудились и вправе немного отдохнуть.

Глава, впрочем, еще не закончена. Нам предстоит подвести итоги и не только...

САМОЕ ГЛАВНОЕ

- Существует много алгоритмов сортировки, обладающих разной сложностью и эффективностью.
- Пузырьковая сортировка — это простейший алгоритм, в котором элементы списка сравниваются и переставляются проход за проходом.
- Пузырьковая сортировка завершается, если на очередном проходе не было сделано ни одной перестановки.
- В большинстве языков программирования есть встроенные функции сортировки.
- Цикл, находящийся внутри другого цикла, называется вложенным.
- Вложенные циклы усложняют структуру программы и могут увеличивать время ее выполнения.
- Полезно изучать алгоритмы сортировки, особенно те, которые включены в стандартную библиотеку языка программирования. Ах да, вы ведь еще не в курсе...

Парни, вы не поверите. Я рассказала Григорию о том, как мы решили задачу, и он долго смеялся. Оказывается, можно было вообще не писать собственный код, так как в Python есть встроенные функции сортировки!

Встроенные средства сортировки есть в большинстве языков программирования.

Да, во многих современных языках программирования имеются функции сортировки. Поэтому чаще всего нет никакой необходимости писать собственный код сортировки. Во-первых, незачем изобретать колесо, а во-вторых, встроенные функции работают более эффективно, чем пузырьковая сортировка, и выполняются значительно быстрее. На самостоятельную реализацию таких функций у вас уйдет слишком много времени. Так почему бы не воспользоваться готовыми решениями?

Но не подумайте, что вы зря читали эту главу. Приемы, которые вы освоили в ходе изучения пузырьковой сортировки, в частности, вложенные списки, пригодятся вам при реализации многих алгоритмов. К тому же пузырьковая сортировка — то, с чего все начинают.

А что касается Python, то для сортировки списка достаточно вызвать одну функцию:

```
scores.sort()
```

Есть много способов настроить порядок сортировки, но это более сложная тема.

Если вас заинтересовала тема сортировки, советуем изучить различные алгоритмы, их преимущества и недостатки. Каждый алгоритм оценивается по скорости выполнения и эффективности использования памяти. Среди наиболее известных алгоритмов — сортировка вставками, сортировка слиянием, быстрая сортировка и многие другие. В Python реализован гибридный алгоритм Timsort, сочетающий сортировку вставками и сортировку слиянием. Обо всем этом можно прочитать в Википедии (`https://ru.wikipedia.org/wiki/Алгоритм_сортировки`).

Что?! Встроенная сортировка? Не мог сказать нам об этом раньше?!

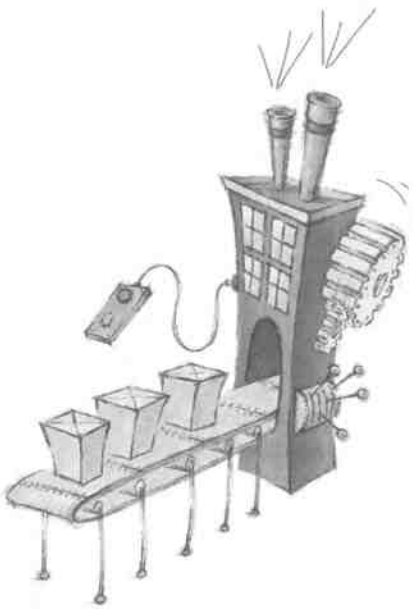
На самый конец у нас припасена небольшая головоломка.

Супержелезяка²

Супержелезяка — это хитроумная штуковина, которая стучит, гремит и даже гудит. Ей не важно, что перемалывать: хоть списки, хоть строки. Но как она работает? А вы сможете разобраться в ее коде?

```
characters = 'тако'

output = ''
length = len(characters)
i = 0
while (i < length):
    output = output + characters[i]
    i = i + 1

length = length * -1
i = -2

while (i >= length):
    output = output + characters[i]
    i = i - 1

print(output)
```

Все, что мы поменяли,— это список, который теперь стал строкой. Но код работает, как и раньше. Почему?



Попробуйте альтернативные варианты строки символов:

characters = 'amanaplanac'

или

characters = 'wasitar'

Почему код работает и со списками, и со строками? На этот непростой вопрос мы частично дадим ответ в главе 6 (и продолжим в последующих главах)

Возьмите карандаш
Решение

Теперь, когда вы получили представление о пузырьковом методе сортировки, давайте применим его для выполнения практического задания, чтобы закрепить приобретенные навыки. Отсортируйте приведенный ниже список так, как это делалось на двух предыдущих страницах.

```
['клубничный', 'банановый', 'ананасовый', 'ягодный']
```

Вот наш список. Не пугайтесь строк, просто сравнивайте их в алфавитном порядке

Проход 1

```
['клубничный', 'банановый', 'ананасовый', 'ягодный']
```
Переставляем

```
['банановый', 'клубничный', 'ананасовый', 'ягодный']
```
Переставляем

```
['банановый', 'ананасовый', 'клубничный', 'ягодный']
```
Не меняем

```
['банановый', 'ананасовый', 'клубничный', 'ягодный']
```

Сравниваем каждый элемент со следующим, проходя по списку. Делаем перестановку, если первое значение больше второго

На первом проходе были перестановки, поэтому нужен второй проход

Проход 2

```
['банановый', 'ананасовый', 'клубничный', 'ягодный']
```
Переставляем

```
['ананасовый', 'банановый', 'клубничный', 'ягодный']
```
Не меняем

```
['ананасовый', 'банановый', 'клубничный', 'ягодный']
```
Не меняем

```
['ананасовый', 'банановый', 'клубничный', 'ягодный']
```

На втором проходе были перестановки, поэтому нужен третий проход

Проход 3

```
['ананасовый', 'банановый', 'клубничный', 'ягодный']
```
Не меняем

```
['ананасовый', 'банановый', 'клубничный', 'ягодный']
```
Не меняем

```
['ананасовый', 'банановый', 'клубничный', 'ягодный']
```
Не меняем

```
['ананасовый', 'банановый', 'клубничный', 'ягодный']
```

На третьем проходе не было перестановок, так что мы закончили, и список отсортирован

Учимся понимать

Решение

Возьмите на себя роль интерпретатора Python и мысленно выполните приведенные ниже фрагменты кода, попытавшись предугадать их результаты. Завершив упражнение, сверьтесь с ответом в конце главы.

```
for i in range(0, 4):
    for j in range(0, 4):
        print(i * j)
```

Оболочка Python

```
0
0
0
0
0
1
2
3
0
2
4
6
0
3
6
9
>>>
```

```
for word in ['як', 'кот', 'пума', 'ягуар', 'собака']:
    for i in range(2, 7):
        letters = len(word)
        if (letters % i) == 0:
            print(i, word)
```

Оболочка Python

```
2 як
3 кот
2 пума
4 пума
5 ягуар
2 собака
3 собака
6 собака
>>>
```

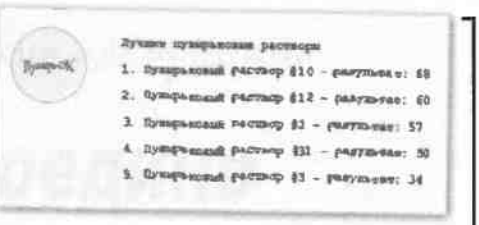
```
full = False

donations = []
full_load = 45

toys = ['робот', 'кукла', 'мяч', 'волчок']

while not full:
    for toy in toys:
        donations.append(toy)
        size = len(donations)
        if (size >= full_load):
            full = True

print('Полон', len(donations), 'игрушек')
print(donations)
```

Оболочка Python

```
Полон 48 игрушек
['робот', 'кукла', 'мяч', 'волчок', 'робот', 'кукла',
'мяч', 'волчок', 'робот', 'кукла', 'мяч', 'волчок',
'робот', 'кукла', 'мяч', 'волчок', 'робот', 'кукла',
'мяч', 'волчок', 'робот', 'кукла', 'мяч', 'волчок',
'робот', 'кукла', 'мяч', 'волчок', 'робот', 'кукла',
'мяч', 'волчок', 'робот', 'кукла', 'мяч', 'волчок',
'робот', 'кукла', 'мяч', 'волчок', 'робот', 'кукла', 'мяч',
'волчок', 'робот', 'кукла', 'мяч', 'волчок']
>>>
```

Возьмите карандаш
Решение

Теперь в программе создаются сразу два списка: один хранит результаты исследований, упорядоченные по убыванию, а второй — соответствующие им номера растворов. Напишите код вывода отчета на основе этих двух списков.



Выводим заголовок

```
print('Лучшие пузырьковые растворы')
for i in range(0, 5):
    print(str(i+1) + '.',
          'Пузырьковый раствор #' + str(solution_numbers[i]),
          '- результат:', scores[i])
```

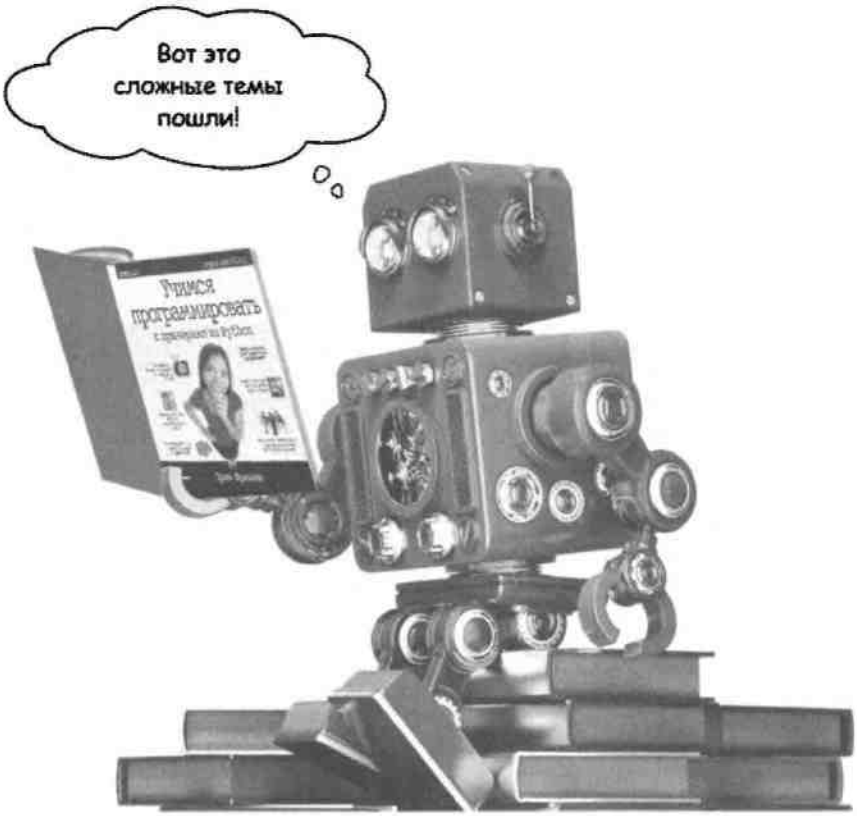
Выполняем пять итераций для вывода пяти наибольших результатов

В каждой строке вывода мы отображаем итоговую позицию раствора (значение i+1), номер раствора из списка solution_numbers и результат из списка scores

Сведем все воедино

Вы уже многое знаете. Пора задействовать знания по максимуму. В этой главе мы сведем все воедино и начнем писать **сложный код**. В то же время мы продолжим пополнять арсенал знаний и навыков программирования. В частности, будет показано, как написать код, который загружает текст, обрабатывает его и выполняет **анализ текстовых данных.** Вы также узнаете, что такое **эвристический алгоритм** и как его реализовать. Приготовьтесь — вас ждет по-настоящему сложная и насыщенная глава.

К концу главы вы поймете, как много узнали о программировании



Добро пожаловать в науку о данных

Слышали про науку о данных? Она занимается вопросами анализа данных и извлечения из них знаний. Этим мы сейчас и займемся. О каких данных идет речь? Любой текст: новости, публикации в блогах, книги и т.п. Мы будем определять, насколько *читабельным* является текст. Другими словами, на кого он ориентирован: пятиклассника или выпускника вуза.

Для этого нам придется очень глубоко погрузиться в анализ текста. Нам предстоит исследовать:

каждое предложение...

каждое слово...

каждый слог...

не говоря уже о каждом символе!

В результате анализа мы получим оценку, которая определит уровень текста: от пятиклассника до выпускника вуза. Что ж, давайте узнаем, как все это работает.

Как вычисляется индекс удобочитаемости

К счастью, формула для оценки удобочитаемости текста давно известна. Ее разработали в американской армии, где много лет изучали вопрос, насколько понятными являются руководства по эксплуатации военной техники. Вот как это работает: берем анализируемый текст и применяем к нему приведенную ниже формулу для вычисления *индекса удобочитаемости*. Сама формула выглядит так.

Детальнее об этом рассказано в Википедии: `https://ru.wikipedia.org/wiki/Индекс_удобочитаемости`

Формула была открыта в 1948 году Рудольфом Флешем, ученым из Колумбийского университета

В формуле используются три переменные, которые нам предстоит вычислить

Мы начнем с общего числа слов в тексте (это число встречается в формуле дважды)

Нам также нужно вычислить общее число слогов в тексте

$$206{,}835 - 1{,}015\left(\frac{\text{всего слов}}{\text{предложений}}\right) - 84{,}6\left(\frac{\text{всего слогов}}{\text{всего слов}}\right)$$

И еще необходимо определить общее число предложений

После подстановки значений переменных формула будет содержать только привычные операции умножения, деления и вычитания чисел с плавающей точкой

Если вам интересно, откуда взялись числа наподобие 206,835, то могу лишь сказать, что за ними стоят годы исследований

Вычислив индекс удобочитаемости с помощью приведенной выше формулы, можно определить уровень текста по следующей таблице.

Предположим, вы пишете рекламный текст. Какой уровень вы бы предпочли?

Индекс	Уровень	Примечания
100–90	5-й класс	Очень легко читается, понятен 11-летнему школьнику
90–80	6-й класс	Легко читается, типичный разговорный язык
80–70	7-й класс	Относительно легко читается
70–60	8-9-й классы	Литературный язык, понятен школьникам 13–15 лет
60–50	10-11-й классы	Относительно тяжело читается
50–30	Студент	Тяжело читается
30–0	Выпускник вуза	Очень тяжело читается, понятен в основном выпускникам вузов

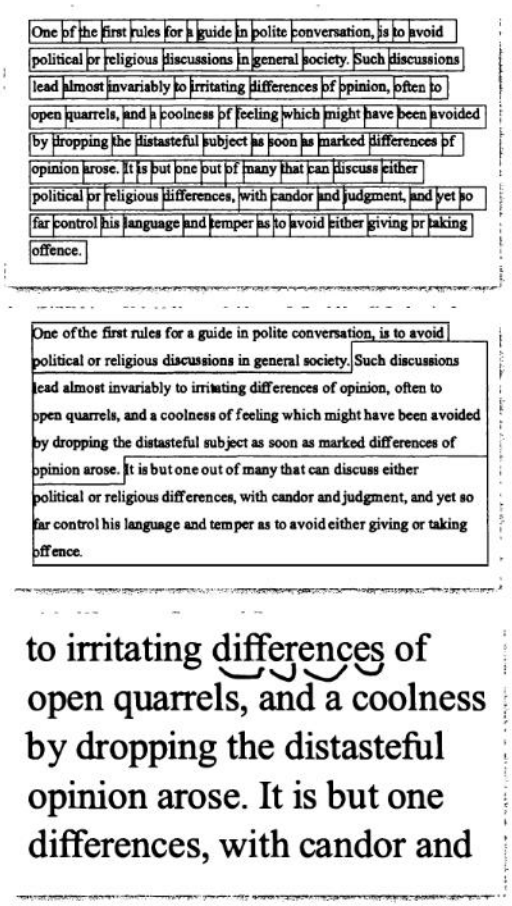
Чем выше индекс, тем проще читать текст

Исходную таблицу можно посмотреть в Википедии

План действий

На первый взгляд, задача проста: нам нужно вычислить не самую сложную формулу. Но при более детальном анализе становится понятно, что основная трудность заключается в вычислении компонентов формулы. Вот нам нужно получить.

1. Общее количество **слов** в тексте. Для этого текст нужно разбить на отдельные слова и подсчитать их.
2. Общее количество **предложений** в тексте. Для этого текст нужно разбить на отдельные предложения и подсчитать их.
3. Общее количество **слогов** в тексте. Для этого каждое слово нужно разбить на отдельные слоги и подсчитать их количество во всем тексте.



Сила мысли

Попробуйте составить алгоритм подсчета слогов в произвольном слове. Предполагается, что в вашем распоряжении нет большого словаря. Напишите алгоритм в виде псевдокода.

Это сложное задание, так что придется напрячь мозги. Запишите решение, к которому удалось прийти в течение пяти минут.

Общий псевдокод

Сначала мы напишем максимально общий псевдокод решения задачи. Детали прояснятся позже. Здесь нет никакой сложной программной логики. Нам просто нужно получить числа, подставляемые в формулу вычисления индекса удобочитаемости. Внимательно изучите комментарии к псевдокоду. Как известно, псевдокод бывает разной степени детализации. В данном случае использован более формальный подход, и поэтому псевдокод напоминает код реальной программы.

Начнем с того, что поместим этот код в функцию

В функцию передается текст книги или статьи, и нам нужно понять, как это сделать

Мы уже знаем о нескольких важных переменных, которые нужны для формулы. Создадим для них локальные переменные. Мы также создадим переменную `score` *для итогового индекса*

DEFINE ***функция*** compute_readability(text):

УСТАНОВКА: здесь мы объявляем все локальные переменные, которые будут использоваться в функции

DECLARE ***переменная*** total_words и присвоить ей 0
DECLARE ***переменная*** total_sentences и присвоить ей 0
DECLARE ***переменная*** total_syllables и присвоить ей 0
DECLARE ***переменная*** score и присвоить ей 0

Каждая переменная вычисляется с помощью отдельной функции

АНАЛИЗ: вычисляем все необходимые значения по отдельности

ASSIGN ***переменная*** total_words результат вызова функции **count_words**(text)
ASSIGN ***переменная*** total_sentences результат вызова функции **count_sentences**(text)
ASSIGN ***переменная*** total_syllables результат вызова функции **count_syllables**(text)

ФОРМУЛА: все значения вычислены, теперь можно воспользоваться формулой для получения общей оценки

ASSIGN ***переменная*** score значение
206,835 – **1,015** * (total_words / total_sentences) – **84,6** * (total_syllables / total_words)

Имея все значения, можно вычислить итоговый индекс...

РЕЗУЛЬТАТЫ: вычислив индекс, сопоставляем его с уровнем текста по таблице, которая приводилась ранее

IF score >= 90:
    PRINT 'Уровень 5-го класса'
ELIF score >= 80:
    PRINT 'Уровень 6-го класса'
ELIF score >= 70:
    PRINT 'Уровень 7-го класса'
ELIF score >= 60:
    PRINT 'Уровень 8–9-го классов'
ELIF score >= 50:
    PRINT 'Уровень 10–11-го классов'
ELIF score >= 30:
    PRINT 'Уровень студента вуза'
ELSE:
    PRINT 'Уровень выпускника вуза'

...и по нему определить уровень текста

Определение уровня текста выглядит полезным фрагментом кода, поэтому перенесем его в отдельную функцию, чтобы им можно было пользоваться повторно

Это большая конструкция `if/elif/else`*, в которой пользователю сообщается уровень текста на основе вычисленного индекса удобочитаемости*

Этот код лучше перенести в отдельную функцию

Нужен текст для анализа

Прежде чем заняться написанием кода, нам нужно подобрать текст для анализа. По большому счету, для наших целей подойдет любой текстовый документ: статья, публикация в блоге, книга и т.п. Но лучше всего, если мы с вами будем работать с одним и тем же документом, чтобы наши результаты совпадали. Осталось определить, что это будет.

Почему бы не проверить самих себя? Мы используем первые несколько абзацев из английского варианта главы 1. Они хранятся в файле *ch6/text.txt*.

Ради интереса мы возьмем фрагмент из главы 1 английского варианта книги

Инструкции о том, как загрузить файлы примеров к книге, были даны во введении. Поскольку мы уже на главе 6, вы давно должны были это сделать

Передача многострочного текста в Python

Файл *ch6/text.txt* представляет собой большой текстовый документ, который нужно как-то загрузить в программу. Вы уже знаете, как добавить строку текста:

```
text = 'The first thing that stands between you'
```

То же самое можно проделать с текстом, хранящимся в файле. По общепринятому соглашению, если текст занимает несколько строк, его следует заключить в тройные кавычки.

Пока ничего не вводите в редакторе IDLE. Мы сделаем это на следующей странице

Начните строку с трех кавычек (подойдут как одинарные, так и двойные кавычки)

Далее введите всю строку, включая символы абзаца

```
text = """The first thing that stands between you and writing your first, real,
piece of code, is learning the skill of breaking problems down into
achievable little actions that a computer can do for you."""
```

Убедитесь в том, что в тексте внутри строки нет трех идущих подряд кавычек, что, конечно же, маловероятно

Завершите строку тремя кавычками

Созданную подобным образом переменную можно использовать в коде Python как самую обычную строку. Теперь сохраним в виде строки текст нескольких первых абзацев главы 1.

Помните: файл Python называется модулем

Текст, хранящийся в файле *ch6/text.txt*, необходимо превратить в файл Python. Для этого создайте в IDLE пустой файл и вставьте в него приведенный ниже код (текст, помещаемый в переменную **text**, берется из файла *text.txt*). Назовите полученный файл *ch1text.py*.

Далее запустите файл на выполнение, и вы должны увидеть на экране сохраненный текст.

Вначале укажите имя строковой переменной, после нее поставьте знак равенства, а за ним — три кавычки. Далее вставьте текст из файла text.txt

```
text = """The first thing that stands between you and writing your first,
real, piece of code, is learning the skill of breaking problems down into
achievable little actions that a computer can do for you. Of course,
you and the computer will also need to be speaking a common language,
but we'll get to that topic in just a bit.

Now breaking problems down into a number of steps may sound a new
skill, but its actually something you do every day. Let's look at an
example, a simple one: say you wanted to break the activity of fishing
down into a simple set of instructions that you could hand to a robot,
who would do your fishing for you. Here's our first attempt to do that,
check it out:

.
.
.

You're going to find these simple statements or instructions are the
first key to coding, in fact every App or software program you've ever
used has been nothing more than a (sometimes large) set of simple
instructions to the computer that tell it what to do."""

print(text)
```

Сбережем несколько деревьев и не будем приводить содержимое файла text.txt целиком

Не забудьте поставить три кавычки в конце

Выведем текст на экран, так как нужно убедиться, что все работает

Создание основной функции

Для начала напишем функцию `compute_readability()`.
В первую очередь объявим необходимые переменные.

```
def compute_readability(text):
    total_words = 0
    total_sentences = 0
    total_syllables = 0
    score = 0
```

Создаем функцию, которая получает текст в качестве аргумента

Объявляем в ней четыре локальные переменные для хранения важных значений внутри функции

Сохраните этот код в файле *analyze.py*.

Далее необходимо придумать, как передать функции `compute_readability()` текст, сохраненный в файле *ch1text.py*. В конце концов, это ведь не строка, а отдельный файл. Как вы помните, файлы с расширением *.py* являются модулями Python, и ранее нам уже доводилось импортировать их с помощью инструкции `import`. Используем ее для импорта модуля *ch1text.py* в файл *analyze.py*. Благодаря этому мы получим доступ к переменным и функциям модуля, имена которых следует предварять его названием. Вот как это делается.

Не забывайте, что модуль — это просто файл с расширением .py, содержащий программный код Python

Модулям посвящена следующая глава

Используем инструкцию `import` для подключения файла ch1text.py

```
import ch1text

def compute_readability(text):
    total_words = 0
    total_sentences = 0
    total_syllables = 0
    score = 0

compute_readability(ch1text.text)
```

Вспомните, что мы определили функцию `compute_readability()` с параметром `text` и несколькими локальными переменными

Вызываем функцию `compute_readability()` и передаем ей строку `text` из файла `ch1text`

Для доступа к переменной `text` необходимо указать перед ней имя модуля `ch1text`

Мы вызываем эту функцию и передаем ей переменную `text` из модуля `ch1text` (другими словами, из файла ch1text.py)

Тест-драйв

Следует убедиться, что введенная нами программа работает корректно. Для этого переместите вызов функции `print()` из модуля *ch1text.py* в файл *analyze.py* и включите его в функцию `compute_readablility()`. Результат работы программы *analyze.py* должен быть таким же, как и при тестовом запуске файла *ch1text.py*.

```
import ch1text

def compute_readability(text):
    total_words = 0
    total_sentences = 0
    total_syllables = 0
    score = 0

    print(text)

compute_readability(ch1text.text)
```

Добавьте эту строку в свою функцию `compute_readability()`

Оболочка Python

```
into pond", or "pull in the fish." But also notice that other
instructions are a bit different because they depend on a condition,
like "is the bobber above or below water?". Instructions might also
direct the flow of the recipe, like "if you haven't finished fishing,
then cycle back to the beginning and put another worm on the hook."
Or, how about a condition for stopping, as in "if you're done" then go
home.

You're going to find these simple statements or instructions are the
first key to coding, in fact every App or software program you've ever
used has been nothing more than a (sometimes large) set of simple
instructions to the computer that tell it what to do.
>>>
```

Вы должны увидеть знакомый текст в окне оболочки

НЕ ЗАБУДЬТЕ удалить вызов `print()` из файла *ch1text.py*

Подсчет числа слов в тексте

Согласно псевдокоду первое, что необходимо подсчитать при вычислении индекса удобочитаемости, — общее количество слов в тексте. Для этого нам придется немного расширить имеющийся инструментарий. Вы уже знаете, как *объединять* строки путем конкатенации, но как *разбить строку* на слова? В Python у строк имеется встроенная функция `split()`, которая разбивает их на слова, сохраняемые в отдельном списке.

Рассмотрим, как работает функция `split()`.

Похоже, что функция `split()` хорошо справляется с разбивкой текста на слова

Оболочка Python

```
['I', 'heard', 'you', 'on',
'the', 'wireless', 'back',
'in', 'fifty', 'two']
>>>
```

Список содержит все слова, извлеченные из оригинальной строки

1. Возьмем любую строку с текстом и пробелами

```
lyrics = 'I heard you on the wireless back in fifty two'
words = lyrics.split()
print(words)
```

2. Вызовем для нее функцию `split()`

3. Используя пробельные символы (пробелы, табуляции и символы новой строки) в качестве разделителей, функция `split()` разбивает строку на слова, добавляя каждое из них в список

4. Выведем полученный список, чтобы увидеть результаты работы функции `split()`

Офисный диалог

Федор. Я уже запутался: мы хотим подсчитать число слов в тексте, но сначала разбиваем текст на слова?

Игорь. Все происходит в два этапа. Сначала мы загружаем все слова в список, а затем определяем длину списка.

Федор. С этапом подсчета слов все понятно, но как функция `split()` разбивает текст на слова?

Юлия. Она находит в тексте разделители — пробельные символы — и выделяет то, что между ними, как отдельные слова.

Федор. Понятно, функция находит пробелы, символы табуляции и новой строки и считает их границами слов.

Юлия. Именно так!

Игорь. А после того как мы получили список слов, подсчитать их количество уже не составляет труда.

Федор. Верно, у нас ведь для этого есть функция `len()`.

Игорь. О ней я и подумал.

Юлия. Ладно, план такой: с помощью функции `split()` разбиваем текст на слова, сохраняем их в отдельном списке, после чего вызываем функцию `len()` для подсчета количества элементов в нем.

Федор. Хороший план!

Не бойтесь задавать вопросы

В: Итак, функция `split()` получает строку и разбивает ее на отдельные слова?

О: Почти. Функция `split()` разбивает строку на подстроки, используя пробельные символы в качестве *разделителей*. Другими словами, всякий раз, когда в тексте встречается разделитель, такой как пробел, табуляция или символ новой строки, функция `split()` считает его концом подстроки. Следует учитывать, что получаемые подстроки не всегда оказываются словами: это могут быть даты, числа и т.п.

В: Что если элементы в строке разделены не пробелами, а, скажем, запятыми?

О: Функция `split()` позволяет задать набор символов, используемых в качестве разделителей. Среди них может быть и запятая. В то же время функция `split()` не настолько гибкая, чтобы сочетать, к примеру, пробелы и запятые. Детали читайте в документации Python. Для поиска слов в тексте можно применять и более сложные инструменты, например регулярные выражения, которые рассматриваются в приложении.

В: Вопрос по синтаксису: а что означает вызов `lyrics.split()`?

О: Пока что вам достаточно знать, что у типов данных в Python есть встроенные функции, которые можно вызывать подобным образом. В частности, строковый тип данных поддерживает функцию `split()`. Таким образом, вызов `lyrics.split()` означает, что функция `split()` вызывается для строки `lyrics`. Подробнее о данном синтаксисе мы поговорим в следующих главах.

Возьмите карандаш

Теперь, когда вам известно назначение функции `split()`, вернемся к функции `compute_readability()`. Согласно псевдокоду нам следует написать функцию `count_words()`, но, как выясняется, с этой задачей прекрасно справляется встроенная функция `split()`. Таким образом, функция `count_words()` нам не нужна. Закончите приведенный ниже код и сверьтесь с решением, приведенным в конце главы. Не забудьте протестировать программу.

```
import ch1text

def compute_readability(text):
    total_words = 0
    total_sentences = 0
    total_syllables = 0
    score = 0

    words = text.split()
    total_words = ________________

    print(words)
    print(total_words, 'слов')
    print(text)

compute_readability(ch1text.text)
```

Разобьем наш текст на слова

Подсказка: Федор, Игорь и Юля уже поняли, как это сделать

Подсчет количества слов занимает одну строку кода — заполните ее

Выведем все слова и счетчик слов; необходимо также удалить старый вызов `print()`

Все верно! По умолчанию функция `split()` разбивает текст по пробельным символам. Несмотря на то что в качестве аргумента ей можно передать другой разделитель, он не сочетается с пробелом. Таким образом, список разделителей не может выглядеть как "пробел, точка, запятая, двоеточие, точка с запятой, восклицательный знак и вопросительный знак". Как следствие, отдельные слова в списке `words` будут заканчиваться знаками препинания.

В нашем случае это допустимо, так как не влияет на подсчет слов. Однако в других задачах подобное поведение будет нежелательным, и вы вскоре узнаете, как его можно исправить.

Оболочка Python

```
['The', 'first', 'thing', 'that', 'stands', 'between', 'you', 'and', 'writing', 'your',
'first,', 'real,', 'piece', 'of', 'code,', 'is', 'learning', 'the', 'skill', 'of', 'breaking',
'problems', 'down', 'into', 'achievable', 'little', 'actions', 'that', 'a', 'computer',
'can', 'do', 'for', 'you.', 'Of', 'course,', 'you', 'and', 'the', 'computer', 'will', 'also',
'need', 'to', 'be', 'speaking', 'a', 'common', 'language,', 'but', "we'll", 'get', 'to',
'that', 'topic', 'in', 'just', 'a', 'bit.', 'Now', 'breaking', 'problems', 'down', 'into',
'a', 'number', 'of', 'steps', 'may', 'sound', 'a', 'new', 'skill,', 'but', 'its', 'actually',
'something', 'you', 'do', 'every', 'day.', 'Let's', 'look', 'at', 'an', 'example,', 'a',
'simple', 'one:', 'say', 'you', 'wanted', 'to', 'break', 'the', 'activity', 'of', 'fishing',
'down', 'into', 'a', 'simple', 'set', 'of', 'instructions', 'that', 'you', 'could', 'hand',
'to', 'a', 'robot,', 'who', 'would', 'do', 'your', 'fishing', 'for', 'you.', 'Here's', 'our',
'first', 'attempt', 'to', 'do', 'that,', 'check', 'it', 'out:', 'you', 'can', 'think',
'of', 'these', 'statements', 'as', 'a', 'nice', 'recipe', 'for', 'fishing.', 'Like', 'any',
'recipe,', 'this', 'one', 'provides', 'a', 'set', 'of', 'steps,', 'that', 'when', 'followed',
'in', 'order,', 'will', 'produce', 'some', 'result', 'or', 'outcome', 'in', 'our', 'case,',
'hopefully,', 'catching', 'some', 'fish.', 'Notice', 'that', 'most', 'steps', 'consists',
'of', 'simple', 'instruction,', 'like', 'cast', 'line', 'into', 'pond', 'or', '"pull',
'in', 'the', 'fish."', 'But', 'also', 'notice', 'that', 'other', 'instructions', 'are',
'a', 'bit', 'different', 'because', 'they', 'depend', 'on', 'a', 'condition,', 'like',
'"is', 'the', 'bobber', 'above', 'or', 'below', 'water?"', 'Instructions', 'might', 'also',
'direct', 'the', 'flow', 'of', 'the', 'recipe,', 'like', '"if', 'you', 'haven't', 'finished',
'fishing,', 'then', 'cycle', 'back', 'to', 'the', 'beginning', 'and', 'put', 'another',
'worm', 'on', 'the', 'hook."', 'Or,', 'how', 'about', 'a', 'condition', 'for', 'stopping,',
'as', 'in', '"if', 'you're', 'done"', 'then', 'go', 'home.', 'You're', 'going', 'to', 'find',
'these', 'simple', 'statements', 'or', 'instructions', 'are', 'the', 'first', 'key', 'to',
'coding,', 'in', 'fact', 'every', 'App', 'or', 'software', 'program', 'you've', 'ever',
'used', 'has', 'been', 'nothing', 'more', 'than', 'a', '(sometimes', 'large)', 'set', 'of',
'simple', 'instructions', 'to', 'the', 'computer', 'that', 'tell', 'it', 'what', 'to', 'do.']
300 слов
>>>
```

Видите, о чем идет речь? При таком способе разбивки некоторые слова завершаются точками, запятыми и кавычками. Также могут встретиться вопросительный и восклицательный знаки, двоеточие и точка с запятой

СИЛА МЫСЛИ

Если бы перед вами стояла задача избавиться от лишних знаков препинания в словах, то как бы вы поступили?

Даже если вы не представляете, как написать такой код, предложите идею решения

Подсчет общего числа предложений

Согласно псевдокоду нашим следующим шагом будет подсчет количества предложений в тексте. Было бы замечательно иметь в своем распоряжении встроенную функцию для решения такой задачи, но, к сожалению, в Python она отсутствует. Нам нужно придумать, как это сделать самостоятельно.

Есть идея: что, если считать знаки препинания, которыми предложение может заканчиваться (точка, вопросительный и восклицательный знаки)? Выяснив число таких знаков в тексте, мы определим примерное количество предложений в нем. Этот способ не идеален, ведь автор может применять нестандартную пунктуацию, но соответствие будет достаточно близким. Кроме того, нам не всегда нужен абсолютно точный подсчет, особенно если он требует значительных усилий.

Как же нам определить количество знаков препинания, которыми может оканчиваться предложение ('.', '?' и '!')? Как вариант — просмотреть все символы текста один за другим, увеличивая значение счетчика каждый раз при нахождении одного из указанных знаков препинания. Мы, правда, не умеем просматривать все символы строки в цикле. Или умеем?

Помните, мы говорили о том, что для просмотра элементов последовательности лучше всего применять цикл `for`? А ведь строка — не что иное, как последовательность символов. Таким образом, для перебора всех символов строки нам понадобится цикл `for`. Приведем пример.

Берем любую строку Python...

```
lyrics = 'I heard you on the wireless back in fifty two'

for char in lyrics:
    print(char)
```

...и проходим по каждому символу в строке

На каждом шаге цикла в переменную `char` записывается очередной символ строки

Отобразим символ, чтобы понять, как все это работает

Оболочка Python

```
I

h
e
a
r
d

y
o
u

o
n

t
h
e

w
i
r
e
l
e
s
s

b
a
c
k

i
n

f
i
f
t
y

t
w
o
>>>
```

Функция count_sentences()

Узнав, как пройти строку символ за символом, давайте создадим каркас функции `count_sentences()`, прежде чем писать код для подсчета знаков препинания.

Как и в псевдокоде, ожидаем, что в функцию передается текст

```
def count_sentences(text):
    count = 0

    for char in text:


    return count
```

Это локальная переменная count

Здесь мы проходим по каждому символу в тексте

Необходимо понять, каким должен быть код в цикле. Является ли символ концевым (завершающим предложение)? Если да, увеличиваем счетчик

Наконец, возвращаем значение счетчика в качестве результата работы функции

По-серьезному

Каркасный код определяет общую структуру программы, но не содержит деталей. Это промежуточный этап между псевдокодом и готовой программой. Создавая каркас, вы получаете представление о том, как будет выглядеть программа, но избегаете низкоуровневого кодирования.

Возьмите карандаш

Напишите код, который проверяет, является ли символ знаком конца предложения (точка, вопросительный или восклицательный знаки), и, если это так, увеличивает на единицу значение переменной `count`.

```
def count_sentences(text):
    count = 0

    for char in text:



    return count
```

Добавьте сюда свой код

Настало время проверить работу программы. Соберите весь код вместе, добавьте его в файл *analyze.py* и выполните программу.

Напомим: мы должны определить функции, прежде чем обращаться к ним. С учетом этого подумайте, где можно определить функцию `count_sentences()`, а где нельзя

```
import ch1text

def count_sentences(text):
    count = 0

    for char in text:
        if char == '.' or char == '?' or char == '!':
            count = count + 1

    return count

def compute_readability(text):
    total_words = 0
    total_sentences = 0
    total_syllables = 0
    score = 0

    words = text.split()
    total_words = len(words)
    total_sentences = count_sentences(text)

    print(words)
    print(total_words, 'слов)
    print(total_sentences, 'предложений')

compute_readability(ch1text.text)
```

Это наша новая функция для подсчета числа предложений

Убедитесь в том, что вы поменяли это с момента последнего тест-драйва

Вызываем нашу новую функцию, передавая ей текст

Добавляем инструкцию вывода, чтобы узнать число предложений

Убедитесь в том, что вы поменяли это с момента последнего тест-драйва

Вот что мы получили

```
Оболочка Python
300 слов
12 предложений
>>>
```

Совершенно верно!

Несмотря на то что код проверки знаков препинания работает правильно, есть более компактный способ проверить вхождение символа в заданный набор. Он основан на использовании булевого оператора in, с которым вы еще не знакомы. Он позволяет проверить, содержится ли значение в заданной последовательности. Помните пример с напитками из главы 4? Давайте проверим, содержится ли конкретный напиток в нашем списке.

Вот наша последовательность в виде списка

```
smoothies = ['кокосовый', 'клубничный', 'банановый', 'ананасовый', 'ягодный']
if 'кокосовый' in smoothies:
    print('Есть кокосовый!')
else:
    print('Увы, нет')
```

Проверяем, содержится ли вариант 'кокосовый' в списке smothies

Вот что мы получаем

```
Оболочка Python
Есть кокосовый!
>>>
```

Как было показано выше, строка — разновидность последовательности, а значит, для поиска нужных символов в ней тоже можно применять оператор `in`. Приведем пример.

Помните программу "Супержелезяка" из главы 4? В ней мы тоже перебирали символы строки в цикле.

Вот наша последовательность в виде строки

```
lyrics = 'I heard you on the wireless back in fifty two'

if 'wireless' in lyrics:
    print('Есть wireless!')
else:
    print('Увы, нет.')
```

Проверяем, содержится ли слово 'wireless' в строке `lyrics`

Оболочка Python

```
Есть wireless!
>>>
```

Возьмите карандаш

Попробуйте переписать функцию `count_sentences()`, сделав ее проще (и понятнее) с помощью оператора `in`. В приведенном ниже варианте функции мы удалили код сравнения со знаками препинания. Вместо этого добавлена локальная переменная `terminals`, в которой перечислены все концевые знаки препинания. Закончите инструкцию `if` так, чтобы она проверяла, является ли текущий символ концевым, используя оператор `in`.

```
def count_sentences(text):
    count = 0

    terminals = '.?!'

    for char in text:

        if ____________________________
            count = count + 1

    return count
```

Добавьте код проверки, включающий оператор `in`

Тест-драйв

Прежде чем переходить к следующему этапу, обновите код программы. Изменения придется вносить только в функцию `count_sentences()`, приведенную ниже. После запуска программы вы должны получить такой же результат, как и на предыдущем тест-драйве.

```
def count_sentences(text):
    count = 0

    terminals = '.?!'
    for char in text:
        if char in terminals:
        if char == '.' or char == '?' or char == '!':
            count = count + 1

    return count
```

Добавьте эти две строки кода

И удалите старую инструкцию сравнения

Результат по-прежнему должен быть таким

```
Оболочка Python
300 слов
12 предложений
>>>
```

Во многих языках программирования — да.

Но не в Python. Вопрос правильный, поскольку в других языках программирования символы считаются отдельным типом данных. Однако в Python они трактуются как строки. Например, символ `'A'` – это строка единичной длины, содержащая букву A.

Если строка — это последовательность, то можно ли обращаться к отдельным ее символам по индексам? Например, будет ли выражение **my_string[1]** соответствовать первому символу строки?

Такой синтаксис разрешен.

Только не забывайте, что выражению `my_string[1]` соответствует второй, а не первый символ строки, ведь индексация начинается с 0. К последнему символу строки можно обратиться с помощью выражения `my_string[-1]`. Как будет показано далее, с помощью такого синтаксиса можно получить доступ не только к отдельным символам, но и к подстрокам.

Отлично! А можно ли аналогичным образом поменять содержимое строки? Например, записать `my_string[1] = 'e'`?

Нет!

Такой способ изменения строк недопустим. Разница между строками и списками заключается в том, что списки — это *изменяемые* последовательности, а строки — *неизменяемые*. Другими словами, можно изменить элемент списка, но не отдельный символ строки. Данное правило применяется почти во всех современных языках программирования. Почему? По мере приобретения опыта в программировании вы поймете, что возможность изменения строк делает код ненадежным. Кроме того, это усложняет задачу написания эффективных интерпретаторов. Для "изменения" строки достаточно создать новую, отредактированную строку — это распространенная практика в современных языках программирования.

Подсчет слогов: знакомство с эвристическими алгоритмами

Вы готовы реализовать алгоритм подсчета слогов? Ну тот, который вы попытались составить во врезке "Сила мысли" в начале главы? Согласен, это было слишком сложное задание: не так-то просто написать алгоритм поиска слогов. Да и не факт, что такой алгоритм вообще существует, если только вы не составите огромную базу слов.

Разбивка слова на слоги – нетривиальная задача, особенно в английском языке. Почему, интересно, в слове "walked" один слог, а в слове "loaded" – два? И таких нестыковок в английском языке множество.

Для решения подобных задач применяются *эвристические алгоритмы*, особенность которых в том, что они не позволяют добиться 100%-го результата. С помощью такого алгоритма можно получить хорошее решение, но не обязательно оптимальное.

Какая польза от эвристического алгоритма? И почему нельзя написать алгоритм, обеспечивающий точное решение? По многим причинам. В нашем случае потому, что попросту не существует абсолютно точного способа определения слогов в таком сложном языке, как английский (опять-таки, если не считать вариант с составлением огромной базы слов). В других случаях причина может заключаться в том, что достижение 100%-но точного результата потребует огромных вычислительных ресурсов, из-за чего применение алгоритма становится нецелесообразным. В то же время приблизительный ответ, выдаваемый эвристическим алгоритмом, требует намного меньше ресурсов. Бывает и так, что не все аспекты предметной области известны разработчикам, поэтому лучшее, что они могут сделать, – дать примерное решение.

Но вернемся к нашему заданию. В связи с тем что процесс разбивки слов на слоги слишком нетривиален, нашей целью станет получение грубой оценки числа слогов в тексте. При желании вы всегда сможете улучшить эвристический алгоритм, и мы дадим ряд советов на этот счет.

Сила мысли

Вы получаете второй шанс: предложите идеи, как можно подсчитать число слогов в произвольном слове. Попробуйте сформулировать какие-то общие правила и запишите здесь свои соображения. Одну идею (касательно английского языка) мы вам подскажем.

Если в слове три буквы или меньше, то обычно оно состоит из одного слога. *Придумайте еще несколько правил*

Не понимаю, к чему такие сложности? Почему нельзя просто использовать орфографический словарь, ведь в нем для каждого слова указана разбивка по слогам?

Интересная мысль, но... Навряд ли вы понимаете, что значит загрузить в программу весь словарь английского языка, причем в таком виде, чтобы слова были разбиты по слогам. Это огромный массив данных, который занимает слишком много места и для работы с которым нам потребуется специализированная база данных или какой-то иной поисковый механизм. Не говоря уже о том, что подобные словари стоят немалых денег.

С учетом этого эвристический алгоритм, который позволяет применить всего несколько простых правил для получения точности 80–90% или даже выше, становится вполне разумной альтернативой.

Особенно если учитывать задачи книги

Офисный диалог продолжается...

Игорь. Я заметил, что слова, в которых менее трех букв, состоят всего из одного слога.

Юлия. Боюсь, даже из этого правила могут быть исключения, но их, по крайней мере, очень мало. Я также заметила, что количество слогов в слове примерно соответствует числу гласных в нем.

Игорь. Хорошо, давай возьмем слово "chocolate"...

Юлия. Я не против целой плитки!

Игорь. Это потом. В слове "chocolate" четыре гласные и три слога.

Юлия. Просто в конце слова буква 'e' непроизносимая. Есть и другие проблемы.

Игорь. Например?

Юлия. Возьмем слово "looker". Оно содержит удвоенное "oo". В подобных случаях нужно учитывать только первую гласную.

Игорь. Верно. Считаем первую 'o', а затем 'e', и получаем два слога. К тому же, как ты справедливо заметила, многие слова содержат немую 'e' в конце, как в слове "home".

Юлия. Да, в этом случае учитываем 'o', но не конечную 'e', и получаем один слог. Кстати, буква 'y' в конце слова образует слог, как в слове "proxy". Это тоже нужно учитывать.

Игорь. Если образуется слог, то конечно.

Юлия. Есть еще ряд особых случаев. Например, в слове "walked" один слог, а в слове "loaded" – два.

Игорь. Можно создать список подобных исключений и сверять по нему слова.

Юлия. Это слишком долго. Давай ограничимся общими правилами, а исключениями займемся, если возникнет такая необходимость.

Игорь. План понятен.

Создание эвристического алгоритма

Похоже, у нас есть рабочие идеи для составления эвристического алгоритма подсчета слогов. Подытожим их, прежде чем приступать к реализации.

- ☐ Если слово содержит менее трех букв, считаем, что в нем один слог.
- ☐ В противном случае считаем количество гласных и принимаем его за число слогов.
- ☐ Уточняем предыдущее правило, удаляя последовательные гласные.
- ☐ Исключаем из подсчета конечное непроизносимое 'е'.
- ☐ Буква 'у' в конце слова считается гласной.

Приступим к написанию функции `count_syllables()`, которая должна возвращать число слогов в переданном ей списке слов.

```
def count_syllables(words):
    count = 0

    for word in words:
        word_count = count_syllables_in_word(word)
        count = count + word_count

    return count
```

Вот наша новая функция, которой передается список слов

Используем локальную переменную count для хранения общего числа слогов

Мы пройдем по каждому слову в списке

Вызываем другую функцию, count_syllables_in_word(), которую напишем далее; она возвращает число слогов в отдельном слове

В локальной переменной word_count хранится число слогов текущего слова. Может быть, это не самое удачное название, но syllables_count слишком напоминало бы название функции. В таких местах лучше оставлять комментарии, чтобы потом не было путаницы

Далее добавляем число слогов в текущем слове к общему счетчику

Наконец, возвращаем общее число слогов во всех словах

Далее составим каркас функции `count_syllables_in_word()`.

```
def count_syllables_in_word(word):
    count = 0

    return count
```

Функция count_syllables_in_word() получает в качестве аргумента строку word

Опять используем локальную переменную count для отслеживания общего числа слогов

Остальной код будет находиться здесь

В конце возвращаем счетчик числа слогов

Добавьте новый код в файл *analyze.py* и убедитесь в отсутствии синтаксических ошибок. Программа должна выдавать тот же результат, что и раньше.

```
def count_syllables(words):
    count = 0

    for word in words:
        word_count = count_syllables_in_word(word)
        count = count + word_count

    return count

def count_syllables_in_word(word):
    count = 0

    return count
```

Добавьте этот код в начало файла, сразу после инструкции `import`

Начинаем кодировать эвристический алгоритм

Мы здесь →

- ☐ Если слово содержит менее трех букв, считаем, что в нем один слог
- ☐ В противном случае считаем количество гласных и принимаем его за число слогов
- ☐ Уточняем предыдущее правило, удаляя последовательные гласные
- ☐ Исключаем из подсчета конечное непроизносимое 'e'
- ☐ Буква 'y' в конце слова считается гласной

Согласно первому пункту нам нужно найти все слова, состоящие менее чем из трех букв. Каждое из таких слов содержит всего один слог. Но как определить длину строки? Оказывается, для этого подходит уже известная нам функции `len()`. Ранее мы применяли ее для определения длины списка, но она работает с любыми последовательностями Python, а строка – это последовательность символов.

Мы часто подчеркиваем это в данной главе

```
def count_syllables_in_word(word):
    count = 0

    if len(word) <= 3:
        return 1
    return count
```

Используем функцию `len()`, чтобы определить длину строки `word`. Если длина меньше или равна 3, выполняем блок кода инструкции `if`

Если длина слова не превышает 3, завершаем вычисления и возвращаем число 1

Подсчет гласных

- ☑ Если слово содержит менее трех букв, считаем, что в нем один слог
- ☐ В противном случае считаем количество гласных и принимаем его за число слогов
- ☐ Уточняем предыдущее правило, удаляя последовательные гласные
- ☐ Исключаем из подсчета конечное непроизносимое 'e'
- ☐ Буква 'y' в конце слова считается гласной

Далее нам нужно подсчитать число гласных в каждом слове. Мы поступим примерно так, как и при поиске концевых знаков препинания: мы сформируем строку, содержащую все гласные, и будем использовать оператор `in` для проверки вхождения каждой буквы слова в эту строку. Символы будут проверяться как в нижнем, так и в верхнем регистре, чтобы учитывать не только строчные, но и прописные буквы.

```
def count_syllables_in_word(word):
    count = 0

    if len(word) <= 3:
        return 1

    vowels = "aeiouAEIOU"
    for char in word:
        if char in vowels:
            count = count + 1

    return count
```

Сначала создаем локальную переменную `vowels`, в которой будут храниться все гласные как в нижнем, так и в верхнем регистре

После этого проходим по каждому символу в слове

Используем оператор `in`, чтобы проверить, соответствует ли текущий символ в переменной `char` одному из символов в строке `vowels`

Если текущий символ входит в строку `vowels`, увеличиваем счетчик гласных на единицу

Игнорирование последовательных гласных

- ☑ Если слово содержит менее трех букв, считаем, что в нем один слог
- ☑ В противном случае считаем количество гласных и принимаем его за число слогов
- ☐ Уточняем предыдущее правило, удаляя последовательные гласные
- ☐ Исключаем из подсчета конечное непроизносимое 'e'
- ☐ Буква 'y' в конце слова считается гласной

Но это еще не все, поскольку нам нужно учесть последовательные гласные. Например, в слове "book" необходимо считать только первую букву 'o', игнорируя вторую, и мы получаем один слог. Точно так же в слове "roomful" учитывается только первая 'o', а также 'u', в результате чего мы получаем два слога.

Таким образом, если мы встречаем гласную, то должны игнорировать все последующие гласные до тех пор, пока не встретится согласная. Далее мы повторяем процесс, пока не достигнем конца слова.

Код игнорирования последовательных гласных

Итак, давайте напишем соответствующий код. Вот что мы сделаем: мы используем булеву переменную, которая будет фиксировать, была ли предыдущая буква гласной. Тогда, если текущая буква тоже гласная, ее нужно пропустить при подсчете слогов.

```
def count_syllables_in_word(word):
    count = 0

    if len(word) <= 3:
        return 1

    vowels = "aeiouAEIOU"
    prev_char_was_vowel = False

    for char in word:
        if char in vowels:
            if not prev_char_was_vowel:
                count = count + 1
            prev_char_was_vowel = True
        else:
            prev_char_was_vowel = False

    return count
```

В этом коде выполняется много разных действий. Потратьте время на то, чтобы разобраться в нем и понять, как он работает

Сначала создаем новую локальную переменную `prev_char_was_vowel` *и задаем ее равной* `False`

Функция проходит по каждому символу в слове, как и раньше

Проверяем, входит ли текущий символ в строку `vowels`

Если текущий символ — гласная, а предыдущий символ — нет, увеличиваем счетчик слогов

В любом случае устанавливаем переменную `prev_char_was_vowel` *равной* `True`*, прежде чем идти дальше*

Если текущий символ — не гласная, задаем переменную `prev_char_was_vowel` *равной* `False`*, прежде чем переходить к следующему символу*

Учимся понимать

Изучая приведенную выше функцию `count_syllables_in_word()`, представьте, что вы интерпретатор Python. Функции передается аргумент `"roomful"`. Мысленно пройдите весь цикл, отслеживая изменения локальных переменных. Выполнив задание, сверьтесь с ответом, приведенным в конце главы.

char	prev_char_was_vowel	count
r	False	0
o		
o		
m		
f		
u		
l		

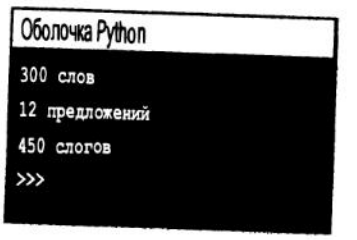
Зафиксируйте значения либо изменения локальных переменных в конце каждой итерации

На данный момент в программе нужно обновить две функции: `count_syllables_in_word()`, в которую добавляется новый код, и `compute_readability()`, из которой вызывается функция `count_syllables()`. Ниже показаны только изменяемые функции, в которых выделен новый код. Внесите необходимые изменения и запустите программу.

```
def count_syllables_in_word(word):
    count = 0

    if len(word) <= 3:
        return 1

    vowels = "aeiouAEIOU"
    prev_char_was_vowel = False

    for char in word:
        if char in vowels:
            if not prev_char_was_vowel:
                count = count + 1
            prev_char_was_vowel = True
        else:
            prev_char_was_vowel = False

    return count
```

```
def compute_readability(text):
    total_words = 0
    total_sentences = 0
    total_syllables = 0
    score = 0

    words = text.split()
    total_words = len(words)
    total_sentences = count_sentences(text)
    total_syllables = count_syllables(words)

    print(total_words, 'слов')
    print(total_sentences, 'предложений')
    print(total_syllables, 'слогов')
```

Убедитесь в том, что получаете то же самое число слов, предложений и слогов, в противном случае тщательно проверьте код и сравните его с кодом на сайте книги



Учет конечной 'e', а также знаков препинания

- ☑ Если слово содержит менее трех букв, считаем, что в нем один слог
- ☑ В противном случае считаем количество гласных и принимаем его за число слогов
- ☑ Уточняем предыдущее правило, удаляя последовательные гласные
- ☐ Исключаем из подсчета конечное непроизносимое 'e'
- ☐ Буква 'y' в конце слова считается гласной

Нам осталось решить две задачи: исключить конечную 'e' и посчитать конечную 'y'. Но, вообще-то, мы забыли еще кое-что: некоторые слова, получаемые при разбивке исходной строки, содержат в конце знаки препинания, которые будут мешать нахождению концевых 'e' и 'y'. Поэтому сначала необходимо удалить знаки препинания.

Просматривая список `words`, вы найдете в нем такие слова, как "first,", "you." и "out:". Нам нужно заменить их на "first", "you" и "out", т.е. научиться извлекать подстроки. В Python они называются *срезами*. Познакомимся с ними поближе.

Вычленяем главное

Поскольку мы упомянули подстроки, давайте разберемся, что же это такое. Подстрока представляет собой строку, которая встречается в другой строке. Предположим, имеется следующая строка:

```
lyrics = 'I heard you on the wireless back in fifty two'
```

Ее подстроками будут `'I'`, `'I heard'`, `'on the wire'`, `'o'` и т.п. В Python имеется удобная возможность выделять подстроки в виде срезов с помощью специального синтаксиса.

Начинаем подстроку с этой позиции

Завершаем подстроку на этой позиции, НЕ включая ее

```
my_substring = lyrics[2:7]
print(my_substring)
```

Здесь ставится двоеточие, а не запятая!

Таким образом, подстрока содержит символы в позициях 2—6, т.е. 'heard'

Оболочка Python

```
heard
>>>
```

Есть и другие способы создания срезов. Если в квадратных скобках опустить первый индекс, то срез будет начинаться с первого элемента.

```
my_substring = lyrics[:6]
print(my_substring)
```

Если первый индекс не указан, значит, считать нужно с начала строки

Оболочка Python

```
I hear
>>>
```

Аналогичным образом, если в квадратных скобках не указать второй индекс, то срез будет заканчиваться последним элементом.

```
my_substring = lyrics[28:]
print(my_substring)
```

Если последний индекс не указан, значит, берем все до конца строки

Оболочка Python

```
back in fifty two
>>>
```

Можно даже использовать отрицательные индексы, как это делалось в списках.

Это означает, что подстрока завершается предпоследним символом строки

```
my_substring = lyrics[28:-1]
print(my_substring)
```

Оболочка Python

```
back in fifty tw
>>>
```

А вот что произойдет, если отрицательный индекс указать первым.

```
my_substring = lyrics[-17:]
print(my_substring)
```

Начать за 17 символов до конца строки и продолжать до конца

Оболочка Python

```
back in fifty two
>>>
```

Срезы применяются не только к строкам!

В Python срезы можно создавать не только для строк, но и для любых последовательностей, таких как списки.

```
smoothies = ['кокосовый', 'клубничный', 'банановый', 'ананасовый', 'ягодный']
```

Вот наш список

А вот несколько примеров извлечения фрагментов списка. Индексация работает так же, как и в случае строк

```
smoothies[2:4]  --дает--> ['банановый', 'ананасовый']

smoothies[:2]   --дает--> ['кокосовый', 'клубничный']

smoothies[3:-1] --дает--> ['ананасовый']
```

Кто что делает?

Мы выполнили ряд операций по созданию срезов, но затем все перемешалось. Не поможете нам расставить все по своим местам? Одно соответствие мы нашли сами.

```
str = 'a man a plan panama'
```

`str[:]`	'man a plan panama'
`str[:2]`	'a man a plan panam'
`str[2:]`	'a '
`str[1:7]`	'a man a plan panama'
`str[3:-1]`	'an a plan panam'
`str[-2:-1]`	'a ma'
`str[0:-1]`	' man a'
`str[0:4]`	'm'

Создание срезов (подстрок)

Благодаря срезам можно из строки "out:" получить строку "out", лишенную конечного знака препинания. По сути, вы уже знаете, как записать такую операцию.

```
process_word = word[0:-1]
```

Получаем подстроку, которая начинается с индекса 0 (начало строки) и доходит до предпоследнего символа строки

Аналогичное выражение можно записать как `[:-1]`

Эта переменная будет содержать новую строку, равную прежней, но без завершающего символа

Осталось понять, в каких случаях из слова нужно удалить последний символ. Это должно происходить, когда последним символом является точка, запятая, точка с запятой, двоеточие, восклицательный или вопросительный знак. Похожую задачу мы уже решали раньше.

```
def count_syllables_in_word(word):
    count = 0

    endings = '.,;!?:'
    last_char = word[-1]

    if last_char in endings:
        processed_word = word[0:-1]
    else:
        processed_word = word

    if len(processed_word) <= 3:
        return 1

    vowels = "aeiouAEIOU"
    prev_char_was_vowel = False

    for char in processed_word:
        if char in vowels:
            if not prev_char_was_vowel:
                count = count + 1
            prev_char_was_vowel = True
        else:
            prev_char_was_vowel = False

    return count
```

Создаем строку, содержащую все концевые знаки препинания

Определяем последний символ текущего слова

Проверяем, встречается ли этот символ в строке `endings`

Если да, записываем в переменную `processed_word` *исходное слово без последнего символа*

В противном случае записываем в переменную `processed_word` *слово целиком*

Далее мы будем использовать переменную `processed_word`*, а не параметр* `word`

Пока не нужно вводить этот код, так как мы его сейчас доработаем

Возьмите карандаш

Теперь переменная `process_word` содержит слово, из которого удален концевой знак препинания. Напишите код для удаления из слова конечной буквы 'e'. Это будет буквально несколько строк кода.

Необходимо добавить в программу код обработки подстрок. Добавляемый код выделен серым фоном. Введите его и запустите программу, чтобы узнать, как изменится число слогов.

```
def count_syllables_in_word(word):
    count = 0

    endings = '.,;!?:'
    last_char = word[-1]


    if last_char in endings:
        processed_word = word[0:-1]
    else:
        processed_word = word

    if len(processed_word) <= 3:
        return 1

    if processed_word[-1] in 'eE':
        processed_word = processed_word[0:-1]


    vowels = "aeiouAEIOU"
    prev_char_was_vowel = False

    for char in processed_word:
        if char in vowels:
            if not prev_char_was_vowel:
                count = count + 1
            prev_char_was_vowel = True
        else:
            prev_char_was_vowel = False

    return count
```

Убедитесь в том, что внесены все изменения, описанные на предыдущей странице

После выявления слов, содержащих три буквы и меньше, вставляем код для удаления конечной непроизносимой 'e'

Обратите внимание на то, как изменилось число слогов!

Оболочка Python

```
300 слов
12 предложений
410 слогов
>>>
```

Завершение эвристического алгоритма

Мы достигли финального этапа эвристического алгоритма. Осталось всего ничего – учесть концевую 'y' при подсчете слогов. Вы уже знаете, как проверить последний символ слова. Если он равен 'y', необходимо увеличить значение локальной переменной `count` на единицу. Сведем все вместе и протестируем финальную версию функции `count_syllables_in_word()`, а заодно и функцию `count_syllables()`.

- ☑ Если слово содержит менее трех букв, считаем, что в нем один слог
- ☑ В противном случае считаем количество гласных и принимаем его за число слогов
- ☑ Уточняем предыдущее правило, удаляя последовательные гласные
- ☑ Исключаем из подсчета конечное непроизносимое 'e'
- ☐ Буква 'y' в конце слова считается гласной

Пора завершить функцию `count_syllables_in_word()`. Выделенный ниже код проверяет, заканчивается ли слово на букву 'y', и если да, то увеличивает счетчик слогов. Протестируйте программу, чтобы узнать, сколько слогов получилось.

```
def count_syllables_in_word(word):
    count = 0

    endings = '.,;!?:'
    last_char = word[-1]


    if last_char in endings:
        processed_word = word[0:-1]
    else:
        processed_word = word

    if len(processed_word) <= 3:
        return 1

    if processed_word[-1] in 'eE':
        processed_word = processed_word[0:-1]


    vowels = "aeiouAEIOU"
    prev_char_was_vowel = False

    for char in processed_word:
        if char in vowels:
            if not prev_char_was_vowel:
                count = count + 1
            prev_char_was_vowel = True
        else:
            prev_char_was_vowel = False

    if processed_word[-1] in 'yY':
        count = count + 1

    return count
```

Проверяем, заканчивается ли слово на 'y' или 'Y', и если да, то увеличиваем счетчик слогов

Похоже, мы нашли несколько слов, заканчивающихся на 'y', поэтому счетчик слогов увеличился

```
Оболочка Python
300 слов
12 предложений
416 слогов
>>>
```

Как видите, мы написали достаточно длинную функцию, что вполне допустимо. Как и абзац текста, функция начинает казаться длинной, когда в процессе изучения ее трудно понять и удержать в голове

Код вычисления индекса удобочитаемости

Конец уже близок, осталось написать код вычисления индекса удобочитаемости по формуле, приведенной в начале главы, и вывести результат на экран. Мы определили все элементы формулы: количество слов, предложений и слогов. Еще раз рассмотрим формулу, согласно которой проводятся расчеты:

$$206{,}835 - 1{,}015\left(\frac{\text{всего слов}}{\text{предложений}}\right) - 84{,}6\left(\frac{\text{всего слогов}}{\text{всего слов}}\right)$$

А вот она же на языке Python:

```
score = (206.835 - 1.015 * (total_words / total_sentences)
                 - 84.6 * (total_syllables / total_words))
```

Помните: формулу можно брать в скобки, если нужно разбить ее на несколько строк

Тест-драйв

Теперь можно вычислить формулу в функции `compute_readability()`. Добавьте в нее выделенный ниже код, включая вызов функции `print()`, и протестируйте работу программы.

```
def compute_readability(text):
    total_words = 0
    total_sentences = 0
    total_syllables = 0
    score = 0

    words = text.split()
    total_words = len(words)
    total_sentences = count_sentences(text)
    total_syllables = count_syllables(words)

    score = (206.835 - 1.015 * (total_words / total_sentences)
                 - 84.6 * (total_syllables / total_words))

    print(total_words, 'слов')
    print(total_sentences, 'предложений')
    print(total_syllables, 'слогов')
    print(score, ' - удобочитаемость')
```

Добавим приведенную ниже формулу

Итак, теперь мы вычислили индекс удобочитаемости!

Добавляем инструкцию вывода, чтобы можно было узнать результат

Оболочка Python

```
300 слов
12 предложений
416 слогов
64.14800000000001 - удобочитаемость
>>>
```

Возьмите карандаш

Мы наконец получили индекс удобочитаемости, и осталось вывести результаты. Напишите функцию `output_results()`, которая получает значение индекса и выводит на экран строку с описанием уровня текста. Повторим фрагмент псевдокода для удобства.

```
IF score >= 90:
        PRINT 'Уровень 5-го класса'
ELIF score >= 80:
        PRINT 'Уровень 6-го класса'
ELIF score >= 70:
        PRINT 'Уровень 7-го класса'
ELIF score >= 60:
        PRINT 'Уровень 8–9-го классов'
ELIF score >= 50:
        PRINT 'Уровень 10–11-го классов'
ELIF score >= 30:
        PRINT 'Уровень студента вуза'
ELSE:
        PRINT 'Уровень выпускника вуза'
```

Это псевдокод

Запишите здесь свой код Python

На этом все! Осталось добавить в программу функцию `output_results()`, и задача решена! На следующих двух страницах выделены все необходимые изменения и дополнения.

```
import ch1text

def count_syllables(words):
   count = 0

   for word in words:
      word_count = count_syllables_in_word(word)
      count = count + word_count

   return count

def count_syllables_in_word(word):
   count = 0

   endings = '.,;!?:'
   last_char = word[-1]

   if last_char in endings:
      processed_word = word[0:-1]
   else:
      processed_word = word

   if len(processed_word) <= 3:
      return 1

   if processed_word[-1] in 'eE':
      processed_word = processed_word[0:-1]

   vowels = "aeiouAEIOU"
   prev_char_was_vowel = False

   for char in processed_word:
      if char in vowels:
         if not prev_char_was_vowel:
            count = count + 1
        prev_char_was_vowel = True
      else:
         prev_char_was_vowel = False

   if processed_word[-1] in 'yY':
      count = count + 1
   return count
```

Это код, который мы уже написали ранее. Он должен быть в файле analyze.py

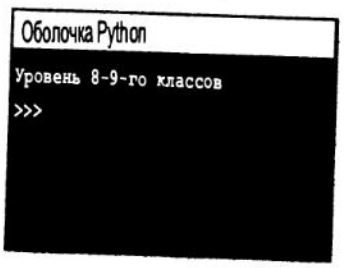
Продолжение на следующей странице

```
def count_sentences(text):
    count = 0

    terminals = '.?!'
    for char in text:
        if char in terminals:
            count = count + 1

    return count

def output_results(score):
    if score >= 90:
        print('Уровень 5-го класса')
    elif score >= 80:
        print('Уровень 6-го класса')
    elif score >= 70:
        print('Уровень 7-го класса')
    elif score >= 60:
        print('Уровень 8-9-го классов')
    elif score >= 50:
        print('Уровень 10-11-го классов')
    elif score >= 30:
        print('Уровень студента вуза')
    else:
        print('Уровень выпускника вуза')

def compute_readability(text):
    total_words = 0
    total_sentences = 0
    total_syllables = 0
    score = 0

    words = text.split()
    total_words = len(words)
    total_sentences = count_sentences(text)
    total_syllables = count_syllables(words)

    score = (206.835 - 1.015 * (total_words / total_sentences)
                     - 84.6 * (total_syllables / total_words))

    print(total_words, 'слов')
    print(total_sentences, 'предложений')
    print(total_syllables, 'слогов')
    print(score, ' - удобочитаемость')
    output_results(score)

compute_readability(ch1text.text)
```

Добавьте код новой функции `output_results()`

Итак, наш текст на уровне 8-9-го классов. Удобочитаемость такого уровня считается приемлемой для большинства книг и статей, так что все очень даже неплохо!

Не забудьте вызвать новую функцию, передав ей индекс удобочитаемости



Задача заключается в том, чтобы получить не слишком высокую, но и не слишком низкую оценку. Низкий индекс удобочитаемости означает, что текст будет трудным для восприятия. Например, индекс 30–50 соответствует уровню учебника для студентов. Это совсем не то, чего ожидает читатель на первых страницах книги, посвященной основам программирования. По правде говоря, он ожидает прямо противоположного.

К счастью, наша книга будет понятна даже старшеклассникам, так что полученная оценка выглядит вполне адекватно. Если бы мы хотели написать роман, ориентированный на молодежь, то ориентировались бы на еще более высокий индекс, скажем, 70–80, чтобы книга была доступна для восприятия школьникам средних классов.

Индекс	Уровень	Примечания
100–90	5-й класс	Очень легко читается, понятен 11-летнему школьнику
90–80	6-й класс	Легко читается, типичный разговорный язык
80–70	7-й класс	Относительно легко читается
70–60	8-9-й классы	Литературный язык, понятен школьникам 13–15 лет
60–50	10-11-й классы	Относительно тяжело читается
50–30	Студент	Тяжело читается
30–0	Выпускник вуза	Очень тяжело читается, понятен в основном выпускникам вузов

Дальнейшие улучшения

Хотите узнать индекс удобочитаемости любимой книги, статьи на сайте или собственного произведения? Придерживайтесь следующего порядка действий.

- Создайте новый файл с расширением *.py*.
- Добавьте в файл многострочный текст, который хотите проанализировать, заключите его в тройные кавычки и присвойте некой переменной (для удобства рекомендуем назвать ее `text`).
- Импортируйте созданный модуль в файл *analyze.py*.
- Вызовите функцию `compute_readability()`, передав ей в качестве аргумента имя переменной `text`, дополненное именем модуля.

В следующей главе вы узнаете способ получше

И не забывайте о том, что эвристический алгоритм можно дополнительно улучшить. Внимательно изучите список words – в нем еще остались проблемы, например двойные кавычки, которые тоже нужно исключить из анализа перед подсчетом слогов. Плюс у нас была идея создать список особых слов.

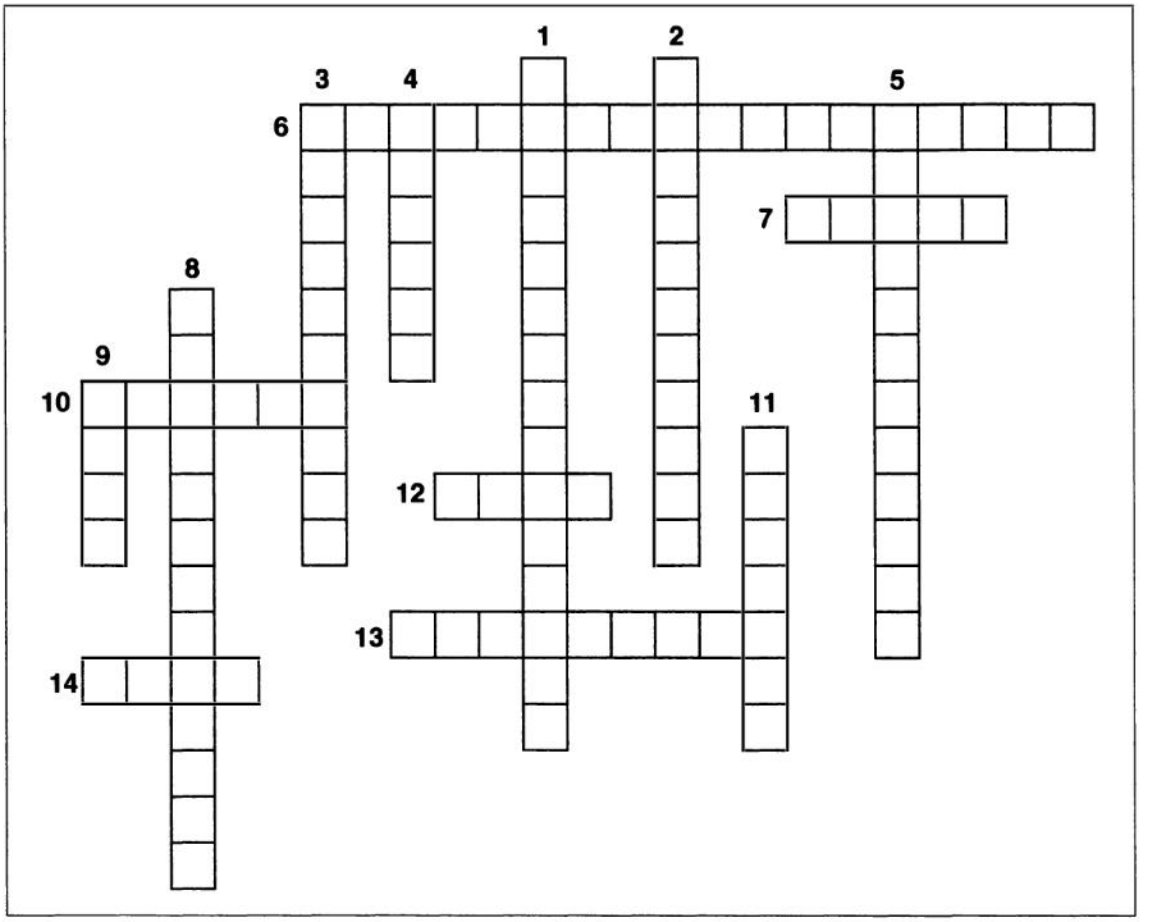
Как видите, работа программиста никогда не заканчивается. В то же время цели, поставленные нами в начале главы, успешно достигнуты. Осталось только подвести итоги и решить кроссворд. В любом случае примите поздравления – это была сложная глава, с которой вы справились!

САМОЕ ГЛАВНОЕ

- В Python многострочный текст нужно заключать в тройные кавычки.
- Длина строки, как и длина списка, определяется с помощью функции `len()`.
- Функция `split()` разбивает строку на слова, заносимые в отдельный список.
- Разделителями слов чаще всего служат пробельные символы.
- Строки — это неизменяемые последовательности, а списки — изменяемые.
- Строки нельзя редактировать в коде, а списки — можно.
- Строковый тип данных является неизменяемым в большинстве современных языков программирования.
- Для посимвольного просмотра строки можно использовать цикл `for`.
- К символам строки можно обращаться по индексу.
- С помощью булевого оператора `in` можно проверить вхождение значений в список или строку.
- Эвристический алгоритм позволяет получить наилучшую оценку или приемлемый ответ, но это не обязательно будет точное решение.
- Эвристические алгоритмы применяются в задачах, где получение точного решения оказывается слишком трудоемким или же способ решения неизвестен.
- В Python поддерживается синтаксис выделения подстроки из строки. Этот же синтаксис применим и к спискам.
- В отличие от других языков программирования, в Python нет отдельного символьного типа: любые текстовые данные здесь представляются как строки.

Кроссворд

Это была непростая глава. Сделайте паузу и попробуйте решить кроссворд.



По горизонтали

6. Набор перечисляемых элементов
7. Количество символов в строке
10. Набор записанных подряд символов
12. Диапазон символов строки
13. Часть строки
14. Фамилия математика, придумавшего формулу вычисления индекса удобочитаемости

По вертикали

1. Свойство текста, определяющее легкость его восприятия
2. Символ, разбивающий строку на слова
3. Система знаков препинания
4. Единичный элемент строки
5. Данные, не подлежащие редактированию
8. Алгоритм, не дающий точного решения
9. Часть слова
11. Книга, содержащая перечень слов языка

Текст, хранящийся в файле *ch6/text.txt*, необходимо превратить в файл Python. Для этого создайте в IDLE пустой файл и вставьте в него приведенный ниже код (текст, помещаемый в переменную `text`, берется из файла *text.txt*). Назовите полученный файл *ch1text.py*.

Далее запустите файл на выполнение, и вы должны увидеть на экране сохраненный текст.

Программа вывела весь текст, это его конец

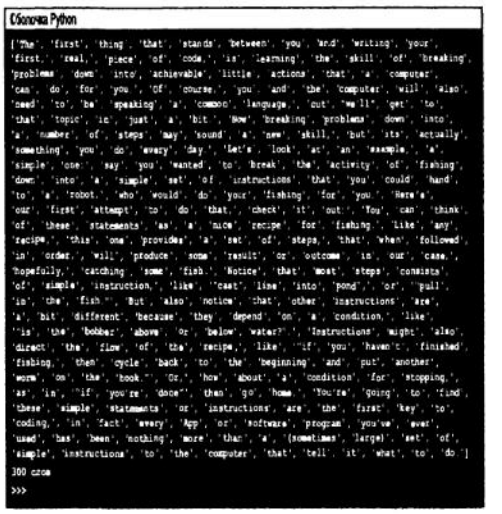
```
Оболочка Python
into pond", or "pull in the fish." But also notice that other
instructions are a bit different because they depend on a condition,
like "is the bobber above or below water?". Instructions might also
direct the flow of the recipe, like "if you haven't finished fishing,
then cycle back to the beginning and put another worm on the hook."
Or, how about a condition for stopping, as in "if you're done" then go
home.

You're going to find these simple statements or instructions are the
first key to coding, in fact every App or software program you've ever
used has been nothing more than a (sometimes large) set of simple
instructions to the computer that tell it what to do.
>>>
```

Возьмите карандаш
Решение

Теперь, когда вам известно назначение функции `split()`, вернемся к функции `compute_readability()`. Согласно псевдокоду нам следует написать функцию `count_words()`, но, как выясняется, с этой задачей прекрасно справляется встроенная функция `split()`. Таким образом, функция `count_words()` нам не нужна. Закончите приведенный ниже код.

```
import ch1text

def compute_readability(text):
    total_words = 0
    total_sentences = 0
    total_syllables = 0
    score = 0

    words = text.split()
    total_words = len(words)

    print(words)
    print(total_words, 'слов')
    print(text)

compute_readability(ch1text.text)
```

Чтобы подсчитать количество слов, передайте функции `len()` *список слов*



Возьмите карандаш
Решение

Напишите код, который проверяет, является ли символ знаком конца предложения (точка, вопросительный или восклицательный знаки), и, если это так, увеличивает на единицу значение переменной `count`.

```
def count_sentences(text):
    count = 0

    for char in text:
        if char == '.' or char == '?' or char == '!':
            count = count + 1

    return count
```

Если переменная char является знаком конца предложения, то увеличиваем значение переменной count на единицу

Возьмите карандаш
Решение

Попробуйте переписать функцию `count_sentences()`, сделав ее проще (и понятнее) с помощью оператора `in`. В приведенном ниже варианте функции мы удалили код сравнения со знаками препинания. Вместо этого добавлена локальная переменная `terminals`, в которой перечислены все знаки окончания предложений. Закончите инструкцию `if` так, чтобы она проверяла, является ли текущий символ терминальным, используя оператор `in`.

```
def count_sentences(text):
    count = 0

    terminals = '.?!'

    for char in text:

        if char in terminals:
            count = count + 1

    return count
```

Так намного короче и понятнее!

Учимся понимать

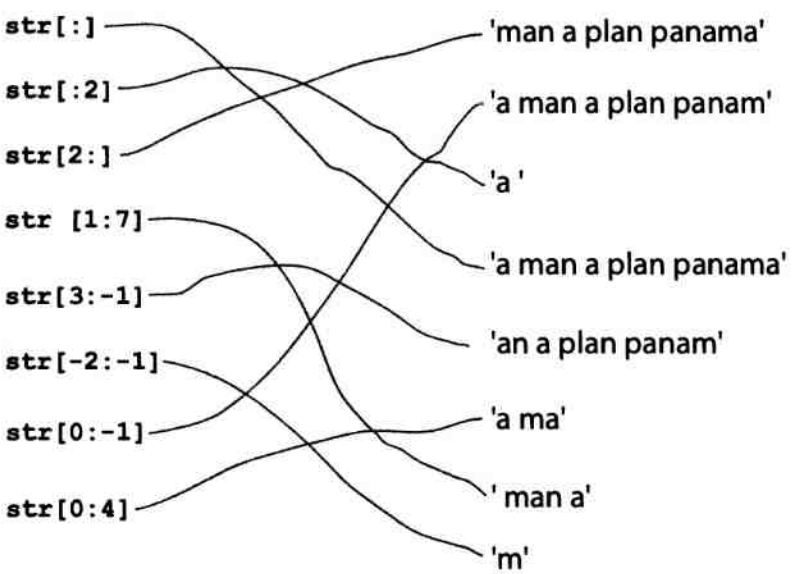
Решение

Изучая приведенную выше функцию **`count_syllables_in_word()`**, представьте, что вы интерпретатор Python. Функции передается аргумент **`"roomful"`**. Мысленно пройдите весь цикл, отслеживая изменения локальных переменных.

char	prev_char_was_vowel	count
r	False	0
o	True	1
o	True	
m	False	
f	False	
u	True	2
l	False	

Кто что делает? Решение

Мы выполнили ряд операций по созданию срезов, но затем все перемешалось. Не поможете нам расставить все по своим местам? Одно соответствие мы нашли сами.



Возьмите карандаш
Решение

Теперь переменная `process_word` содержит слово, из которого удален концевой знак препинания. Напишите код для удаления из слова конечной буквы 'e'. Это будет буквально несколько строк кода.

Если последний символ в строке `process_word` *равен либо 'e', либо 'E', то...*

```
if processed_word[-1] in 'eE':
    processed_word = processed_word[0:-1]
```

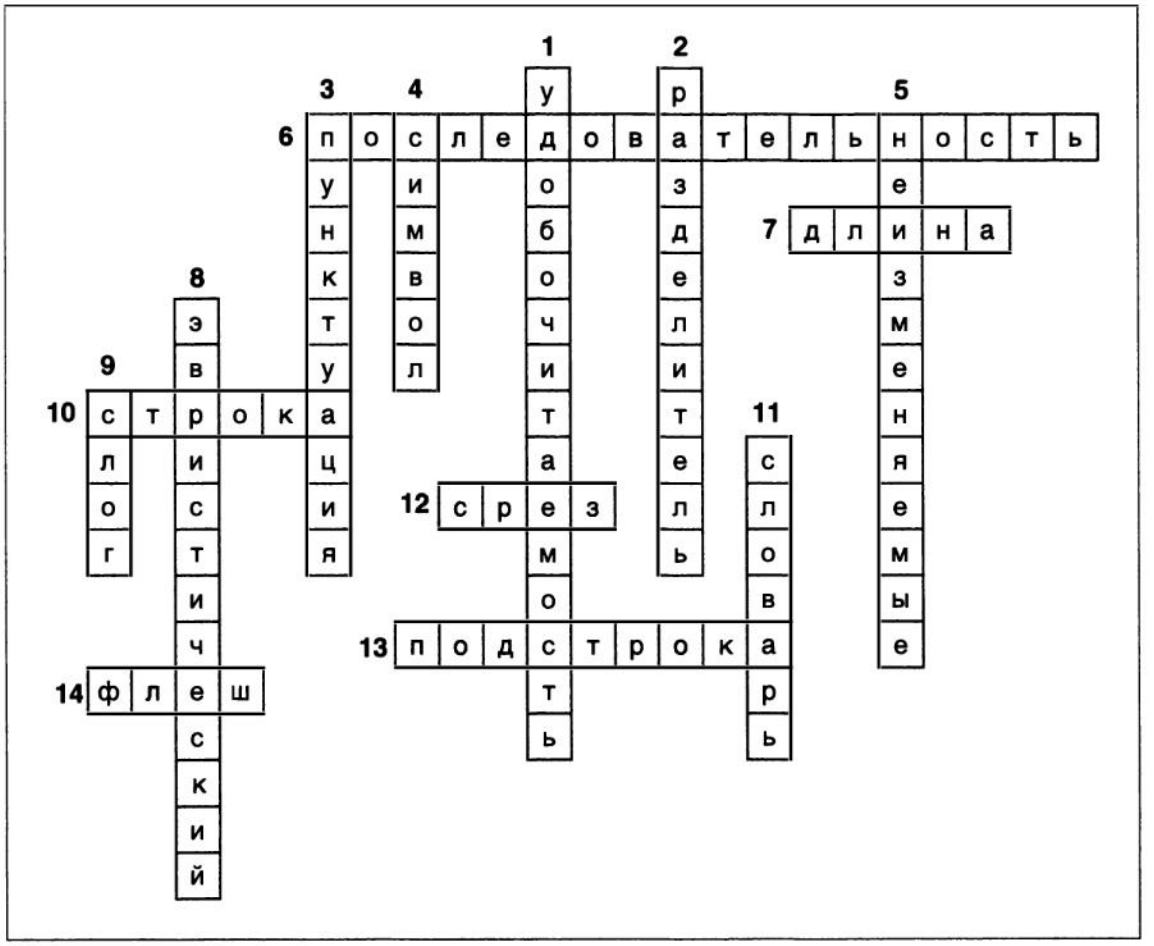
...сохраняем в переменной исходную строку без последнего символа

Возьмите карандаш
Решение

Мы наконец получили индекс удобочитаемости, и осталось вывести результаты. Напишите функцию `output_results()`, которая получает значение индекса и выводит на экран строку с описанием уровня текста.

```
def output_results(score):
   if score >= 90:
      print('Уровень 5-го класса')
   elif score >= 80:
      print('Уровень 6-го класса')
   elif score >= 70:
      print('Уровень 7-го класса')
   elif score >= 60:
      print('Уровень 8-9-го классов')
   elif score >= 50:
      print('Уровень 10-11-го классов')
   elif score >= 30:
      print('Уровень студента вуза')
   else:
      print('Уровень выпускника вуза')
```

Кроссворд: решение



7 классы, объекты, методы и модули

Модульное программирование

Написанные вами программы становятся все более сложными. Это заставляет тщательнее продумывать структуру и организацию программ. Вы уже знаете, что функции позволяют группировать строки кода в компактные блоки, удобные для повторного использования. Вы также знаете, что наборы функций и переменных можно объединять в модули, подключаемые к программам. В этой главе мы вернемся к теме модулей и рассмотрим, как работать с ними эффективнее (и даже делиться ими с другими пользователями). Кроме того, вас ждет знакомство с ключевыми строительными блоками программ — *объектами*. Вы увидите, что в Python они встречаются повсеместно, даже там, где вы не ожидаете их встретить.

Текст из его книги *Little Brother*

it," I said. "You're the best coder I know.
genius, Jolu. I would be honored if you'd

ngers some more. "It's just -- You know.
an's the smart one. Darryl was... He was your
l, the guy who had it all organized, who
Being the programmer, that was my thing. It
ying you didn't need me."

ch an idiot. Jolu, you're the best-qualified
this. I'm really, really, really --"

. Stop. Fine. I believe you. We're all really
w. So yeah, of course you can help. We can

If you've never programmed a computer, you should. There's nothing like it in the whole world. When you program a computer, it does *exactly* what you tell it to do. It's like designing a machine -- any machine, like a car, like a faucet, like a gas-hinge for a door -- using math and instructions. It's awesome in the truest sense: it can fill you with awe.

A computer is the most complicated machine you'll ever use. It's made of billions of micro-miniaturized transistors that can be configured to run any program you can imagine. But when you sit down at the keyboard and write a line of code, those transistors do what you tell them to.

Most of us will never build a car. Pretty much none of us will ever create an aviation system. Design a building. Lay out a city.

Офисный диалог

Григорий. Все просто: отправим ему файл *analyze.py*. Он импортирует его и вызовет функцию `compute_readability()`. Дело в шляпе!

Федор. Согласен. Модуль – это просто файл Python, который несложно выслать.

Игорь. Боюсь, все не совсем так. Да, модуль – это обычный файл, его не сложно подключить с помощью инструкции `import`, но есть нюансы.

Григорий. Какие именно?

Игорь. Ну например, если вы не забыли, наш файл *analyze.py* анализирует текст из главы 1.

Федор. Да, это было удобно для тестирования анализатора. Кори придется переписать модуль для работы со своим текстом.

Игорь. Вообще-то, было бы правильно, чтобы другие пользователи могли использовать наш код без дополнительного редактирования. К тому же я хотел бы сохранить тестовый код на случай, если в будущем мы захотим улучшить эмпирический алгоритм. Должно быть какое-то другое решение.

Григорий. Есть идеи?

Игорь. Я тут кое-что разузнал. Оказывается, существует способ организации модулей, при котором и мы сохраняем наш тестовый код, и Кори сможет использовать его для своего анализа.

Григорий. Хотелось бы узнать подробности.

Федор. Только не говори, что все придется переписывать!

Игорь. Все не настолько сложно. Уверен, результат вам понравится. Внесенные изменения сделают код более удобным для повторного использования.

Знакомство с модулями

Как вы уже знаете, для импорта модуля применяется инструкция `import`:

```
import random
```

Импорт модуля `random`

Это позволяет ссылаться на содержащиеся в нем переменные и функции, предваряя их именем модуля.

Мы вызываем функцию `randint()` из модуля `random`

```
num = random.randint(0, 9)
```

Начинаем с имени модуля...

...после которого ставим оператор-точку...

...а затем указываем имя функции или переменной модуля

Мы уже не раз использовали в книге оператор-точку. В данном случае он означает "поискать функцию `randint()` в модуле `random`".

Как будет показано далее, оператор-точка применяется также при работе с объектами

Не бойтесь задавать вопросы

В: Откуда Python знает, где искать импортируемый модуль?

О: Хороший вопрос, ведь в операции импорта указывается только имя модуля, но не путь к нему. Как же Python находит модули? Сначала он просматривает собственный список встроенных модулей (`random` как раз один из них) и, если не находит нужный модуль, то просматривает локальную папку, из которой запускалась программа. В случае необходимости можно настраивать поиск и в других папках.

В: Мне встречался термин *библиотека Python*. Имеет ли он отношение к модулям?

О: Библиотека — это более общий термин, под которым чаще всего понимают модули, выложенные в открытый доступ. Есть еще термин *пакет*, означающий набор модулей, которые предназначены для совместного использования.

В: Что произойдет, если я, к примеру, импортирую модуль `random` и другой модуль, который тоже импортирует модуль `random`? Не возникнет ли конфликт?

О: Нет, Python отслеживает подключаемые модули, чтобы не приходилось импортировать их снова и снова. Кроме того, допускается, чтобы программа и внешний модуль импортировали один и тот же модуль.

Офисный диалог продолжается...

Федор. Да, но только в таком виде, чтобы с ним могли работать другие пользователи.

Игорь. На данный момент мы тестируем в коде текст главы 1.

Юлия. А нельзя ли это убрать?

Игорь. Можно, но мы хотим оставить данный тест на случай, если понадобится улучшить эвристический алгоритм.

Юлия. И как же нам быть?

Федор. Игорь выяснил интересную особенность модулей Python. Оказывается, можно структурировать файл модуля таким образом, чтобы при запуске он знал, запущен ли он как основная программа или импортирован в другой файл Python.

Юлия. Что это нам даст?

Игорь. Сама посуди: если кто-то напрямую запускает файл *analyze.py*, то наверняка это мы, а значит, мы хотим выполнить тестовый код. Если же кто-то другой импортирует файл *analyze.py*, значит, тестовый код следует пропустить.

Юлия. Теперь более-менее понятно. А как это сделать?

Игорь. Есть специальная переменная, позволяющая узнать, запущен ли файл *analyze.py* напрямую как основная программа. В этом случае мы выполняем тестовый код, а иначе игнорируем его. Сейчас покажу, как это сделать...

Глобальная переменная __name__

При запуске файла Python интерпретатор создает специальную глобальную переменную `__name__` (два подчеркивания, слово `name` и снова два подчеркивания). В эту переменную записывается одно из двух значений: если файл вызван напрямую как основная программа, то переменная будет содержать строку `"__main__"`, в противном случае она будет содержать имя модуля (в нашем случае `"analyze"`). Благодаря наличию переменной `__name__` существует универсальное соглашение, позволяющее проверить способ запуска модуля Python.

Если программа вызвана напрямую, то в этом блоке можно выполнить любой код

```
if __name__ == '__main__':
    print("Это основная программа.")
```

Важно понимать: этот вызов функции `print()` будет проигнорирован, если модуль импортируется

Тест-драйв

Давайте проверим, как работает переменная `__name__`. Ниже приведены два файла Python. Введите оба файла в IDLE, сохраните их *в одной и той же папке*, запустите по очереди и сравните результаты.

```
if __name__ == '__main__':
    print("Это основная программа.")
else:
    print("Это простой модуль.")
```

just_a_module.py

```
import just_a_module

print('Привет от main.py!')
```

main.py

Оболочка Python

```
Это основная программа
>>>
```

В данном случае модуль `just_a_module` считает себя основной программой

Вот что мы получаем при выполнении обоих файлов

Оболочка Python

```
Это простой модуль.
Привет от main.py!
>>>
```

В данном случае программа `main.py` импортирует модуль `just_a_module`

Офисный диалог продолжается...

Григорий. Получается, нужно всего лишь проверить, содержит ли переменная `__name__` строку `"__main__"`, и, если это так, выполнить тестовый код. В противном случае ничего делать не нужно. После этого программу можно будет отдавать Кори.

Федор. Так и сделаем... Минуточку! А как Кори узнает, какую именно функцию использовать?

Григорий. Давай не все сразу. К этой проблеме мы вернемся позже, а пока внесем указанные изменения.

Федор. Согласен!

Обновление файла analyze.py

Также скопируйте файл ch1text.py из папки ch6 в папку ch7

Скопируйте файл *analyze.py* из папки *ch6* в папку *ch7*, после чего откройте его и внесите следующие изменения.

Это начало файла analyze.py

```
import ch1text

def count_syllables(words):
    count = 0

    for word in words:
        word_count = count_syllables_in_word(word)
        count = count + word_count

    return count

def count_syllables_in_word(word):
    count = 0

    endings = '.,;!?:'
    last_char = word[-1]
```

В начале модуля удалите инструкцию импорта файла ch1text: мы переместим ее в конец файла

```
def compute_readability(text):
    total_words = 0
    total_sentences = 0
    total_syllables = 0
    score = 0

    words = text.split()
    total_words = len(words)
    total_sentences = count_sentences(text)
    total_syllables = count_syllables(words)

    score = (206.835 - 1.015 * (total_words / total_sentences)
                     - 84.6 * (total_syllables / total_words))


    output_results(score)

if __name__ == "__main__":
    import ch1text
    print('Текст главы 1:')
    compute_readability(ch1test.text)
```

Это конец файла analyze.py

Проверяем, содержит ли переменная __name__ значение "__main__"

Если да, импортируем файл ch1text и определяем индекс удобочитаемости

Добавляем также эту инструкцию

Не забудьте сделать отступ в четыре пробела перед вызовом функции compute_readability()

Да, инструкцию import можно использовать в любом месте программы!

Тест-драйв

Внесите указанные выше изменения в файл *analyze.py* и запустите его для проверки. Поскольку он запускается как основная программа, вы должны получить тот же результат, что и в главе 6. Не забудьте скопировать файл *ch1text.py* в ту же папку.

Все работает, как и ожидалось. Мы запустили файл analyze.py как основную программу, поэтому она проверяет текст файла ch1text.py

```
Оболочка Python
Текст главы 1:
Уровень 8-9-го классов
>>>
```

Использование файла analyze.py в качестве модуля

Как другим пользователям работать с нашим кодом? Они должны сначала импортировать модуль `analyze`, а затем вызвать функцию `compute_readability()`, передав ей строку с анализируемым текстом. Давайте создадим новый файл *cory_analyze.py* и запишем в нем соответствующий код.

Импорт модуля

```
import analyze

analyze.compute_readability("""
If you've never programmed a computer, you should. There's nothing like it in the
whole world. When you program a computer, it does exactly what you tell it to do.
It's like designing a machine:  any machine, like a car, like a faucet, like a gas
hinge for a door using math and instructions. It's awesome in the truest sense it
can fill you with awe.

A computer is the most complicated machine you'll ever use. It's made of billions
of micro miniaturized transistors that can be configured to run any program you
can imagine. But when you sit down at the keyboard and write a line of code, those
transistors do what you tell them to.

Most of us will never build a car. Pretty much none of us will ever create an
aviation system. Design a building. Lay out a city.""")
```

Вызываем функцию `analyze.compute_readability()` и передаем ей текст

Помните: можно использовать три двойные кавычки подряд для создания многострочного текста

Тест-драйв

Введите файл *cory_analyze.py* и проверьте его работу. Поскольку файл *analyze.py* запускается в качестве модуля, тестовый код в нем не выполняется. Соответственно, результаты работы программы будут другими.

Вводить текст из книги Кори совсем не обязательно. Вы найдете его в файле ch7/cory.txt

Текст Кори соответствует уровню 7-го класса. Это идеально для книги, которая ориентирована на подростков!

Оболочка Python

```
Уровень 7-го класса
>>>
```

С этим нам поможет Python.

В Python справочную документацию можно включить непосредственно в исходный код. Вы уже умеете добавлять комментарии в код, но они нужны для вставки пояснений к отдельным инструкциям. В Python также поддерживаются *документирующие строки*, позволяющие создавать высокоуровневую документацию для тех, кто будет работать с модулем (но не собирается анализировать его построчно).

Документирующие строки в том или ином виде есть в большинстве современных языков программирования

Документирующие строки имеют очень простой формат: они добавляются в текстовом виде в начало модуля в качестве общего описания и в каждое объявление функции (а также в каждое объявление класса, о чем мы поговорим позже).

Как применять документирующие строки? Нужно ли открывать файл модуля, чтобы читать их? Нет, есть более простой способ: в Python имеется функция `help()`, позволяющая просматривать документацию с помощью интерпретатора.

Давайте добавим несколько документирующих строк в файл *analyze.py*, чтобы понять, как они работают.

Добавление документирующих строк в файл analyze.py

Мы снабдим наш модуль документирующими строками, содержащими справочные сведения по его использованию.

```
"""В модуле analyze вычисляется индекс удобочитаемости Флеша-Кинкейда для
   заданного текста и на основе полученной оценки определяется уровень
   удобочитаемости текста.

"""

def count_syllables(words):
    """Эта функция получает список слов и возвращает общее
       число слогов во всех словах.
    """
    count = 0

    for word in words:
        word_count = count_syllables_in_word(word)
        count = count + word_count

    return count

def count_syllables_in_word(word):
    """Эта функция получает слово в строковом формате и возвращает
       число слогов в нем. Учтите, что это эвристическая функция,
       поэтому ее результат не всегда будет точным на 100%.
    """
    count = 0

    endings = '.,;!?:'   # это возможные знаки препинания в конце слов
    last_char = word[-1]

    if last_char in endings:
        processed_word = word[0:-1]
    else:
        processed_word = word

    if len(processed_word) <= 3:
        return 1

    if processed_word[-1] in 'eE':
        processed_word = processed_word[0:-1]


    vowels = "aeiouAEIOU"
    prev_char_was_vowel = False

    for char in processed_word:
        if char in vowels:
            if not prev_char_was_vowel:
                count = count + 1
            prev_char_was_vowel = True
        else:
            prev_char_was_vowel = False

    if processed_word[-1] in 'yY':
        count = count + 1

    return count
```

Можно добавить многострочный текст в начало модуля...

...и под любым объявлением функции

В справку Python включаются только документирующие строки. Комментарии в коде, подобные этому, не включаются

Продолжение на следующей странице...

```
def count_sentences(text):
    """Эта функция подсчитывает число предложений в тексте, используя
       точки, вопросительный и восклицательный знаки в качестве
       концевых символов.
    """
    count = 0

    terminals = '.?!'
    for char in text:
        if char in terminals:
            count = count + 1

    return count

def output_results(score):
    """Эта функция получает индекс Флеша-Кинкейда и сообщает
       соответствующий ему уровень удобочитаемости текста.
    """
    if score >= 90:
        print('Уровень 5-го класса')
    elif score >= 80:
        print('Уровень 6-го класса')
    elif score >= 70:
        print('Уровень 7-го класса')
    elif score >= 60:
        print('Уровень 8-9-го классов')
    elif score >= 50:
        print('Уровень 10-11-го классов')
    elif score >= 30:
        print('Уровень студента вуза')
    else:
        print('Уровень выпускника вуза')

def compute_readability(text):
    """Эта функция получает строку текста произвольной длины и сообщает
       уровень удобочитаемости в соответствии с имеющейся шкалой оценок.
    """
    total_words = 0
    total_sentences = 0
    total_syllables = 0
    score = 0

    words = text.split()
    total_words = len(words)
    total_sentences = count_sentences(text)
    total_syllables = count_syllables(words)

    score = (206.835 - 1.015 * (total_words / total_sentences)
                     - 84.6 * (total_syllables / total_words))

    output_results(score)

if __name__ == "__main__":
    import ch1text
    print('Текст главы 1:')
    compute_readability(ch1text.text)
```

Еще несколько документирующих строк

Вы вольны документировать код настолько подробно, насколько посчитаете нужным. Python в данном случае не накладывает никаких ограничений. В других языках программирования к документирующим строкам выдвигаются определенные требования, позволяющие стандартизировать их формат

Добавьте документирующие строки в файл *analyze.py*. При желании можете дополнить их, как посчитаете нужным. После этого необходимо выполнить определенные действия. Помните, мы говорили о том, что Python из соображений эффективности не позволяет повторно импортировать один и тот же модуль? Вместо этого он постоянно хранит в памяти его кешированную версию. Именно поэтому, несмотря на внесенные в модуль *analyze.py* изменения, Python будет продолжать работать с его старой, кешированной версией (лишенной документирующих строк). **Чтобы обойти это ограничение, нужно выйти из IDLE.** *Полностью завершите работу в редакторе кода*, после чего снова запустите IDLE, откройте файл *cory_analyze.py* и выполните его, чтобы удостовериться в отсутствии ошибок в документирующем коде. Теперь перейдите в окно консоли и следуйте приведенным ниже инструкциям для просмотра документации.

Все зависит от операционной системы и версии Python

Не пренебрегайте запуском файла cory_analyze.py, так как он импортирует модуль analyze.py, а это позволяет убедиться в правильности расположения модуля

Оболочка Python

```
>>> help(analyze)
Help on module analyze:

NAME
    analyze

DESCRIPTION
    В модуле analyze вычисляется индекс удобочитаемости флеша-Кинкейда для
    заданного текста и на основе полученной оценки определяется уровень
    удобочитаемости текста.

FUNCTIONS
    compute_readability(text)
        Эта функция получает строку текста произвольной длины и сообщает
        уровень удобочитаемости в соответствии с имеющейся шкалой оценок.

    count_sentences(text)
        Эта функция подсчитывает число предложений в тексте, используя
        точки, вопросительный и восклицательный знаки в качестве
        концевых символов.

    count_syllables(words)
        Эта функция получает список слов и возвращает общее
        число слогов во всех словах.

    count_syllables_in_word(word)
        Эта функция получает слово в строковом формате и возвращает
        число слогов в нем. Учтите, что это эвристическая функция,
        поэтому ее результат не всегда будет точным на 100%.

    output_results(score)
        Эта функция получает индекс флеша-Кинкейда и сообщает
        соответствующий ему уровень удобочитаемости текста.
```

Вот как получить справку по модулю `analyze`

Здесь мы получаем справку обо всем модуле `analyze`

Оболочка Python

```
>>> help(analyze.compute_readability)
Help on function compute_readability in module analyze:

compute_readability(text)
    Эта функция получает строку текста произвольной длины и сообщает
    уровень удобочитаемости в соответствии с имеющейся шкалой оценок.
```

Можно также получить справку по отдельной функции модуля

Сила мысли

Раз уж вы задумались над тем, как работают с модулем *analyze.py* другие программисты, то, может быть, имеет смысл перестроить его? Что, если программист захочет, к примеру, узнать индекс удобочитаемости в числовом виде? Подумайте о том, что нужно изменить в коде модуля.

Знакомство с другими модулями Python

Теперь, когда вы научились создавать собственные модули и снабжать их справочной документацией, пришло время познакомиться с другими важными модулями. С наиболее полезными из них мы будем работать в последующих главах, также ряд модулей описан в приложении. А пока просто упомянем наиболее популярные модули.

Что еще за черепашки?

Это одна из наших любимых тем, поскольку данный модуль встроен в Python и позволяет изучать программирование в игровой форме. Импортируйте модуль `turtle`, и можете приступать к созданию черепашек! Но сначала нужно узнать, что собой представляют такие черепашки.

В Python черепашки живут на сетке. Центр сетки — это точка с координатами (0, 0)

У черепашек в Python есть перо, с помощью которого они рисуют

Каждая черепашка на сетке имеет местоположение, или координаты (x, y)

Перо может быть поднято или опущено, а еще у него есть цвет и размер (толщина)

У каждой черепашки есть направление, указывающее в ту сторону, куда она движется

Черепашки бывают разных форм: черепаховая, как в данном случае, круглая, стреловидная и т.п.

Все это можно считать атрибутами черепашки

Чеперашка может двигаться вперед, назад или в конкретную точку сетки

Черепашка может повернуть вправо или влево, что меняет направление ее движения

Если перо черепашки опущено, то она оставляет след при движении

Все это можно считать параметрами поведения черепашки

А вы ожидали учебный курс уровня Массачусетского технологического института?
Так он перед вами! Анимируемые черепашки были придуманы в МТИ еще в 60-х годах прошлого столетия одним из пионеров компьютерных наук, Сеймуром Пейпертом. С тех пор они оказали огромное влияние на разработку языков программирования и нашли широкое применение в обучении компьютерных инженеров и математиков (не говоря уже о подростках, изучающих программирование). Так что просим вас воздержаться от поспешных оценок, поскольку в этой главе (а также в нескольких последующих главах) нам предстоит сделать много полезного с помощью черепашек. Надеемся, вы их тоже полюбите.

Создание собственной черепашки

Давайте же не мешкая займемся созданием черепашки!
Для начала импортируем модуль `turtle`.

```
import turtle
```

Пока что не нужно вводить код, мы сделаем это чуть позже

Команда создания черепашки выглядит следующим образом.

Модуль turtle

Оператор-точка

Напоминает вызов функции, только обычно имя функции не начинается с большой буквы

При создании черепашки появляется такое окно

```
slowpoke = turtle.Turtle()
```

Назначим новой черепашке переменную slowpoke

Так создается новая черепашка

Заставим черепашку сделать что-нибудь.

Оператор-точка

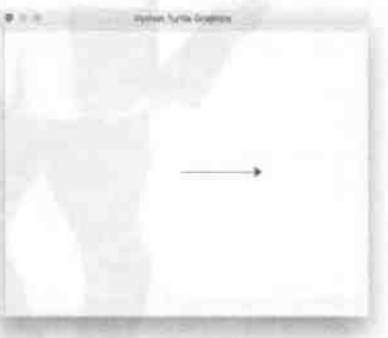
Одно из поведений черепашки

Аргумент, как в обычной функции

```
slowpoke.forward(100)
```

Переменная, ссылающаяся на новую черепашку

Эта команда заставляет черепашку переместиться на 100 позиций вперед

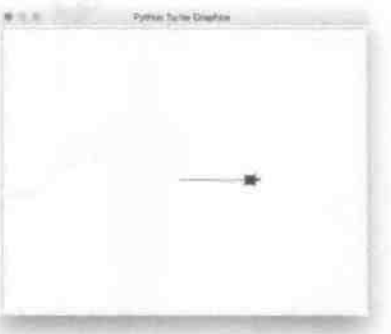


Итак, наша черепашка переместилась на 100 позиций вперед, нарисовав линию в направлении движения. Но вы заметили, что экранная фигура не очень-то напоминает черепашку? Исправим это упущение, задав правильный атрибут и перезапустив код.

```
slowpoke.shape('turtle')
```

Задаем атрибут shape черепашки равным 'turtle'

Так лучше!



Изучение черепашек

Настало время и вам попробовать свои силы. Давайте заставим черепашку выполнить какие-то действия, написав соответствующий код. Сохраните его в файле *turtle_test.py*.

```
import turtle

slowpoke = turtle.Turtle()
slowpoke.shape('turtle')

slowpoke.forward(100)
slowpoke.right(90)
slowpoke.forward(100)
slowpoke.right(90)
slowpoke.forward(100)
slowpoke.right(90)
slowpoke.forward(100)
slowpoke.right(90)

turtle.mainloop()
```

Сначала импортируем модуль `turtle`

Далее создаем новую черепашку. Вскоре мы детальнее поговорим о том, как создаются черепашки, а пока достаточно знать, что эта команда создает для вас личную черепашку

Далее назначаем черепашке правильную форму вместо стандартного треугольника

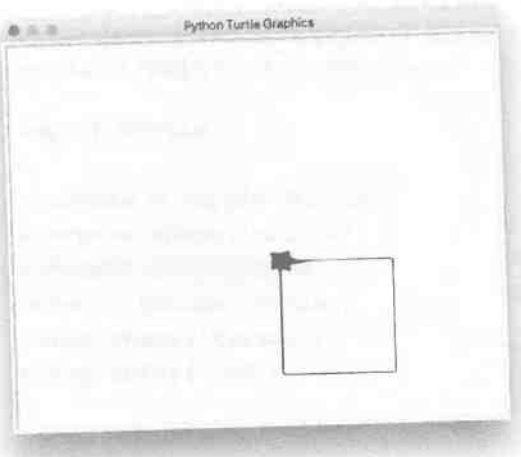
С помощью функции `forward()` приказываем черепашке переместиться вперед на 100 позиций, а с помощью функции `right()` заставляем ее повернуть вправо на 90°

Повторяем то же самое еще три раза

О назначении этой команды мы поговорим позже, но по сути она позволяет модулю `turtle` контролировать все события, происходящие в его окне, включая закрытие программы при щелчке на кнопке Close. Эта строка должна быть последней

Протестируйте код, запустив программу. Вы увидите, что черепашка, двигаясь вперед, четырежды поворачивает направо, оставляя после себя след в форме квадрата.

Вот что мы получили!



Если вы не видите это окно и в оболочке нет никаких ошибок, то поищите его позади окон Python. В некоторых системах окно модуля не появляется автоматически поверх остальных окон

Не называйте свой файл *turtle.py*

Следите за тем, чтобы имя тестового файла не совпадало с именем модуля, иначе у вас возникнут серьезные проблемы: при каждом обращении к модулю `turtle` будет загружаться ваш файл. Это касается любых создаваемых вами файлов: не давайте им имена стандартных модулей, особенно тех, которые импортируете в собственные программы.

Не бойтесь задавать вопросы

В: Угол поворота черепашки задается в градусах?

О: Да. В частности, для полного оборота по часовой стрелке нужно повернуть черепашку на 360 градусов. Поворот на 90 градусов соответствует четверти оборота и т.д.

В: Что означает аргумент `100` при перемещении вперед?

О: Сто единиц. В данном случае единицы соответствуют экранным пикселям. Таким образом, команда `turtle.forward(50)` дает указание черепашке переместиться вперед 50 пикселей.

В: Почему черепашка обретает нужную форму только по команде? Я думал, она уже является черепашкой!

О: По историческим причинам черепашка изначально имеет треугольную форму. Она также может выглядеть как квадратик, кружок, стрелка; разрешается даже назначать ей произвольную картинку. Но что может быть забавнее настоящей черепашки?

Возьмите карандаш

Позади уже почти семь глав книги, поэтому вам должно быть вполне по силам следующее упражнение. Преобразуйте приведенный выше код рисования квадрата в отдельную функцию, которая называется `make_square()` и имеет единственный аргумент: черепашку. Постарайтесь также удалить весь повторяющийся код. Сверьтесь с нашим решением, которое приведено в конце главы.

Обязательно изучите нашу версию, поскольку именно с ней мы будем работать далее

Еще одна черепашка

Как насчет друга для нашей черепашки? Давайте напишем код создания второй черепашки.

```
import turtle

slowpoke = turtle.Turtle()
slowpoke.shape('turtle')
pokey = turtle.Turtle()
pokey.shape('turtle')
pokey.color('red')

def make_square(the_turtle):
    for i in range(0, 4):
        the_turtle.forward(100)
        the_turtle.right(90)

make_square(slowpoke)
pokey.right(45)
make_square(pokey)

turtle.mainloop()
```

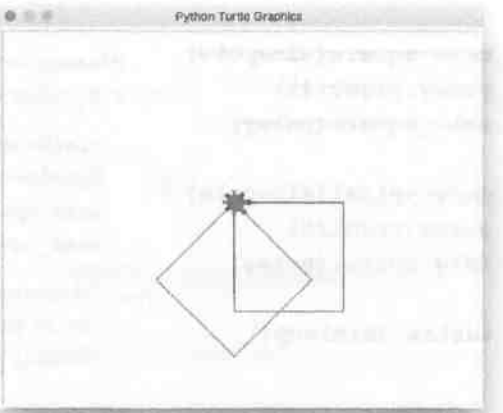
Это новый код, который мы обобщили в предыдущем упражнении

Создаем вторую черепашку и присваиваем ее переменной pokey. По форме она тоже черепаха, а цвет задан красным

Разворачиваем вторую черепашку на 45° вправо

Передаем черепашку pokey в функцию make_square()

В этом коде черепашка slowpoke рисует квадрат, как и раньше, а черепашка pokey рисует повернутый красный квадрат



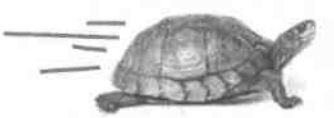
Если функция make_square() кажется вам незнакомой, просмотрите ответ к последнему упражнению

С помощью функций, оказывается, можно создавать удивительные вещи. Давайте создадим новую функцию на основе функции make_square() для получения интересного графического эффекта.

```
import turtle

slowpoke = turtle.Turtle()
slowpoke.shape('turtle')
slowpoke.color('blue')
pokey = turtle.Turtle()
pokey.shape('turtle')
pokey.color('red')
```

Ради интереса поменяем цвет пера черепашки slowpoke на синий

Продолжение на следующей странице...



```
def make_square(the_turtle):
    for i in range(0, 4):
        the_turtle.forward(100)
        the_turtle.right(90)

def make_spiral(the_turtle):
    for i in range(0, 36):
        make_square(the_turtle)
        the_turtle.right(10)

make_square(slowpoke)
pokey.right(45)
make_square(pokey)

make_spiral(slowpoke)
pokey.right(5)
make_spiral(pokey)

turtle.mainloop()
```

Добавим новую функцию make_spiral()

Функция make_spiral() вызывает функцию make_square() 36 раз и каждый раз заставляет черепашку повернуть на 10°

Удаляем эти обращения к функции make_square()

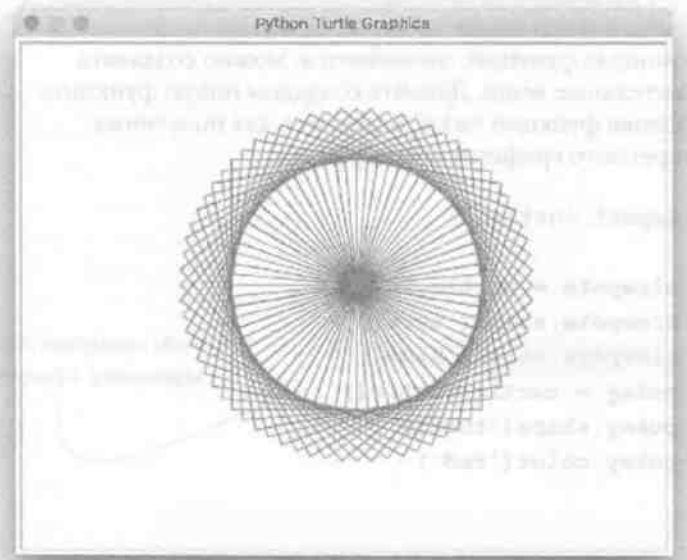
Теперь мы будем вызывать функцию make_spiral() вместо make_square()

Черепашка pokey повернет на 5° вправо, прежде чем начать рисовать спираль

Обратите внимание на то, что у каждой черепашки своя позиция, свое направление, свой цвет и т.п.

Тест-драйв

Обновите код в файле *turtle_test.py* и протестируйте его. Такой результат вы ожидали получить?



Вот что у нас получилось — напоминает игрушку-спирограф

Поэкспериментируйте с параметрами и атрибутами функций, чтобы понять, как они влияют на конечный результат. А мы продолжим эксперименты на следующей странице

ДРУГИЕ ОПЫТЫ С ЧЕРЕПАШКАМИ

Мы подготовили для вас еще несколько экспериментов с черепашками. Внимательно изучите каждый из них, подумайте, что будет делать черепашка, а затем запустите код, чтобы проверить свои предположения. Попробуйте поменять аргументы функций. Как при этом изменится результат на экране?

Эксперимент #1

```
for i in range(5):
    slowpoke.forward(100)
    slowpoke.right(144)
```

Что произойдет, если поменять это число? Или это?

Удалите из файла turtle_test.py все, кроме первых трех строк (и последнего вызова mainloop()), после чего вставьте этот код посредине

Нарисуйте здесь результат

Эксперимент #2

```
slowpoke.pencolor('blue')
slowpoke.penup()
slowpoke.setposition(-120, 0)
slowpoke.pendown()
slowpoke.circle(50)

slowpoke.pencolor('red')
slowpoke.penup()
slowpoke.setposition(120, 0)
slowpoke.pendown()
slowpoke.circle(50)
```

Мы используем несколько новых функций: задаем цвет пера, поднимаем перо, перемещаем его в новую позицию и опускаем, после чего рисуем окружность

Что произойдет, если удалить вызовы функции penup()?

Эксперимент #3

```
def make_shape(t, sides):
    angle = 360/sides
    for i in range(0, sides):
        t.forward(100)
        t.right(angle)

make_shape(slowpoke, 3)
make_shape(slowpoke, 5)
make_shape(slowpoke, 8)
make_shape(slowpoke, 10)
```

Что произойдет при других значениях, например 1, 2, 50?

Как обычно, ответы приведены в конце главы

Кто вы, черепашки?

Вернемся к коду создания самой первой черепашки. На первый взгляд, в нем нет ничего необычного: в единственной инструкции вызывается функция `Turtle()` модуля `turtle`.

Модуль turtle

Оператор-точка

Выглядит как вызов функции

```
slowpoke = turtle.Turtle()
```

Получается, что в модуле `turtle` есть функция, отвечающая за создание черепашек. Но что это за "зверь" такой: черепашка? Нам известны стандартные типы данных: целые числа, строки, списки, булевы значения и др. Черепашка — новый тип данных? Чтобы узнать это, обратимся к справочной системе Python.

Оболочка Python

```
>>> import turtle
>>> help(turtle.Turtle)
Help on class Turtle in module turtle:

class Turtle()
 |
 | When a Turtle object is created or a function derived from some
 | Turtle method is called a TurtleScreen object is automatically created.
 |
 |
 |
 |
 |
 |
 | Methods defined here:
 |
 | __init__(self, shape='classic', undobuffersize=1000, visible=True)
 |     Initialize self.  See help(type(self)) for accurate signature.
```

Запрашиваем справку по функции `turtle.Turtle()`

Это явно не функция

Класс?

Методы?

Объект?

Не самое понятное описание, но ничего страшного

Мы показали только основное, чтобы сберечь пару деревьев

Ознакомившись с несколько запутанным описанием, мы видим, что `Turtle` — это либо класс, либо объект, либо то и другое (чем бы оно ни было), причем обладающий некими методами. Но что такое класс и объект? Пришло время сказать, что Python (как и большинство современных языков) — *объектно-ориентированный* язык программирования. Это означает, что нам пора наконец поближе познакомиться с классами, объектами и методами.

Что такое объект?

Все мы имеем представление об объектах, поскольку сталкиваемся с ними каждый день. Автомобили, смартфоны, холодильники и даже кухонная посуда — все это объекты. Каждый из них имеет определенные *характеристики* (внутреннее состояние) и обладает неким *поведением*. Возьмем, к примеру, автомобиль. Вот его характеристики:

- производитель;
- модель;
- расход топлива;
- скорость;
- пробег;
- состояние двигателя (запущен/остановлен).

Это лишь то, что пришло нам в голову; вы наверняка придумаете что-то получше

У автомобиля также есть поведение. Он может:

- заводиться;
- глушить мотор;
- ехать;
- тормозить.

Все это в равной степени относится и к программным объектам. Основное преимущество объектов перед другими программными структурами заключается в том, что их состояние и поведение *связаны*. Например, у булевого значения есть состояние, но нет поведения, а у функции есть поведение, но нет состояния. Объекты Python позволяют сопоставлять состояние с поведением. Например, заводя машину (`car.start()`), вы переводите ее двигатель из остановленного состояния (`car.engine_on = False`) в запущенное (`car.engine_on = True`). Аналогичным образом, если автомобиль начинает тормозить (`car.brake()`), то такая его характеристика, как скорость (`car.speed`), начинает падать.

Что же тут такого, спросите вы? Ведь все то же самое можно делать с помощью переменных и функций. Просто когда вы начинаете мыслить в категориях объектов, вы переходите к решению задачи на более высоком уровне: вы рассматриваете программу как набор взаимодействующих объектов, а не как нагромождение переменных и функций.

Черепашки в этом смысле — хороший пример. Можно написать громоздкий код для управления их координатами и цветами, а можно использовать объекты `turtle`, которые самостоятельно управляют своим состоянием, позволяя нам сконцентрироваться на более общих вещах (например, как с помощью двух черепашек нарисовать спираль). Конечно, это упрощенный пример, но он позволяет понять суть объектов.

```
make: 'Chevy'
model: 'Bel Air'
fuel: 8
speed: 0
mileage: 1211
engine_on: False
def start():
def turn_off():
def brake():
```

'57 Chevy

Состояние и поведение в одном объекте

```
make: 'Mini'
model: 'Cooper'
fuel: 2
speed: 14
mileage: 43190
engine_on: True
def start():
def turn_off():
def brake():
```

Mini

Объекты есть не только в Python, но и почти во всех современных языках программирования

```
make: 'Pontiac'
model: 'Fiero'
fuel: 10
speed: 56
mileage: 196101
engine_on: True
def start():
def turn_off():
def brake():
```

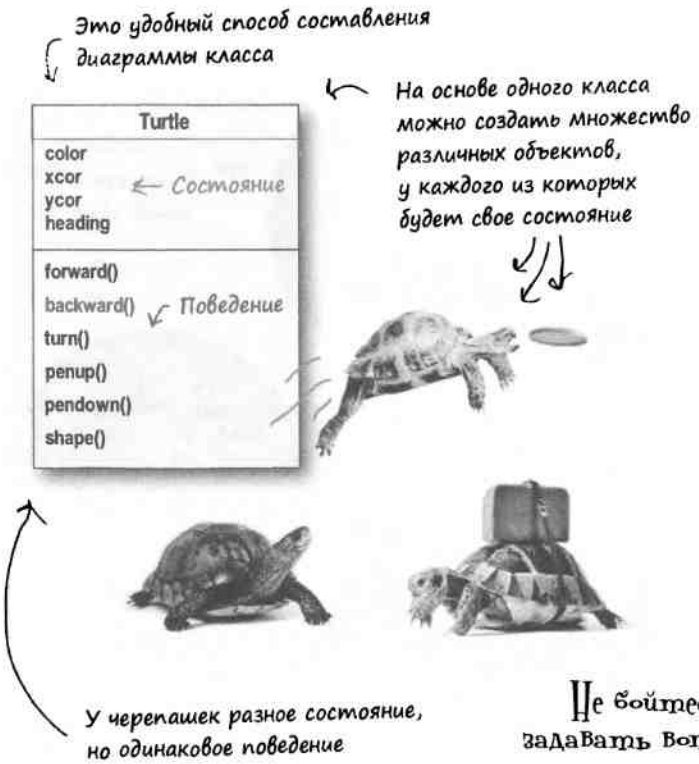
Fiero

```
make: 'DeLorean'
model: 'DMC-12'
fuel: 6
speed: 88
mileage: 10125
engine_on: True
def start():
def turn_off():
def brake():
```

DeLorean

А что тогда класс?

Если планируется создавать большое количество объектов, к примеру, черепашек, то необходимо, чтобы все они имели одинаковое поведение. Никому ведь не хочется всякий раз заново писать функции перемещения. В то же время каждая черепашка должна иметь свое уникальное состояние: местоположение, направление движения, цвет и т.п. Класс — это шаблон или стандарт, по которому создаются однотипные объекты.



Класс — это не объект, а спецификация.

Класс представляет собой шаблон, по которому создаются объекты. Он сообщает Python, как следует создавать объекты данного типа. Каждый объект, созданный на основе класса, будет обладать собственными значениями внутреннего состояния. Например, с помощью класса `Turtle` можно создать дюжину черепашек, и у каждой будет свой цвет, своя форма, свои координаты, свои установки пера (опущено или поднято) и т.п.

При этом все черепашки одного класса будут иметь одинаковое поведение: они будут одинаково перемещаться вперед и назад, поворачивать и управлять пером.

Не бойтесь задавать вопросы

В: Зачем нужно создавать объекты на основе классов?

О: Без классов и объектов все задачи пришлось бы решать с помощью базовых типов данных: чисел, строк, списков и т.п. В объектно-ориентированном программировании мы имеем дело высокоуровневыми типами данных, которые лучше соответствуют специфике решаемой задачи. Например, в симуляторе рыбалки намного проще оперировать объектами рыб и объектом пруда, чем управлять огромным набором переменных и функций.

В: Класс — это тип данных?

О: Да. Каждый класс в Python является отдельным типом, таким как строка или список.

В: Можно ли создавать собственные классы?

О: Конечно! Это одно из преимуществ объектно-ориентированного программирования: любой язык программирования (не только Python) можно расширять собственными классами. Можно даже расширять функциональность классов, созданных другими программистами. В этой главе мы будем работать только с готовыми классами, а о создании новых классов поговорим в главе 12.

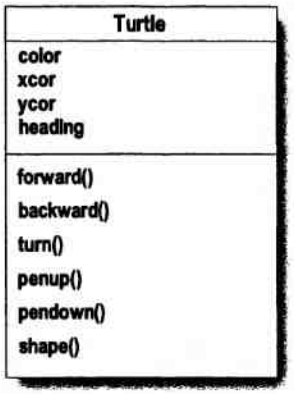
В: В объекте содержится много данных (значения переменных и определения функций). Каким образом обычная переменная может хранить целый объект?

О: Помните, мы говорили о том, как в переменной можно сохранить целый список? Переменная в действительности хранит ссылку на область памяти, в которой находится список, т.е. переменная представляет собой указатель на список. То же самое происходит и в случае объектов. Переменная хранит ссылку на объект, находящийся в памяти, а не сам объект.

Класс определяет характеристики объекта и его поведение

На предыдущей странице была приведена диаграмма класса. Давайте познакомимся с ней поближе. Диаграмма класса показывает две вещи: какие переменные хранятся в объекте и каковы его функциональные возможности.

Это диаграмма класса Turtle



Значения, известные объекту Turtle (состояние)

Действия, выполняемые объектом Turtle (поведение)

Значения, отслеживаемые объектом, известны как

- **атрибуты.**

А выполняемые объектом действия известны как

- **методы.**

Атрибуты описывают состояние или характеристики объекта (его данные); у каждого объекта свои значения атрибутов. Атрибуты во многом напоминают локальные переменные, но объявленные внутри объекта. Как и переменные, атрибуты могут иметь значения любых типов данных Python. В будущем вам не раз встретится термин *переменная экземпляра*. Это то же самое, что и атрибут объекта. По сути, всякий раз, когда вы слышите термин *экземпляр*, можете смело заменять его термином *объект*. Таким образом, переменная экземпляра равнозначна локальной переменной объекта, которая представляет собой атрибут.

Методы определяют действия, которые может выполнять объект. Методы проще всего рассматривать как функции объекта. Метод отличается от функции тем, что работает с атрибутами объекта.

Возьмите карандаш

Заполните диаграмму класса Radio, указав, какие атрибуты и методы ему могут понадобиться.

Использование объектов и классов

В этой главе мы узнаем, как использовать *готовые* объекты и классы. Оказывается, в нашем распоряжении имеется огромное количество классов, созданных другими программистами; их можно свободно применять в коде. Для этого следует знать о том, какие действия умеет выполнять выбранный вами объект (иными словами, его методы) и какие атрибуты он поддерживает. И естественно, нужно знать, как создавать такой объект (по-другому называемый *экземпляром*). Нам уже доводилось создавать объекты `Turtle`, но давайте рассмотрим процедуру еще раз.

О создании собственных классов и объектов мы поговорим в главе 12

Модуль `turtle`

Оператор-точка

Имя класса `Turtle`

```
slowpoke = turtle.Turtle()
```

Это вызов внутреннего метода инициализации класса `Turtle`, который создает новый объект и настраивает его состояние

Результатом вызова метода инициализации становится новый объект `Turtle`, созданный на основе спецификации класса `Turtle`. Новый объект `Turtle` записывается в переменную `slowpoke`

В объектно-ориентированном программировании объекты порождаются, а каждый объект называется экземпляром

В этой короткой строке кода содержится много важных сведений. Изучим ее детальнее. Во-первых, в ней указывается, что объект создается на основе класса `Turtle` модуля `turtle`. Именно для этого применяется оператор-точка, который в данном случае не связан с обращением к атрибуту или методу.

```
turtle.Turtle
```

Обратите внимание на то, что по соглашению имя класса пишется с прописной буквы (далее будет показано несколько исключений из этого правила)

Здесь мы получаем класс `Turtle` из модуля `Turtle`

В объектно-ориентированном программировании такие методы инициализации объектов называются конструкторами. Услышав термин "конструктор", будьте уверены: речь идет о методе, с помощью которого инициализируется объект

Далее мы вызываем класс, как будто это функция. Как такое возможно?

```
turtle.Turtle()
```

Класс вызывается как функция?

У каждого класса есть специальный метод, называемый *конструктором*. Этот метод получает объект, в котором все необходимые атрибуты установлены в значения по умолчанию. У конструктора важное назначение: он возвращает созданный им объект (называемый *экземпляром*).

Созданный нами объект сохраняется в переменной `slowpoke`.

```
slowpoke = turtle.Turtle()
```

Возвращаем новый объект, созданный по спецификации класса `Turtle`

У конструкторов тоже могут быть аргументы, как будет показано далее

Что такое атрибуты и методы?

Получив в свое распоряжение объект, можно приступать к вызову его методов.

```
slowpoke.turn(90)
slowpoke.forward(100)
```

Для вызова метода укажите имя объекта, точку и затем имя метода. Другими словами, метод вызывается так же, как и функция, только с указанием имени объекта

При обращении к методу всегда нужно указывать имя объекта. Без этого метод не будет выполнен (он не будет знать, к каким переменным состояния обращаться)

А как насчет атрибутов? Нам еще не доводилось работать с кодом, в котором запрашиваются или устанавливаются значения атрибутов. Для доступа к атрибуту объекта применяется уже известный вам точечный синтаксис. Предположим, что у класса `Turtle` есть атрибут `shape` (в реальности его нет, но условимся, что это так). Тогда для получения или установки значения такого атрибута можно использовать следующий синтаксис.

```
slowpoke.shape = 'turtle'
print(slowpoke.shape)
```

Получить или задать значение атрибута объекта можно с помощью оператора-точки (синтаксис точно такой же, как и при обращении к переменной модуля)

Если бы такой атрибут существовал, программа вывела бы `'turtle'`

Хорошо, но почему тогда в более ранних примерах форма черепашки задавалась с помощью метода `shape()` объекта `Turtle`? Почему бы не использовать для этих целей атрибут `shape`? Мы вполне могли бы так поступить, но в объектно-ориентированном программировании принято, чтобы для получения и задания значений атрибутов применялись методы. Это называется *инкапсуляцией*, о чем мы поговорим в главе 12. Инкапсуляция обеспечивает более строгий контроль над состоянием объектов, предотвращая произвольное изменение их атрибутов.

А пока просто помните о том, что для доступа ко многим атрибутам объектов используются методы; обратиться к ним напрямую нельзя. Ниже приведен пример: для задания и получения атрибута `shape` применяется метод `shape()`.

Можно вызвать метод `shape()`, чтобы изменить внутренний атрибут `shape`, сделав черепашку круглой

```
slowpoke.shape('circle')
print(slowpoke.shape())
```

Если вызвать метод `shape()` без аргументов, то будет получено текущее значение атрибута

Объекты и классы повсюду

Вооружившись знаниями о классах и объектах, можно по-новому взглянуть на синтаксис Python. Мы уже говорили о том, что Python — объектно-ориентированный язык, и, по большому счету, объекты в нем повсюду. Судите сами.

В Python список — это класс

Здесь мы вызываем конструктор списка для создания нового, пустого списка

```
my_list = list()
my_list.append('первый')
my_list.append('второй')
my_list.reverse()
print(my_list)
```

Вызываем метод append()...

...а затем метод reverse(), который ранее нам не встречался

Все эти методы меняют внутреннее состояние объекта списка

Оболочка Python

```
['второй', 'первый']
>>>
```

Помните, мы говорили о том, что имя класса всегда начинается с прописной буквы? К сожалению, это не относится к некоторым встроенным классам Python (по историческим причинам). Так что учитывайте, что имена отдельных встроенных классов начинаются со строчной буквы!

Следующий пример класса должен быть вам хорошо знаком.

Оказывается, строки — это тоже классы. Вот как создать экземпляр строки с помощью конструктора

```
greeting = str('Привет, читатель!')
shout = greeting.upper()
print(shout)
```

Заметьте, что у конструктора есть аргумент

Вызов одного из строковых методов

Оболочка Python

```
ПРИВЕТ, ЧИТАТЕЛЬ!
>>>
```

А как насчет такого синтаксиса?

Поскольку списки, строки и числа с плавающей точкой являются встроенными объектами Python, не требуется в явном виде вызывать конструктор класса, как в данном случае. Python обрабатывает такие значения самостоятельно

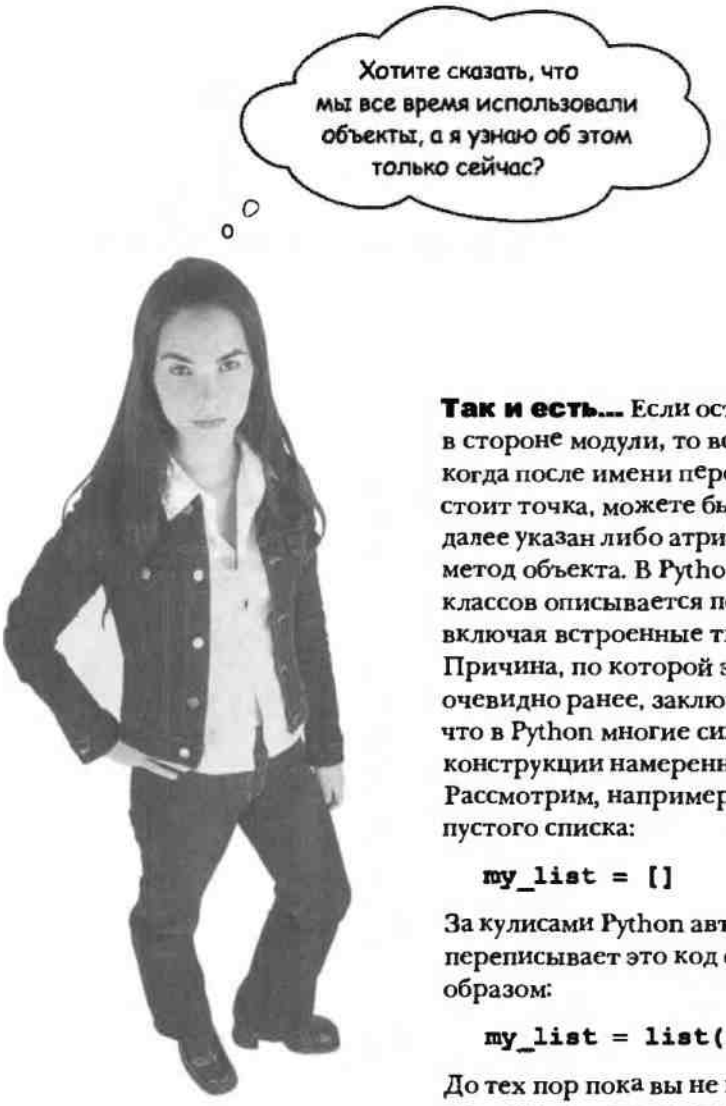
Все правильно, еще один класс...

```
pi = float(3.1415)
is_int = pi.is_integer()
print(pi, is_int)
```

...и еще один метод

Оболочка Python

```
3.1415 False
>>>
```



Так и есть... Если оставить в стороне модули, то всякий раз, когда после имени переменной стоит точка, можете быть уверены: далее указан либо атрибут, либо метод объекта. В Python с помощью классов описывается почти все, включая встроенные типы данных. Причина, по которой это не было очевидно ранее, заключается в том, что в Python многие синтаксические конструкции намеренно упрощены. Рассмотрим, например, создание пустого списка:

```
my_list = []
```

За кулисами Python автоматически переписывает это код следующим образом:

```
my_list = list()
```

До тех пор пока вы не начинаете применять методы списков, вы не догадываетесь, что имеете дело с объектами:

```
my_list.append(42)
```

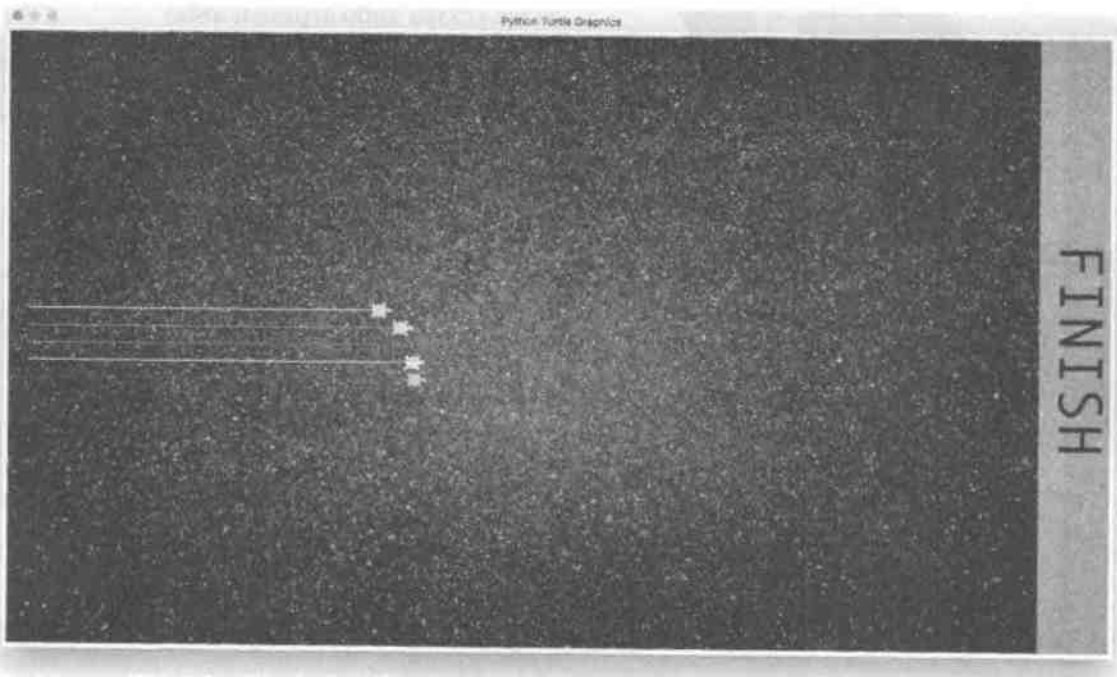
Теперь, когда все прояснилось, помните, что в Python объекты повсюду. Пора начать применять их на практике.

Черепашьи бега

Как вы уже знаете, основным преимуществом объектов является то, что каждый из них хранит собственное состояние, притом что все объекты одного класса имеют одинаковое поведение (методы).

Вам **также** известно, что каждая черепашка представлена собственным объектом – отдельным экземпляром, обладающим своим набором атрибутов. Это означает, что у каждой черепашки уникальные характеристики: цвет, положение, направление движения, форма и др. Давайте на основе имеющейся информации создадим простую игру. Вам доводилось наблюдать черепашьи бега?

Мы создадим группу черепашек, у каждой из которых будет свой цвет и своя позиция, и заставим их гонять по экрану. Делайте ставки!



Линия старта будет у левого края окна

Черепашки будут мчаться к правому краю окна. Победит та из них, которая первой пересечет финишную черту

Планирование игры

Если рассматривать черепашек как объекты, то процесс разработки игры существенно упростится. Без объектов нам пришлось бы создавать и обновлять огромное количество переменных для отслеживания позиций черепашек, что очень утомительно. При работе с объектами `Turtle` для перемещения черепашек по экрану достаточно использовать встроенные методы, ведь каждая черепашка хранит собственное состояние.

Для начала запишем псевдокод игры.

Сначала необходимо создать черепашек и назначить каждой из них собственный цвет и позицию на стартовом поле. Каждая черепашка — это отдельный экземпляр класса `Turtle`, так что мы можем это сделать!

1. **Начать игру.**
   Создать несколько черепашек, каждую со своим цветом и позицией на стартовом поле.

Гонка будет продолжаться до тех пор, пока одна из черепашек не победит, достигнув финишной черты. Но нужен алгоритм определения победителя. Создадим переменную `winner` и первоначально установим ее равной `False`, присвоив ей значение `True`, только когда черепашка достигнет финишной черты

2. **Начать гонку.**
   Задать переменную **winner** равной False.

   Пока переменная **winner** равна False:

   A. Для каждой черепашки:

Теперь будем циклично переключаться между черепашками, заставляя их двигаться до тех пор, пока одна из них не победит

      i. Выбрать случайную величину продвижения вперед, скажем, от 0 до 2 пикселей.

      Передвинуться вперед.

Генерируем случайное число от 0 до 2 и перемещаем черепашку на соответствующее число пикселей вперед. Это делается для каждой черепашки

      ii. Проверить, не пересекла ли черепашка финишную черту. Если да, то:

         a. Задать переменную **winner** равной True.

При каждом перемещении необходимо проверять, не пересекла ли черепашка финишную черту. Если это произошло, задаем переменную `winner` равной `True`

3. **Игра завершается.**
   Объявляем победителя!

Как только переменная `winner` становится равной `True`, цикл завершается, и мы объявляем имя победителя

Приступаем к написанию кода игры

В первую очередь нужно создать игровое поле и всех принимающих участие в забеге черепашек, после чего можно приступать к реализации игровой логики. Мы уже знаем, какие модули нам понадобятся: `turtles` и `random`, поэтому сразу их импортируем. Нам также нужна глобальная переменная для хранения списка участников забега. Создайте новый файл *race.py* и введите в него следующий код.

```
import turtle
import random

turtles = list()
```

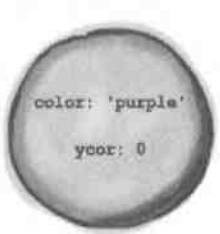
Нам нужны эти модули

Мы используем список для хранения всех черепашек. Чтобы закрепить в памяти синтаксис конструктора, мы создадим пустой список путем вызова конструктора списка. Можно также использовать сокращение [], как это делалось ранее

Начальная настройка

Согласно приведенному выше псевдокоду настройка игры заключается в создании группы черепашек, имеющих собственные атрибуты. Но прежде необходимо понять, какие именно черепашки нам нужны.

Атрибуты `xcor` и `ycor` определяют координаты (x, y) черепашки в окне



Пурпурная

Красная

Коричневая

Синяя

Зеленая

Turtle
color xcor ycor heading
forward() backward() turn() penup() pendown() shape()

Это объекты, порожденные от класса `Turtle`

У каждой черепашки есть имя, цвет и координата `ycor`, определяющая ее вертикальную позицию на стартовом поле. Все остальные атрибуты тоже имеются, но нам понадобятся только эти

Код настройки игры

Теперь напишем код создания и инициализации объектов черепашек. На предыдущей странице были перечислены атрибуты черепашек — всех их нужно где-то хранить, чтобы можно было выполнить инициализацию черепашек. Для этого нам понадобятся два списка атрибутов: в первом будут указываться координаты y, а во втором — цвета. В результате при создании объектов черепашек их атрибутам можно будет присваивать необходимые значения из соответствующих списков. Разумеется, начальную настройку лучше всего выполнять в отдельной функции.

1 Начать игру.
Создать несколько черепашек, каждую со своим цветом и позицией на стартовом поле.

```
import turtle
import random

turtles = list()

def setup():
    global turtles
    startline = -480

    turtle_ycor = [-40, -20, 0, 20, 40]
    turtle_color = ['blue', 'red', 'purple', 'brown', 'green']

    for i in range(0, len(turtle_ycor)):
        new_turtle = turtle.Turtle()
        new_turtle.shape('turtle')
        new_turtle.setpos(startline, turtle_ycor[i])
        new_turtle.color(turtle_color[i])
        turtles.append(new_turtle)

setup()
turtle.mainloop()
```

Определим функцию `setup()` для создания и позиционирования черепашек

Переменная `startline` хранит координату x линии старта

В следующей главе мы изучим структуру данных, которая лучше подходит для хранения этих значений, чем два списка

Это начальные значения для атрибутов каждой черепашки, хранимые в двух параллельных списках

Создаем цикл по всем черепашкам

Для каждой черепашки создаем новый объект `Turtle`, устанавливаем для него форму `'turtle'` и задаем позицию на стартовом поле

Назначаем черепашке цвет

Добавляем новую черепашку в глобальный список

Метод `setpos()` задает координаты (x, y) черепашки. В данном случае мы перемещаем черепашку на стартовое поле

Не забудьте вызвать функцию `setup()`, а после нее — функцию `turtle.mainloop()`

Помните метод `append()` из главы 4? Он добавляет элемент в существующий список, в данном случае это список `turtles`. Только в главе 4 мы не использовали термин 'метод'

Спешка ни к чему!

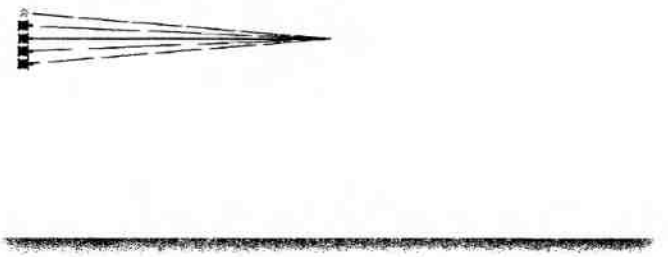
Приведенный на предыдущей странице код не настолько прост, как кажется. Мы, конечно, могли бы проанализировать его строка за строкой, но ведь это уже глава 7, и вас наверняка интересуют вещи посерьезнее. В любом случае не торопитесь, внимательно изучите код и убедитесь, что понимаете каждую его строку, прежде чем двигаться дальше.

Тест-драйв

Обновите код игры в файле *race.py* и протестируйте его.

В вашей системе стандартное окно может оказаться меньшего размера, чем показано здесь. Если черепашки выпадают за левый край окна, растяните его, пока не увидите их

Шаг в правильном направлении, но все выглядит немного странно. Что произошло?



Все черепашки правильного цвета и находятся в правильных позициях. Но вспомните, что их жизнь начинается в точке с координатами (0, 0) в центре окна, и в процессе их перемещения к стартовым позициям они нарисовали линии, что совсем не нужно. Необходимо добавить код для начального поднятия пера

Раз уж мы заговорили об интерфейсе игры, необходимо сделать окно немного крупнее и задать другой фон

Придется вносить изменения в код. В начальной позиции перо черепашки должно быть поднято, чтобы она не оставляла след при перемещении к стартовому полю. Кроме того, мы увеличиваем окно и добавляем в него фоновое изображение. Обновите программу и протестируйте ее снова. Файл *pavement.gif* следует скопировать из папки материалов к главе 7 и поместить в ту же папку, что и файл *race.py*.

```
import turtle
import random

turtles = list()

def setup():
    global turtles
    startline = -620
    screen = turtle.Screen()
    screen.setup(1290, 720)
    screen.bgpic('pavement.gif')

    turtle_ycor = [-40, -20, 0, 20, 40]
    turtle_color = ['blue', 'red', 'purple', 'brown', 'green']

    for i in range(0, len(turtle_ycor)):
        new_turtle = turtle.Turtle()
        new_turtle.shape('turtle')
        new_turtle.penup()
        new_turtle.setpos(startline, turtle_ycor[i])
        new_turtle.color(turtle_color[i])
        new_turtle.pendown()
        turtles.append(new_turtle)

setup()
turtle.mainloop()
```

Не упустите это изменение. Необходимо сместить черепашек дальше влево. Помните: центр окна имеет координаты (0, 0), поэтому координата x, равная -620, находится ближе к левому краю окна

В этих трех строках кода используется не рассматривавшийся ранее объект Screen. С его помощью можно сделать окно больше и поменять его фон

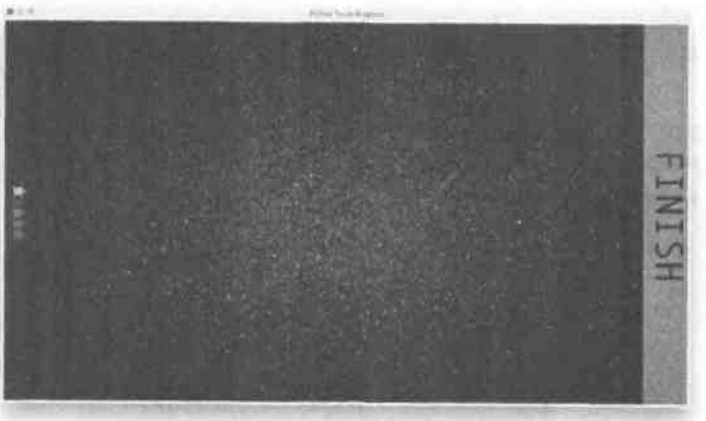
Поднимем перо, прежде чем перемещаться

И опустим его вниз, когда черепашка окажется на стартовом поле

Вы уже сейчас наверняка сможете понять, что делает этот код, но детальнее мы рассмотрим объект Screen в главе 10

Финишная черта является частью фонового изображения

Вот что у нас получилось! Все черепашки стоят на старте и готовы к гонке. Теперь можно начинать игру



У нас имеются пять объектов Turtle, каждый со своим внутренним состоянием, включая цвет и позицию

Начало забега

Пора начинать забег. Все необходимые инструкции приведены в псевдокоде, осталось реализовать их.

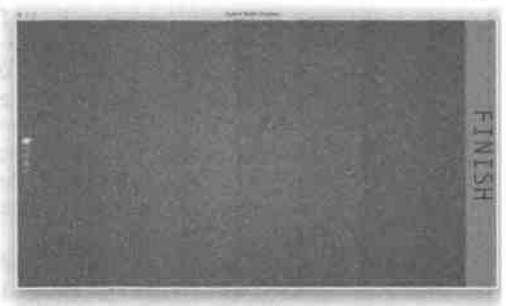
2 Начать гонку.
Задать переменную **winner** равной False.
Пока переменная **winner** равна False:
A Для каждой черепашки:
i Выбрать случайную величину продвижения вперед, скажем, от 0 до 2 пикселей.
Передвинуться вперед.
ii Проверить, не пересекла ли черепашка финишную черту. Если да, то:
a Задать переменную **winner** равной True.

2 **Сначала создаем переменную `winner`.** Мы сделаем это в новой функции, которую назовем `race()`. Переменная `winner` должна быть булевой и инициализироваться значением `False`. В функции `race()` также будет использоваться глобальная переменная `turtles`, которую нужно объявить соответствующим образом.

```
def race():
    global turtles
    winner = False
```

Кроме того, добавим в функцию локальную переменную, задающую координату *x* финишной черты.

Координата 590 на оси x находится здесь




```
def race():
    global turtles
    winner = False
    finishline = 590
```

Нужно сделать так, чтобы забег продолжался до выявления победителя. Для этого создаем цикл `while`, который будет выполняться, пока переменная `winner` не станет равна `True`.

```
def race():
    global turtles
    winner = False
    finishline = 590

    while not winner:
```

Цикл выполняется до тех пор, пока переменная winner не станет равна True

A **Далее нужно реализовать перемещение черепашек.** Сначала создаем цикл `for/in` для перебора глобального списка `turtles`.

```
def race():
    global turtles
    winner = False
    finishline = 590

    while not winner:
        for current_turtle in turtles:
```

На каждой итерации цикла while мы проходим по списку черепашек, давая им возможность переместиться по экрану

i **Сгенерируем случайные числа, задающие расстояния, на которые должны переместиться черепашки.** Каждое случайное число выбирается из диапазона от 0 до 2 – именно на такое количество пикселей смещается черепашка.

```
def race():
    global turtles
    winner = False
    finishline = 590

    while not winner:
        for current_turtle in turtles:
            move = random.randint(0,2)
            current_turtle.forward(move)
```

Сгенерируем случайное число от 0 до 2 и переместим черепашку на соответствующее число пикселей вперед

ii **Наконец, нужно определить победителя.** Заметьте, что победителем становится черепашка, атрибут `xcor` которой больше или равен переменной `finishline`, т.е. `590`. Таким образом, условие проверки победителя оказывается очень простым. Если одна из черепашек пересекла финишную черту, присваиваем переменной `winner` значение `True` и объявляем победителя.

```
def race():
    global turtles
    winner = False
    finishline = 590

    while not winner:
        for current_turtle in turtles:
            move = random.randint(0, 2)
            current_turtle.forward(move)

            xcor = current_turtle.xcor()
            if (xcor >= finishline):
                winner = True
                winner_color = current_turtle.color()
                print('Победитель --', winner_color[0])
```

Определяем координату x черепашки, чтобы понять, пересекла ли она финишную черту. Для этого используем метод `xcor()`, возвращающий координату x объекта

Сравниваем ее с координатой финиша

Если координата x больше, у нас есть победитель

Задаем переменную `winner` равной `True`

Используем метод `color()`, чтобы определить цвет черепашки-победителя

Учтите, что метод `color()` возвращает два значения: цвет пера и цвет заливки. Нас интересует первое, поэтому используем индекс 0

3 Игра завершается. Объявляем победителя!

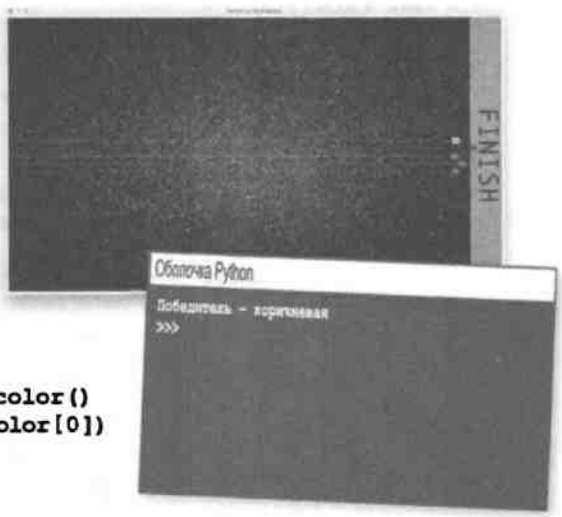
Игра готова — можно провести тестовый забег. Внесите описанные выше изменения в файл *race.py*, код которого приведен ниже, и запустите программу, чтобы определить победителя.

```
import turtle
import random

turtles = list()

def setup():
    global turtles
    startline = -620
    screen = turtle.Screen()
    screen.setup(1290, 720)
    screen.bgpic('pavement.gif')

    turtle_ycor = [-40, -20, 0, 20, 40]
    turtle_color = ['blue', 'red', 'purple', 'brown', 'green']

    for i in range(0, len(turtle_ycor)):
        new_turtle = turtle.Turtle()
        new_turtle.shape('turtle')
        new_turtle.penup()
        new_turtle.setpos(startline, turtle_ycor[i])
        new_turtle.color(turtle_color[i])
        new_turtle.pendown()
        turtles.append(new_turtle)

def race():
    global turtles
    winner = False
    finishline = 590

    while not winner:
        for current_turtle in turtles:
            move = random.randint(0, 2)
            current_turtle.forward(move)

            xcor = current_turtle.xcor()
            if (xcor >= finishline):
                winner = True
                winner_color = current_turtle.color()
                print('Победитель -', winner_color[0])

setup()
race()
turtle.mainloop()
```

Не забывайте начать гонку!

Не забывайте, что величина каждого перемещения определяется случайным образом, поэтому результаты всегда будут разными

Коричневая победила!



Сила мысли

Предположим, что на одной и той же итерации цикла `while` обнаруживаются сразу два победителя. Что произойдет при этом? Кто будет объявлен победителем? Можно ли считать такое поведение корректным?

Кто что делает?

В объектно-ориентированном программировании свой жаргон. В этом упражнении вам нужно сопоставить приведенные ниже термины с их определениями.

класс	Данные, хранящиеся в объекте
объект	Шаблон объекта
метод	Создание экземпляра класса
порождение	Действие, выполняемое объектом
атрибут	Создается по шаблону
экземпляр	Другое название объекта

Все ведь только началось, и вот глава уже подошла к концу.

Не печальтесь, тема объектов не закончена

Все верно, это была лишь вершина айсберга. Объектно-ориентированное программирование – гигантская тема, достойная отдельной книги, и данная глава служит лишь введением в тему. В следующий раз, столкнувшись с **классом** в модуле, вы сможете создать его **объект** с помощью **конструктора**. Вы будете знать, что у объекта есть **атрибуты** и **методы**, доступные для использования в коде. Причем у каждого **экземпляра** объекта имеется собственный набор атрибутов.

Помните о том, что в Python можно получить справку по классу с помощью команды наподобие `help(Turtle)`. *В случае команды типа* `help(turtle)` *выдается справка по указанному модулю. Только не забудьте сначала импортировать модуль! Мы продолжим изучать классы и объекты в последующих главах*

Из главы вы узнали, что все типы данных в Python в действительности являются классами. В дальнейшем нам предстоит много работать с классами и объектами. В главе 12 будет показано, как создавать собственные классы. Встретимся там!

И кстати, это еще не конец главы. Вас ждет детективная история, после чего мы подведем итоги и решим кроссворд.

Ого! Какая шустрая зеленая черепашка!

Странные события на черепашьих бегах

После официального релиза программы начали происходить странные вещи: зеленая черепашка стала все время побеждать, причем с огромным преимуществом. Полиция подозревает, что кто-то хакнул код. Не поможете разобраться, в чем там дело?

```
import turtle
import random

turtles = list()

class SuperTurtle(turtle.Turtle):
    def forward(self, distance):
        cheat_distance = distance + 5
        turtle.Turtle.forward(self, cheat_distance)

def setup():
    global turtles
    startline = -620
    screen = turtle.Screen()
    screen.setup(1290,720)
    screen.bgpic('pavement.gif')

    turtle_ycor = [-40, -20, 0, 20, 40]
    turtle_color = ['blue', 'red', 'purple', 'brown', 'green']

    for i in range(0, len(turtle_ycor)):
        if i == 4:
            new_turtle = SuperTurtle()
        else:
            new_turtle = turtle.Turtle()
        new_turtle.shape('turtle')
        new_turtle.penup()
        new_turtle.setpos(startline, turtle_ycor[i])
        new_turtle.color(turtle_color[i])
        new_turtle.pendown()
        turtles.append(new_turtle)

def race():
    global turtles
    winner = False
    finishline = 590

    while not winner:
        for current_turtle in turtles:
            move = random.randint(0,2)
            current_turtle.forward(move)

            xcor = current_turtle.xcor()
            if (xcor >= finishline):
                winner = True
                winner_color = current_turtle.color()
                print('Победитель --', winner_color[0])

setup()
race()

turtle.mainloop()
```

Внимательно изучите код. Что в нем изменилось? Что делает новый код? Полностью разобраться в нем вы сможете только в главе 12, но все же попробуйте понять, как он работает.

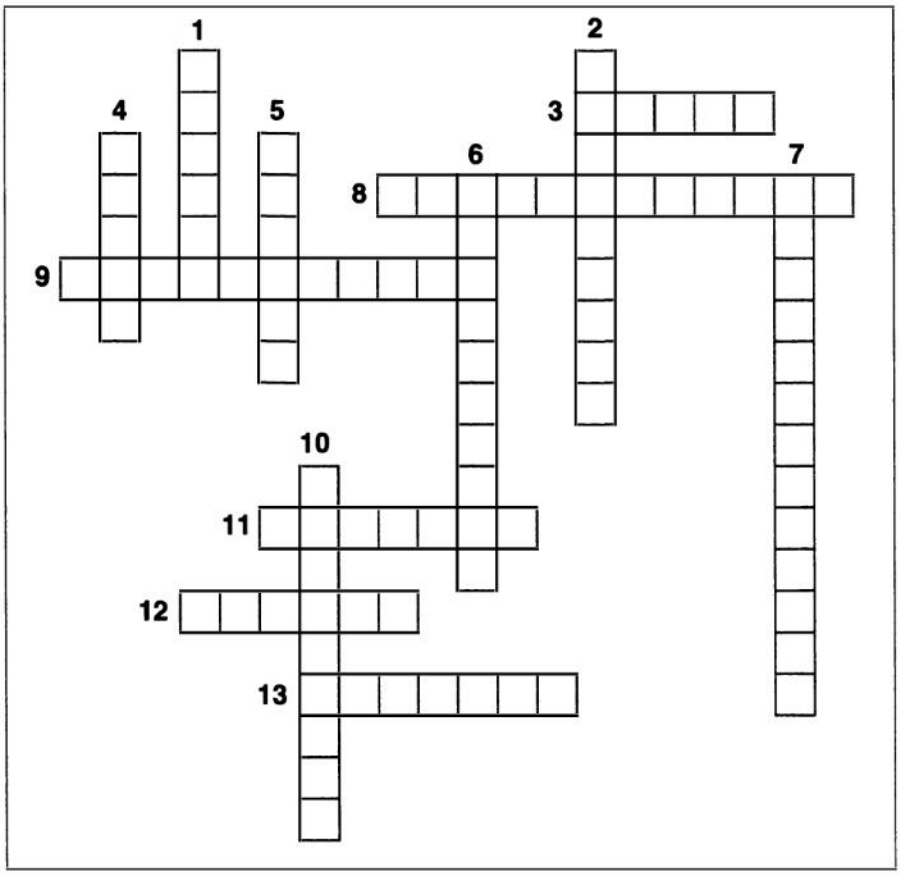
CRIME SCENE DO NOT ENTER CRIME SCENE DO NOT ENTER CRIMESCENE DO NOT ENTER

САМОЕ ГЛАВНОЕ

- В модулях хранятся коллекции переменных, функций и классов.
- Переменная `__name__` позволяет определить, импортируется ли модуль или запускается как отдельная программа (в этом случае она имеет значение `"__main__"`).
- С помощью функции `help()` можно ознакомиться с документацией к функциям, модулям и классам в оболочке Python.
- Для встраивания справочных сведений в программу применяются документирующие строки.
- В Python имеется множество модулей самого разного предназначения: математические функции, разработка пользовательских интерфейсов, веб-службы, функции даты/времени и т.п.
- Модуль `turtle` реализует графическую систему анимируемых черепашек, первоначально разработанную в Массачусетском технологическом институте.
- Объекты `turtle` создаются на координатной сетке. Они умеют перемещаться и рисовать.
- Черепашки — это объекты Python, имеющие собственные данные и поведение.
- Данные, хранимые в объектах Python, называются *атрибутами*.
- Атрибут может иметь любой тип данных, поддерживаемый в Python.
- Поведение объекта задается его методами.
- Метод — это функция, вызываемая по отношению к объекту.
- Для доступа к атрибутам и методам применяется точечная нотация.
- Объекты создаются на основе классов. Класс задает спецификацию объектов данного типа.
- Когда мы создаем новый объект, мы порождаем его.
- Созданный объект называется *экземпляром*.
- Объект создается с помощью конструктора — специального метода, определенного в классе.
- В конструкторе выполняются все действия, связанные с инициализацией объекта.
- Все типы данных в Python являются классами, включая числа, строки, списки и пр.

Кроссворд

Чтобы закрепить тему модулей, попробуйте решить кроссворд.



По горизонтали

3. Шаблон объектов
8. Справка к модулю
9. Метод, отвечающий за создание объекта
11. Цвет одной из черепашек
12. Структура данных, хранящая состояние и поведение
13. Значение, хранимое в объекте

По вертикали

1. Единица измерения, задающая угол поворота черепашки
2. Объект класса
4. Функция, задающая поведение объекта
5. Одно из направлений движения черепашек
6. Цвет одной из черепашек
7. Начальное конфигурирование объекта
10. Тип объекта, придуманный, чтобы помочь в обучении программированию

Возьмите карандаш
Решение

Позади уже почти семь глав книги, поэтому вам должно быть вполне по силам следующее упражнение. Преобразуйте приведенный выше код рисования квадрата в отдельную функцию, которая называется `make_square()` и имеет единственный аргумент-черепашку. Постарайтесь также удалить весь повторяющийся код.

```
import turtle

slowpoke = turtle.Turtle()
slowpoke.shape('turtle')

def make_square(the_turtle):
    the_turtle.forward(100)
    the_turtle.right(90)
    the_turtle.forward(100)
    the_turtle.right(90)
    the_turtle.forward(100)
    the_turtle.right(90)
    the_turtle.forward(100)
    the_turtle.right(90)

make_square(slowpoke)

turtle.mainloop()
```

Определяем функцию `make_square()`

И вызываем ее

Шаг 1: поместить код в функцию и вызвать ее

Обратите внимание на то, что в функцию `make_square()` можно передать любую черепашку, а не только `slowpoke`

```
import turtle

slowpoke = turtle.Turtle()
slowpoke.shape('turtle')

def make_square(the_turtle):
    for i in range(0, 4):
        the_turtle.forward(100)
        the_turtle.right(90)

make_square(slowpoke)

turtle.mainloop()
```

Нам не нужен дублирующийся код, достаточно создать цикл по четырем элементам

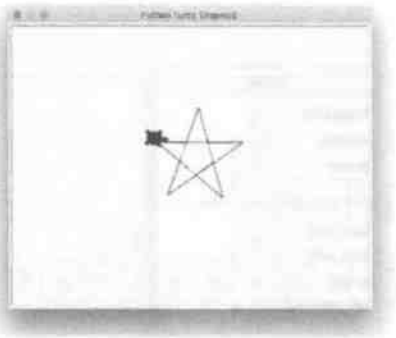
Шаг 2: устранить дублирование кода в функции, создав цикл для вызова методов `forward()` и `right()` нужное число раз

ДРУГИЕ ОПЫТЫ С ЧЕРЕПАШКАМИ РЕШЕНИЯ

Мы подготовили для вас еще несколько экспериментов с черепашками. Внимательно изучите каждый из них, подумайте, что будет делать черепашка, а затем запустите код, чтобы проверить свои предположения. Попробуйте поменять аргументы функций. Как при этом изменится результат на экране?

Эксперимент #1

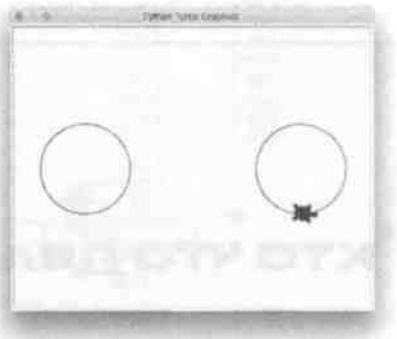
```
for i in range(5):
    slowpoke.forward(100)
    slowpoke.right(144)
```



Вот наш результат. А у вас получилось нарисовать что-нибудь необычное, подбирая значения аргументов?

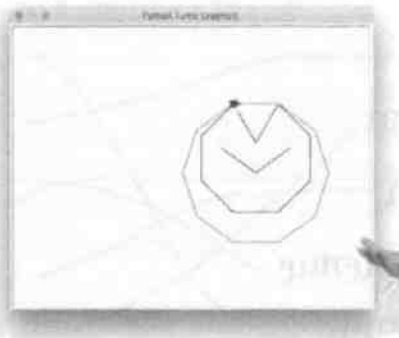
Эксперимент #2

```
slowpoke.pencolor('blue')
slowpoke.penup()
slowpoke.setposition(-120, 0)
slowpoke.pendown()
slowpoke.circle(50)

slowpoke.pencolor('red')
slowpoke.penup()
slowpoke.setposition(120, 0)
slowpoke.pendown()
slowpoke.circle(50)
```



Эксперимент #3

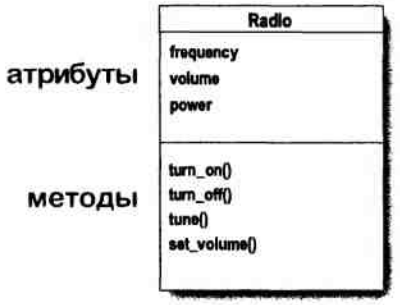
```
def make_shape(t, sides):
    angle = 360/sides
    for i in range(0, sides):
        t.forward(100)
        t.right(angle)

make_shape(slowpoke, 3)
make_shape(slowpoke, 5)
make_shape(slowpoke, 8)
make_shape(slowpoke, 10)
```



Возьмите карандаш
Решение

Заполните диаграмму класса `Radio`, указав, какие атрибуты и методы ему могут понадобиться.

Ответов в данном случае может быть множество. Это один из вариантов

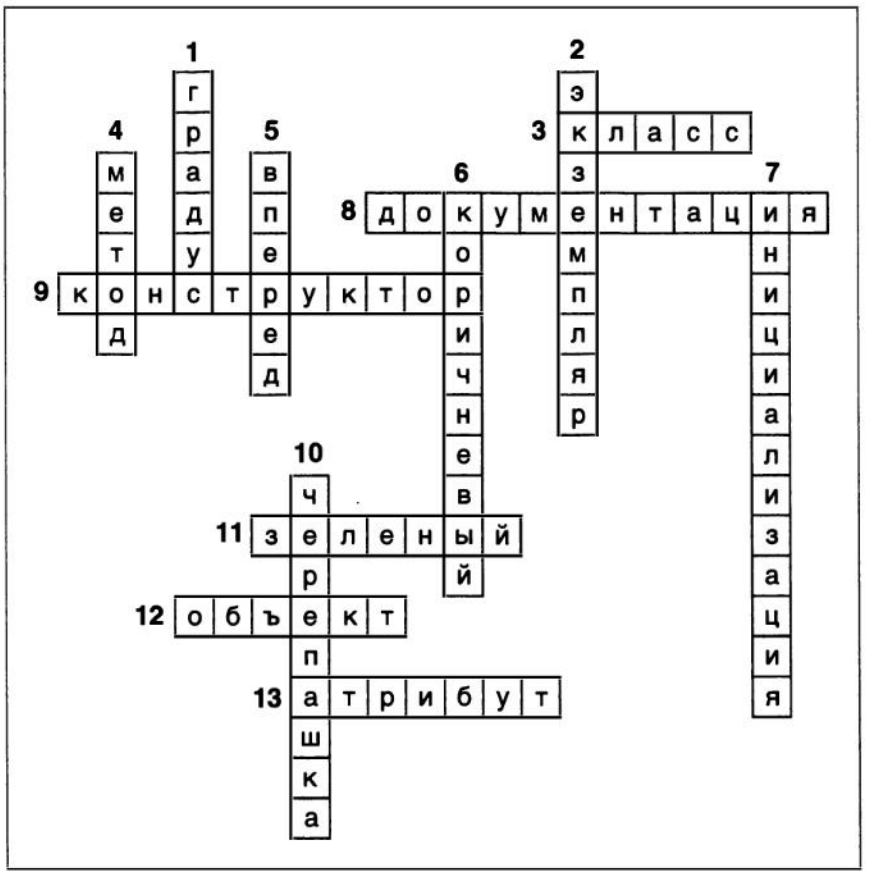


Кто что делает? Решение

В объектно-ориентированном программировании свой жаргон. В этом упражнении вам нужно сопоставить приведенные ниже термины с их определениями.

класс	Данные, хранящиеся в объекте
объект	Шаблон объекта
метод	Создание экземпляра класса
порождение	Действие, выполняемое объектом
атрибут	Создается по шаблону
экземпляр	Другое название объекта

Кроссворд: решение



8 рекурсия и словари

И снова об индексах и циклах

Настало время перейти на новый уровень. Пока что мы придерживались итеративного стиля программирования, создавая структуры данных наподобие списков, строк и диапазонов чисел и обрабатывая их в циклах поэлементно. В этой главе мы пересмотрим подход к программированию и выбору структур данных. Наш новый стиль программирования будет требовать написания *рекурсивного*, т.е. вызывающего самого себя, кода. Кроме того, вы познакомитесь с новым типом данных — *словарем*, который больше напоминает ассоциативный массив, чем список. Воспользовавшись новыми знаниями, мы научимся эффективно решать множество задач. Заранее предупредим: это достаточно сложные темы. Пройдет какое-то время, прежде чем вы освоите все нюансы, но поверьте, оно того стоит.

Иной стиль программирования

Нам придется поменять способ мышления, так как мы слишком зациклились на итеративном стиле программирования. Нужно освоить совершенно иной подход к решению задач.

Но сначала давайте возьмем простую задачу и попробуем решить ее так, как это делалось в предыдущих главах. Предположим, имеется список чисел, которые нужно просуммировать. Числа могут означать что угодно, например количество шариков в карманах у вас и ваших друзей. В Python для вычисления суммы значений, хранящихся в списке, применяется функция `sum()`.

Список, в котором указано количество шариков у каждого из друзей

```
marbles = [10, 13, 39, 14, 41, 9, 3]

print('Сумма равна', sum(marbles))
```

Используем встроенную функцию `sum()` для подсчета числа шариков

```
Оболочка Python
Сумма равна 129
>>>
```

Но давайте ради интереса самостоятельно напишем аналогичную функцию, которая будет вычислять сумму чисел привычным для нас способом, т.е. в цикле.

Определим функцию для подсчета суммы чисел

```
def compute_sum(list):
    sum = 0

    for number in list:
        sum = sum + number

    return sum

print('Сумма равна', compute_sum(marbles))
```

Создаем локальную переменную `sum` для хранения накопительного итога и устанавливаем ее равной 0

Проходим по списку и добавляем каждое число к сумме

Наконец, возвращаем сумму

Выполним проверку, применив функцию `compute_sum()` к списку `marbles`

Получен тот же результат

```
Оболочка Python
Сумма равна 129
>>>
```

СИЛА МЫСЛИ

Представим, что разработчики Python решили убрать из языка все конструкции, связанные с циклами (в частности, циклы `for` и `while`). Как тогда подсчитать сумму чисел, хранящихся в списке?

Встроенную функцию `sum()` тоже нельзя использовать!

Мы пойдем другим путем

Оказывается, есть другой подход к решению задач, хорошо известный компьютерным инженерам и опытным программистам. Поначалу это покажется каким-то магическим трюком, но никакого фокуса тут нет! Вернемся к задаче подсчета шариков и рассмотрим два сценария: *простой случай* и *рекурсия*.

Простой случай

Это простейший из всех возможных сценариев. Что может быть проще, чем подсчет элементов пустого списка? Чему будет равна сумма в этом случае? Конечно же нулю!

Пустой список

```
compute_sum([])
```

Это простейший случай: если имеется пустой список, то мы заранее знаем, что сумма равна 0

Рекурсия

Теперь рассмотрим рекурсивный сценарий. В данном случае мы решаем упрощенную версию той же самой задачи. Вот как это работает: мы берем первый элемент списка и прибавляем к нему сумму остальных элементов.

```
[10, 13, 39, 14, 41, 9, 3]

10 + compute_sum([13, 39, 14, 41, 9, 3])
```

Как упростить задачу? Попробуем вычислить сумму списка, в котором на один элемент меньше

Мы немного упростили задачу: чтобы вычислить сумму списка, мы сложим 10 с суммой меньшего списка

Код для обоих случаев

Итак, у нас есть два сценария. Давайте напишем код для каждого из них. Как уже было сказано, поначалу все будет выглядеть немного непривычно, поэтому мы будем создавать новую функцию пошагово.

Простой случай

В простейшем случае задача проста: проверяем, является ли список пустым; если да, возвращаем значение 0.

```
def recursive_compute_sum(list):
    if len(list) == 0:
        return 0
```

Здесь мы проверяем, является ли список пустым (другими словами, равна ли его длина нулю); если да, возвращаем 0

Рекурсия

А вот код рекурсивного сценария менее очевиден. Будем двигаться шаг за шагом. Мы знаем, что нужно извлечь первый элемент списка и сложить его с суммой всех последующих элементов. Для наглядности создадим две переменные: в одной будет храниться первый элемент, в другой — оставшаяся часть списка (без первого элемента).

```
def recursive_compute_sum(list):
    if len(list) == 0:
        return 0
    else:
        first = list[0]
        rest = list[1:]
```

Простейший случай

Запишем в одну переменную первый элемент списка, а в другую переменную — остальной список

Это выражение возвращает список, начинающийся с элемента с индексом 1 и заканчивающийся последним элементом

А что если список содержит лишь один элемент? Будет получен пустой список

Теперь необходимо сложить первый элемент с суммой остальной части списка.

```
def recursive_compute_sum(list):
    if len(list) == 0:
        return 0
    else:
        first = list[0]
        rest = list[1:]
        sum = first + сумма остальной части списка
```

Необходимо просуммировать остаток списка, но разве это не то же самое, что мы сейчас кодируем? Суммировать список при суммировании списка? Ну и задачка!

Суммируется первый элемент и сумма остальной части списка

Но как это закодировать?

Нужно понять, как просуммировать оставшуюся часть списка. Догадываетесь? Нет ли в нашем распоряжении функции, которая вычисляет сумму списка? Как насчет функции `recursive_compute_sum()`?

```
def recursive_compute_sum(list):
    if len(list) == 0:
        return 0
    else:
        first = list[0]
        rest = list[1:]
        sum = first + recursive_compute_sum(rest)
        return sum
```

Не забудьте вернуть сумму после ее вычисления!

Раз уж функция `recursive_compute_sum()` вычисляет сумму списка, вызовем ее для вычисления суммы списка меньшего размера

Сохраните код функции `recursive_compute_sum()` вместе с кодом тестирования в файле *sum.py* и запустите программу, выполнив команду **Run > Run Module**. Убедитесь в том, что программа правильно вычисляет сумму списка. Настоящая магия!

```
marbles = [10, 13, 39, 14, 41, 9, 3]

def recursive_compute_sum(list):
    if len(list) == 0:
        return 0
    else:
        first = list[0]
        rest = list[1:]
        sum = first + recursive_compute_sum(rest)
        return sum

sum = recursive_compute_sum(marbles)
print('Сумма равна', sum)
```

Получили тот же результат, что и в цикле

```
Оболочка Python
Сумма равна 129
>>>
```

А разве мы только что не нарушили правило, запрещающее вызывать функцию до ее объявления? Ведь функция **recursive_compute_sum()** вызывается из своего собственного определения!

Нет, все справедливо. Помните: инструкции тела функции не выполняются до тех пор, пока она не будет вызвана в основном коде. В нашем случае функция `recursive_compute_sum()` сначала определяется, а вызывается в следующей строке:

```
sum = recursive_compute_sum(marbles)
```

Только в этот момент она вызывает саму себя. Но теперь она уже определена в программе, поэтому указанное правило не нарушается.

Если все это кажется вам немного запутанным, то ничего удивительного. Понимание придет с практикой. Пишите побольше рекурсивных функций и анализируйте их выполнение, чтобы понять, почему они работают.

Другими словами, нам пора перейти к выполнению практических заданий.

Вскоре мы займемся анализом рекурсивного кода

Давайте попрактикуемся

Не нужно бояться рекурсии: ее непросто понять, но, освоив ее принципы, вы поймете, что игра стоила свеч. Мы могли бы еще долго анализировать принципы работы функции `recursive_compute_sum()`, однако самый эффективный путь — побольше практиковаться, пытаясь решать различные задачи рекурсивным способом.

Давайте попрактикуемся на другой задаче. Помните программу составления палиндромов, рассмотренную в главе 4? Палиндромы — это слова, которые читаются одинаково в обоих направлениях.

Читается одинаково в прямом...

→

tacocat

←

...и обратном направлениях

Можно привести множество примеров палиндромов, например "madam", "radar" и "kayak". Это даже могут быть целые фразы (если не считать знаки препинания и пробелы), к примеру, "a nut for a jar of tuna", "a man, a plan, a canal: panama" или даже "a man, a plan, a cat, a ham, a yak, a yam, a hat, a canal: panama". Не верите? Прочитайте задом наперед.

Возьмите карандаш

Отвлекитесь от рекурсии и представьте, что вам нужно написать функцию, которая проверяет, является ли слово палиндромом. Для этого можно использовать любые инструменты Python, изученные в главах 1–7. Чтобы описать ход ваших мыслей, составьте псевдокод решения.

Нахождение палиндромов рекурсивным способом

Можем ли мы написать рекурсивную функцию, распознающую палиндромы? Все ли мы знаем для этого? Давайте попробуем. Помните, что нужно делать? Для получения рекурсивной функции мы сначала рассматриваем простейший случай, а затем начинаем упрощать задачу, рекурсивно вызывая ту же самую функцию. Проанализируем оба сценария.

Простой случай

Простейший случай самый очевидный. В действительности их тут даже два. Во-первых, как насчет пустой строки? Является ли она палиндромом? Поскольку она читается одинаково в обоих направлениях, то да, это палиндром.

Пустая строка

```
is_palindrome('')
```

Это простейший случай: если получена пустая строка, то у нас палиндром

Но есть еще один очень простой случай, который стоит рассмотреть: одиночная буква. Это палиндром? Она читается одинаково в обоих направлениях, так что да.

```
is_palindrome('a')
```

Одиночная буква — тоже палиндром, ведь она читается одинаково в обоих направлениях

Рекурсия

С рекурсивным случаем все немного сложнее. Как вы помните, нам нужно каким-то образом упростить задачу перед рекурсивным вызовом функции `is_palindrome()`. Что, если сначала сравнить два крайних символа строки, и, если они равны, вызвать функцию для оставшейся (более короткой) средней части строки?

Проверяем, совпадают ли крайние символы

```
'tacocat'
 'acoca'
```

А далее вызываем функцию `is_palindrome()`, чтобы проверить, является ли средняя часть строки палиндромом

Код рекурсивной функции распознавания палиндромов

Мы успешно описали простейший и рекурсивный случаи и можем переходить к написанию кода функции. Как и прежде, код простейшего случая тривиален, а вот над рекурсивным кодом придется подумать. Подобно тому, как это было при вычислении суммы списка, нам нужно разобраться, как упростить задачу, рекурсивно вызывая функцию из самой себя.

Простой случай

Тут все просто: проверяем, получена ли пустая строка или строка, содержащая одиночный символ.

```
def is_palindrome(word):
    if len(word) <= 1:
        return True
```

В простейшем случае проверяем, содержит ли слово пустую строку (длина равна 0) или одиночный символ (длина равна 1). Если да, то возвращаем `True`

Рекурсия

Теперь перейдем к рекурсивному случаю. Сначала нужно упростить задачу, сравнив два крайних символа строки. Если они равны, то для получения палиндрома все оставшиеся символы должны быть попарно равны. Если любая из пар не совпадает, возвращаем `False`.

```
def is_palindrome(word):
    if len(word) <= 1:
        return True
    else:
        if word[0] == word[-1]:


        else:
            return False
```

Снова простейший случай

Проверяем, равен ли первый символ последнему. Если нет, возвращаем `False`

Рекурсивный вызов будет выполнен на следующем этапе

Далее необходимо осуществить рекурсивный вызов. Функция проверила равенство двух внешних символов, а значит, мы имеем палиндром, *если оставшаяся часть строки является палиндромом.*

```
def is_palindrome(word):
    if len(word) <= 1:
        return True
    else:
        if word[0] == word[-1]:
            return is_palindrome(word[1:-1])
        else:
            return False
```

Если крайние символы совпадают, то необходимо проверить, является ли средняя часть слова палиндромом. Такая функция у нас уже есть, поэтому вызовем ее

Необходимо вернуть результат вызова функции `is_palindrome()`, т.е. `True` или `False`

Сохраните код функции `is_palindrome()` вместе с показанным ниже кодом тестирования в файле *palindrome.py* и выполните команду **Run > Run Module**. Проверьте правильность работы функции, проследив за выводимыми в окне консоли результатами. Попробуйте также задать собственные слова-кандидаты.

```
def is_palindrome(word):
    if len(word) <= 1:
        return True
    else:
        if word[0] == word[-1]:
            return is_palindrome(word[1:-1])
        else:
            return False

words = ['tacocat', 'radar', 'yak', 'rader', 'kayjak']
for word in words:
    print(word, is_palindrome(word))
```

Взгляните еще раз на полученный код. Проще ли он, чем версия с циклом?

Похоже, все работает!

Оболочка Python

```
tacocat True
radar True
yak False
rader False
kayjak False
>>>
```

Не бойтесь задавать вопросы

В: Как узнать, завершится ли когда-нибудь рекурсивная функция?

О: Другими словами, если функция все время вызывает саму себя, то как она остановится? Для этого и нужно условие простейшего случая. Если оно выполняется, значит, задачу можно решить напрямую, не прибегая к рекурсивному вызову. Это конечная точка, по достижении которой рекурсия прекращается.

В: Хорошо, но как узнать, будет ли когда-нибудь достигнут простейший случай?

О: Помните: при каждом следующем рекурсивном вызове наша задача немного упрощается. Если программа написана правильно, то рано и поздно решение сводится к простейшему случаю.

В: Я понимаю, как вызвать функцию из самой себя (так же, как и любую другую функцию), но как при этом не запутаться в ее аргументах? При каждом следующем рекурсивном вызове функция получает новый набор аргументов?

О: Хороший вопрос. Вы правы: при каждом вызове функция заново получает аргументы, а поскольку мы рекурсивно вызываем *ту же самую функцию*, ее параметры все время переназначаются согласно новым аргументам. Но не нужно думать, что при таком переназначении работа функции будет нарушена. Этого не произойдет. А все потому, что Python, как и большинство современных языков программирования, отслеживает все вызовы функции, запоминая соответствующие значения параметров (локальных переменных). Сейчас мы рассмотрим этот вопрос более подробно.

Как интерпретатор Python управляет рекурсией, отслеживая вложенные вызовы одной и той же функции? Чтобы разобраться в этом, давайте проанализируем работу функции `is_palindrome()`.

```
def is_palindrome(word):  ❶
    if len(word) <= 1:  ❷
        return True
    else:
        first = word[0]  ❸
        last = word[-1]
        middle = word[1:-1]
        if first == last:  ❹
            return is_palindrome(middle)  ❺
        else:
            return False
```

Вот наш код. Чтобы сделать его понятнее, мы добавили в него несколько локальных переменных. Они помогут нам проследить за тем, что происходит при выполнении кода

Рассмотрим, как это работает

`is_palindrome('radar')`

❶ Столкнувшись с вызовом функции, Python (как и любой другой язык программирования) первым делом создает структуру данных для хранения ее параметров и локальных переменных. Это называется *кадром*. В нашем случае в кадр заносится значение параметра `word`.

Кадр 1
`word = 'radar'`

❷ Далее проверяем длину строки `word`. В данном случае она больше 1.

❸ На следующем этапе создаются три локальные переменные, в которые записываются начальный и конечный символы строки, а также средняя часть строки. Все они тоже включаются в кадр.

Кадр 1
`word = 'radar'`
`first = 'r'`
`last = 'r'`
`middle = 'ada'`

❹ Теперь проверяем, равны ли первый и последний символы. Поскольку они равны, рекурсивно вызываем функцию `is_palindrome()`.

`return is_palindrome(middle)`

Согласно кадру 1 переменная `middle` *содержит строку* `'ada'`

---

❶ Мы перешли к следующему вызову функции, а значит, для хранения ее параметров и локальных переменных нужен новый кадр. Python хранит кадры подобно этажерке, один над другим. Такой набор кадров называется *стеком вызовов*.

❷ Длина параметра `word` по-прежнему больше 1, поэтому выполняется блок инструкций `else`.

Кадр 2
`word = 'ada'`

Кадр 1
`word = 'radar'`
`first = 'r'`
`last = 'r'`
`middle = 'ada'`

❸ Снова вычисляем значения локальных переменных и добавляем их в кадр.

❹ Как видите, первая и последняя буквы опять равны.

Следовательно, нам придется рекурсивно вызвать функцию `is_palindrome()` еще раз.

```
return is_palindrome(middle)
```

Согласно кадру 2 переменная **middle** *содержит строку* `'d'`

Кадр 2
```
word = 'ada'
first = 'a'
last = 'a'
middle = 'd'
```

Кадр 1
```
word = 'radar'
first = 'r'
last = 'r'
middle = 'ada'
```

---

❶ Мы перешли к очередному вызову функции, поэтому создается новый кадр со значениями ее параметров и локальных переменных. На этот раз параметр `word` содержит строку `'d'`.

❷ Длина параметра `word` наконец-то не превышает 1, поэтому функция сразу же возвращает `True`. Когда функция завершается, ее кадр удаляется, или *выталкивается*, из стека.

Кадр 3
```
word = 'd'
```

Кадр 2
```
word = 'ada'
first = 'a'
last = 'a'
middle = 'd'
```

Кадр 1
```
word = 'radar'
first = 'r'
last = 'r'
middle = 'ada'
```

Когда функция завершается, ее кадр выталкивается из стека

---

❸ Теперь нужно вернуть результат второго вызова функции `is_palindrome()`, который равен `True`. Ее кадр тоже удаляется из стека.

Кадр 2
```
word = 'ada'
first = 'a'
last = 'a'
middle = 'd'
```

Кадр 1
```
word = 'radar'
first = 'r'
last = 'r'
middle = 'ada'
```

Когда функция завершается, ее кадр выталкивается из стека

---

❹ И снова нужно вернуть результат вызова функции `is_palindrome()`, который равен `True`. Это последний кадр, который оставался в стеке, а значит, цепочка вызовов завершилась (получением результата `True`).

Кадр 1
```
word = 'radar'
first = 'r'
last = 'r'
middle = 'ada'
```

И снова выталкиваем кадр из стека

Когда первоначальный вызов функции `is_palindrome()` завершается, возвращается значение `True`.

Стек вызовов функции `is_palindrome()` *пуст*

Возвращает `True`

```
is_palindrome('radar')
```

Возьмите карандаш

Попробуйте проанализировать рекурсивный код самостоятельно. Как насчет функции `recursive_compute_sum()`?

```
def recursive_compute_sum(list):
    if len(list) == 0:
        return 0
    else:
        first = list[0]
        rest = list[1:]
        sum = first + recursive_compute_sum(rest)
        return sum

recursive_compute_sum([1, 2, 3])
```

Уже знакомый вам код

Вызов рекурсивной функции

`recursive_compute_sum([1, 2, 3])`

Кадр 1
list = [1, 2, 3]
first = 1
rest = [2, 3]

Первый шаг сделан за вас. Параметр `list` содержит список `[1, 2, 3]`, также вычислены локальные переменные `first` и `rest`

`recursive_compute_sum([2, 3])`

Кадр 2
list =
first =
rest =

Кадр 1
list = [1, 2, 3]
first = 1
rest = [2, 3]

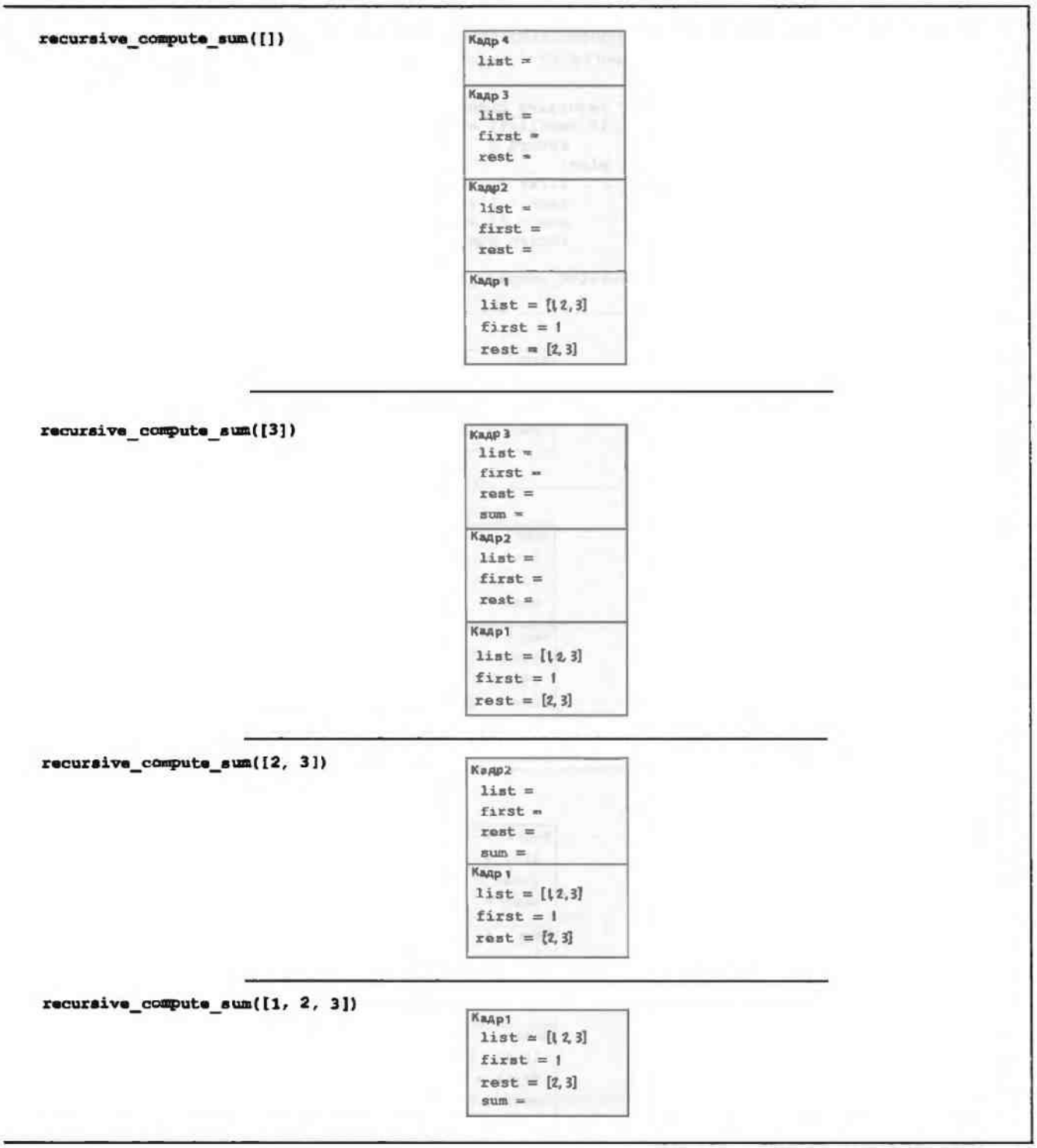
Выполните остальные вычисления и заполните кадры стека

`recursive_compute_sum([3])`

Кадр 3
list =
first =
rest =

Кадр 2
list =
first =
rest =

Кадр 1
list = [1, 2, 3]
first = 1
rest = [2, 3]


recursive_compute_sum([])
Кадр 4
list =
Кадр 3
list =
first =
rest =
Кадр2
list =
first =
rest =
Кадр 1
list = [1, 2, 3]
first = 1
rest = [2, 3]
recursive_compute_sum([3])
Кадр 3
list =
first =
rest =
sum =
Кадр2
list =
first =
rest =
Кадр1
list = [1, 2, 3]
first = 1
rest = [2, 3]
recursive_compute_sum([2, 3])
Кадр2
list =
first =
rest =
sum =
Кадр 1
list = [1, 2, 3]
first = 1
rest = [2, 3]
recursive_compute_sum([1, 2, 3])
Кадр1
list = [1, 2, 3]
first = 1
rest = [2, 3]
sum =

Посиделки

Тема дискуссии: **цикл или рекурсия — что лучше?**

Цикл

Ну конечно я, разве непонятно? Программисты во много раз чаще используют циклы, чем рекурсию.

Серьезно? Все современные языки программирования поддерживают рекурсию, но программисты почему-то выбирают циклы.

Вообще-то, это книга о Python. Или я что-то перепутал?

Эффективное? Ты шутишь?! Слышала что-нибудь о стеке вызовов?

Всякий раз, когда функция вызывает саму себя, интерпретатор Python вынужден создавать небольшую структуру данных, предназначенную для хранения всех параметров и локальных переменных текущей функции. При рекурсивных вызовах интерпретатору приходится поддерживать целый стек таких структур, который становится все больше и больше с каждым следующим вызовом функции. На все это расходуется оперативная память, и если вызовов оказывается слишком много, то в какой-то момент — бац! — программа "слетает".

Рекурсия

Все зависит от языка программирования.

А как насчет таких языков, как LISP, Scheme или Clojure? В них рекурсия применяется намного чаще, чем циклы.

Не в этом дело. Просто есть программисты, которые хорошо понимают принципы рекурсии и знают, насколько это элегантное и эффективное решение.

Конечно слышала, и читатели тоже, но уж просвети нас, пожалуйста.

Цикл

Все верно, но при рекурсивном вызове это может происходить неконтролируемое число раз, и рано или поздно возникнут проблемы.

И почему-то это не останавливает миллионы программистов от циклической обработки палиндромов.

Разве что для уникумов, которые смогли понять рекурсию.

Согласись, для большинства задач итеративный подход лучше.

Слишком много возни ради большей понятности кода.

Ты имеешь в виду такие "супермегасложные" задачи, как распознавание палиндромов?

Я ничего не говорил о том, что говорить о книге в самой книге... Ой! Да ну тебя!

Рекурсия

Но ведь так работают все языки программирования, даже старые, не только Python. Описанный процесс происходит при любом вызове функции.

А вот и нет. Во многих рекурсивных алгоритмах никаких проблем не возникает, плюс есть немало приемов, позволяющих держать ситуацию под контролем. Но самое главное — это понятность рекурсивного кода. Пример с палиндромами очень наглядный. Решение с циклом было просто уродливым.

Я лишь подчеркиваю, что в определенных алгоритмах рекурсивный подход проще для понимания и кодирования.

Да ладно, немного практики, и все становится понятно.

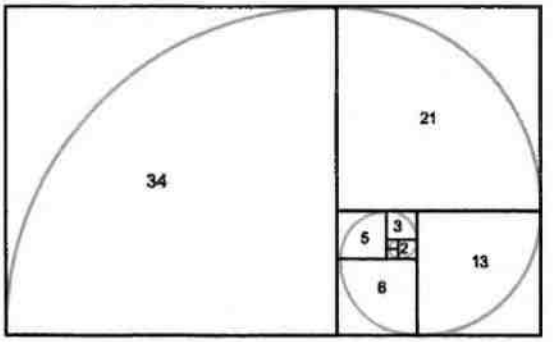
Не лучше, а естественнее. Впрочем, есть и задачи, для которых рекурсивный подход естественнее.

Дело не в том, что код становится более читабельным. Просто есть задачи, которые естественным образом решаются с помощью рекурсии, тогда как итеративный подход оказывается чересчур трудоемким.

Конечно же нет! Но, думаю, к концу книги мы столкнемся с такими задачами.

Кстати, тебе не кажется, что говорить о книге в самой книге — это и есть рекурсия? Я проникаю повсюду.

ЛАБОРАТОРИЯ РЕКУРСИИ



Сейчас мы протестируем код рекурсивной функции, вычисляющей *числа Фибоначчи*. Эти числа образуют последовательность, в которой каждое число равно сумме двух предыдущих чисел.

Вычисляется она следующим образом.

fibonacci(0) = 0
fibonacci(1) = 1
⋮
fibonacci(n) = fibonacci(n−1) + fibonacci(n−2)

Если вызвать функцию с аргументом 0, то она вернет 0, а если с аргументом 1, то она вернет 1

Для любого другого числа n мы получаем числа Фибоначчи путем сложения fibonacci(n-1) и fibonacci(n-2)

Числа Фибоначчи имеют отношение к золотому сечению, которое часто встречается в природе и считается признаком хорошего дизайна

Ниже приведено несколько первых чисел Фибоначчи.

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(2) = 1
fibonacci(3) = 2
fibonacci(4) = 3
fibonacci(5) = 5
fibonacci(6) = 8

Любое число последовательности вычисляется путем сложения двух предыдущих чисел Фибоначчи

Она продолжается числами 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181 и т.д.

В лаборатории был разработан алгоритм вычисления чисел Фибоначчи. Вот его код.

Используем приведенное выше определение

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Простой случай

Рекурсия

Если n равно 0 или 1, просто возвращаем это число

В противном случае возвращаем сумму двух предыдущих чисел Фибоначчи путем рекурсивного вызова функции `fibonacci()`

Обратите внимание: функция `fibonacci()` *рекурсивно вызывается дважды!*

Алгоритм необходимо протестировать. В лаборатории рекурсии все должно работать правильно и быстро! Для этого мы написали небольшой тестовый код, в котором используется новый модуль, `time`, позволяющий фиксировать время выполнения программы.

```
import time
```

Мы используем модуль time для хронометрирования кода

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Это рекурсивная функция

Тестовый код

```
for i in range(20, 55, 5):
    start = time.time()
    result = fibonacci(i)
    end = time.time()
    duration = end - start
    print(i, result, duration)
```

В качестве теста вычислим числа Фибоначчи с номерами от 20 до 50 с шагом 5. Если все пройдет успешно, мы вычислим 100 чисел

Время начала

Вычисляем число Фибоначчи

Время конца

Определяем длительность

Выводим результаты

Мы будем замерять длительность вычисления каждого числа Фибоначчи. Для этого нам понадобится модуль time. Подробнее о модулях для работы со значениями даты и времени будет рассказано в приложении А

Ваша задача — ввести код и выполнить тест. Полученные данные занесите в приведенную ниже таблицу, указав номер n, число Фибоначчи и длительность его вычисления в секундах. Чтобы программу можно было использовать в производственных целях, она должна вычислить первые 100 чисел Фибоначчи менее чем за 5 секунд. Соответствует ли программа этому критерию по результатам теста?

Если вычисления длятся слишком долго, то программу можно остановить, закрыв окно оболочки Python

Вот что мы получили для первого теста, где n=20. Результаты будут зависеть от скорости компьютера

Тестирование чисел Фибоначчи

Номер	Ответ	Длительность
20	6765	0,002 секунды

Пока что все проходит достаточно быстро

Остальные результаты на следующей странице; сравните их со своими

ПРОВАЛ ОПЫТА С РЕКУРСИЕЙ

Итак, чтобы соответствовать производственным стандартам, наша программа должна успеть вычислить 100 чисел Фибоначчи за 5 секунд. И как прошел тест? Что? Успели пообедать, а программа все еще работает? Не переживайте, мы все заранее посчитали. Результаты приведены ниже, и они, мягко говоря, не впечатляют.

Код работает правильно в том смысле, что мы получаем правильные результаты

Тестирование чисел Фибоначчи

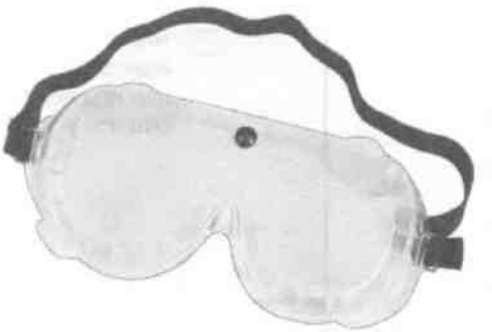
Номер	Ответ	Длительность
20	6765	0,002 секунды
25	75025	0,04 секунды
30	832040	0,4 секунды
35	9227465	4,8 секунды
40	102334155	56,7 секунды
45	1134903170	10,5 минуты
50	12586269025	1,85 часа

Ваши показатели могут быть другими, в зависимости от скорости компьютера

Вычисления поначалу проходят очень быстро, но затем все сильнее замедляются по мере увеличения числа n. При n=50 у нас ушло почти 111 минут на вычисление числа Фибоначчи!

Да уж, хуже некуда... Мы надеялись, что программа сумеет вычислить 100 первых чисел Фибоначчи за отведенные для этого 5 секунд, но тест показал, что уже на 50-м числе расчеты длятся больше часа.

Что пошло не так? Почему все так долго? Вы пока подумайте над причинами провала, а мы вернемся в лабораторию рекурсии после знакомства с одной интересной структурой данных. (Может быть, с ее помощью у нас все получится?)



Антисоциальная сеть

Вы уже знаете достаточно много, чтобы заняться действительно серьезными вещами. Вам вполне по силам приступить к своему первому коммерческому проекту. Как вам идея для стартапа: *новая социальная сеть*? Что? Всем хватает Facebook и его конкурентов? Не волнуйтесь, у нас кое-что другое: *антисоциальная сеть*! Здесь вы сможете делать публикации типа "оставь улыбку всяк сюда входящий" или "счастливым вход запрещен". У антисоциальной сети есть также уникальная особенность: возможность в любой момент определить самого антисоциального пользователя. В общем, идея на миллион долларов, осталось только написать код.

Начнем с простого: нужно составить список пользователей. Для каждого из них мы будем хранить имя и адрес электронной почты.

Сила мысли

Для хранения имен и почтовых адресов пользователей мы будем использовать уже известные нам структуры. Пусть в одном списке хранятся имена, а в другом — соответствующие им почтовые адреса. Это должны быть параллельные списки (см. главу 4). Иными словами, имени с индексом 42 в первом списке будет соответствовать почтовый адрес с таким же индексом в другом списке.

```
names = ['Ким', 'Джон', 'Джош']
emails = ['kim@oreilly.com', 'john@abc.com', 'josh@wickedlysmart.com']
```

Это напоминает то, как мы хранили цвета и позиции черепашек в предыдущей главе

Какие недостатки есть у такого подхода?

- ☐ Вставка новых имен и адресов требует поддержания согласованности между двумя списками.
- ☐ Чтобы найти кого-то, придется просмотреть весь список.
- ☐ Что, если потребуется добавить еще один атрибут, например пол или номер телефона? Это все усложнит.
- ☐ Никаких недостатков нет. Все работает.
- ☐ Трудно понять, как данные связаны между собой, потому что имена и адреса хранятся в разных списках.
- ☐ При удалении пользователя нужно синхронизировать списки.
- ☐ _____

Знаете еще что-то?

Ах, если бы существовала такая структура данных, которая позволяла бы присваивать элементам запоминающиеся имена без всякой индексации! И чтобы данные можно было добавлять, не заботясь о том, в какой позиции они будут храниться. И чтобы значения можно было быстро находить, не просматривая весь список. Как было бы здорово! Знаю, это лишь мои мечты...

Знакомство со словарем

Встречайте новый тип данных Python: *словарь*. Его еще называют ассоциативным массивом. Это очень удобная и гибкая структура данных, которую мы будем применять для решения самых разных задач, включая создание нашей антисоциальной сети (в частности, она поможет нам хранить данные о пользователях). Но сначала познакомимся с концепцией словаря и принципами его работы.

Первое, что отличает словари от других структур данных, таких как списки, — это их *неупорядоченность*. В списках все значения упорядочены согласно индексам. Например, для получения третьего значения нужно обратиться к элементу с индексом 3. В словарях доступ к значениям осуществляется не по индексам, а по связанным с ними ключам. Каждому значению словаря соответствует свой ключ.

Создание словаря

```
my_dictionary = {}
```

Используйте фигурные скобки для создания пустого словаря, в котором можно хранить ключи и значения

Помните: фигурные скобки у словарей, квадратные — у списков

Добавление элементов в словарь

Элементы словаря представлены парами "ключ — значение". Вот как в словаре сохраняется телефонный номер 867-5309, принадлежащий Дженни.

```
my_dictionary['Дженни'] = '867-5309'
```

Записываем значение, в данном случае номер телефона в строковом формате, в строковый ключ 'Дженни'

В таком формате можно сохранить сколько угодно контактов.

```
my_dictionary['Пол'] = '555-1201'
my_dictionary['Дейвид'] = '321-6617'
my_dictionary['Джейми'] = '771-0091'
```

Можно хранить произвольное число пар "ключ — значение"

```
my_dictionary['Пол'] = '443-0000'
```

При записи значения в существующий ключ прежнее значение затирается

Получение значений из словаря

Для доступа к элементу словаря требуется указать только его ключ.

```
phone_number = my_dictionary['Дженни']
print("Номер телефона Дженни:", phone_number)
```

Оболочка Python

```
Номер телефона Дженни: 867-5309
>>>
```

Типы данных значений и ключей

Ключи словаря могут быть числами, строками или булевыми значениями. В качестве значения можно использовать любые типы данных Python. Ниже приведено несколько примеров допустимых значений.

```
my_dictionary['Возраст'] = 27
my_dictionary[42] = 'ответ'
my_dictionary['Баллы'] = [92, 87, 99]
```

Можно хранить строковый ключ и целочисленное значение, целочисленный ключ и строковое значение, строковый ключ и список и т.п. Значением может быть любой тип Python

Допускаются и другие типы ключей, но мы их пока не рассматриваем

Удаление ключа

```
del my_dictionary['Дейвид']
```

Удаляем из словаря ключ 'Дейвид' вместе с его значением

Инструкция del может применяться и к другим структурам, например спискам

У словарей есть также метод pop(), который удаляет ключ, возвращая его значение

Проверка существования ключа

В Python применяется унифицированный синтаксис для проверки принадлежности значения к тому или иному типу коллекции, будь то список, строка или словарь. Вот как узнать, имеется ли нужный ключ в словаре.

Используйте оператор in для проверки того, имеется ли указанный ключ в словаре

```
if 'Дженни' in my_dictionary:
    print('Нашел ее:', my_dictionary['Дженни'])
else:
    print('Нужно достать ее телефон')
```

Вот код для удаления нужной записи из словаря

```
if 'Дейвид' in my_dictionary:
    del(my_dictionary['Дейвид'])
```

Не бойтесь задавать вопросы

В: Что будет, если попытаться удалить несуществующий ключ?

О: Будет сгенерировано исключение KeyError. Об обработке исключений мы поговорим позже, но лучше избегать их, проверяя существование ключа перед его использованием.

В: Я правильно понимаю, что в словаре не может быть повторяющихся ключей?

О: Да. Каждый ключ словаря уникален. Например, в словаре my_dictionary может быть только один ключ 'Ким'. Если записать в него другое значение, то оно заменит предыдущее.

В: Словари — очень удобные структуры данных, но мне кажется, что отсутствие индексации делает их менее эффективными. Насколько быстро обрабатываются данные в больших словарях?

О: Помните, в начале главы было сказано о том, что нам придется поменять способ мышления? Так вот, как ни странно, словари оказываются эффективнее списков во многих задачах. Вскоре вы узнаете, почему.

В: Если для словаря поддерживается встроенный оператор del, то как насчет оператора len?

О: Он тоже поддерживается. Оператор len вернет общее количество ключей в словаре.

Просмотр словаря

Помните о том, что словарь хранит неупорядоченные данные. Это означает, что, просматривая его элементы, нельзя надеяться на то, что они следуют в определенном порядке.

Как и в случае списков и строк, используем цикл `for/in` для перебора всех ключей словаря

```
for key in my_dictionary:
    print(key, ':', my_dictionary[key])
```

Выведем ключ...

...а затем — его значение

```
Оболочка Python
Дженни : 867-5309
Пол : 443-0000
Джейми : 771-0091
Возраст : 27
42 : ответ
Баллы : [92, 87, 99]
>>>
```

В вашем случае порядок может быть другим

Строковые словари

Как и в случае списков, при создании словарей допускается использовать строковый формат объявления элементов.

Каждая пара "ключ — значение" разделяется двоеточием и завершается запятой (кроме последней пары)

Так создается полноценный словарь

```
harry = {'Имя': 'Гарри',
         'Фамилия': 'Поттер',
         'Факультет': 'Гриффиндор',
         'Друзья': ['Рон', 'Гермиона'],
         'Год рождения': 1980}
```

Содержимое словаря можно вывести на экран.

```
print(my_dictionary)
```

```
Оболочка Python
{'Дженни': '867-5309', 'Пол': '443-
0000', 'Джейми': '771-0091', 'Возраст':
27, 42: 'ответ', 'Баллы': [92,
87, 99]}
>>>
```

Не полагайтесь на порядок следования элементов словаря

Даже если ключи словаря постоянно следуют в одном и том же порядке, вы не должны рассчитывать на то, что он будет сохраняться при выполнении программы в других системах или реализациях Python. Полагаясь на порядок хранения значений в словаре, вы рано или поздно столкнетесь с проблемами.

Возьмите карандаш

Получив базовые знания о словарях, давайте применим их на практике. Проанализируйте приведенный ниже код и постарайтесь предугадать результаты его работы.

```
movies = []        # Список
movie = {}         # Словарь

movie['Название'] = 'Forbidden Planet'
movie['Год'] = 1957
movie['Рейтинг'] = '*****'
movie['Год'] = 1956

movies.append(movie)

movie2 = {'Название': 'I Was a Teenage Werewolf',
          'Год': 1957, 'Рейтинг': '****'}
movie2['Рейтинг'] = '***'

movies.append(movie2)

movies.append({'Название': 'Viking Women and the Sea Serpent',
               'Год': 1957,
               'Рейтинг': '**'})

movies.append({'Название': 'Vertigo',
               'Год': 1958,
               'Рейтинг': '*****'})

print('Рекомендации фильмов')
print('--------------------')
for movie in movies:
    if len(movie['Рейтинг']) >= 4:
        print(movie['Название'], '(' + movie['Рейтинг'] + ')', movie['Год'])
```

Словари в антисоциальной сети

Вы уже знаете 95% того, что необходимо знать для работы со словарями в Python, осталось применить знания на практике. Вы умеете сохранять и извлекать элементы словарей, но ведь это далеко не все. Простота словарей обманчива. Давайте попробуем задействовать словари в проекте нашей антисоциальной сети.

Помните, мы хотели хранить коллекцию имен пользователей вместе с адресами электронной почты. Мы начали с двух списков, что оказалось не самым удобным решением, поскольку при добавлении и удалении имен требовалось синхронизировать оба списка. К тому же для поиска нужного имени приходилось просматривать весь список, а появление дополнительной характеристики, такой как пол, означало необходимость создания нового списка. Слишком сложно! Посмотрим, упростится ли задача при использовании словарей.

Здесь в двух списках хранятся имена и адреса нескольких пользователей

```
names = ['Ким', 'Джон', 'Джош']
emails = ['kim@oreilly.com', 'john@abc.com', 'josh@wickedlysmart.com']
```

Заменим списки словарем.

Так намного понятнее

```
users = {'Ким' : 'kim@oreilly.com',
         'Джон': 'john@abc.com',
         'Джош': 'josh@wickedlysmart.com'}
```

Что произойдет при добавлении или удалении пользователя?

```
users['Анна'] = 'anna@gmail.com'
del users['Джон']
```

Добавляем

Удаляем

Как видите, все просто: не нужно беспокоиться о синхронизации двух списков

Как узнать адрес одного из пользователей? Предположим, нам нужно получить адрес Джоша.

```
if 'Джош' in users:
    print("Электронная почта Джоша:", users['Джош'])
```

Сначала проверяем, существует ли ключ 'Джош', а затем получаем адрес электронной почты

Оболочка Python

```
Электронная почта Джоша: josh@wickedlysmart.com
>>>
```

Должен признать, с точки зрения программирования пользоваться словарем удобнее, чем двумя списками. Но эффективнее ли это, чем списки с явной индексацией?

Словарь намного эффективнее. Чтобы понять, почему это так, нужно знать, как словари обрабатываются интерпретатором. Возьмем, к примеру, список. Если попытаться найти в списке имя "Джош", вы должны будете просмотреть список последовательно, начиная с первого элемента, пока не будет найдено имя "Джош". В худшем случае список придется просмотреть целиком!

Словари используют структуру данных, называемую *хеш-таблицей*. Это разновидность массива, расположение элементов в котором определяется с помощью специальной *хеш-функции*, получающей ключ в качестве аргумента. Таким образом, индекс элемента вычисляется путем однократного вызова хеш-функции, а не методом грубой силы (полный просмотр списка). К счастью, вся работа с хеш-функциями выполняется автоматически, без нашего участия.

Конечно, хеш-функции не идеальны, и порой одному индексу соответствует сразу несколько значений (словарь знает, как поступать в подобных случаях), но такое случается нечасто. В итоге среднее время поиска значений в словаре по ключу является *константным* (т.е. не зависящим от размера словаря). Таким образом, поиск по ключу оказывается намного эффективнее поиска по индексу.

Это компьютерный термин

Добавление новых атрибутов

Применение словарей для управления именами пользователей и адресами электронной почты в нашей антисоциальной сети оказалось удачным решением, но не забывайте, что в будущем мы планируем добавлять и другие характеристики, например пол. В случае параллельных списков нам пришлось бы создать третий список, а как быть при хранении данных в словарях? Понадобится еще один словарь?

```
email = {'Ким' : 'kim@oreilly.com',
         'Джон': 'john@abc.com',
         'Джош': 'josh@wickedlysmart.com'}

genders = {'Ким' : 'ж',
           'Джон': 'м',
           'Джош': 'м'}
```

Можно сделать это так, но тогда опять придется управлять двумя словарями при поиске, добавлении или удалении пользователя

Такой подход работоспособен, но он потребует одновременного управления сразу двумя словарями, а это совсем не то, что нам нужно. Для более эффективного решения задачи нужно придумать способ обойтись всего одним словарем. Что, если использовать словарь для хранения сразу всех атрибутов каждого пользователя?

```
attributes = {
         'Эл. почта' : 'kim@oreilly.com',
         'Пол': 'ж',
         'Возраст': 27,
         'Друзья': ['Джон', 'Джош']
}
```

Это словарь, хранящий атрибуты Ким. Подобный словарь можно создать для каждого пользователя

Мы добавили в словарь атрибутов новый список, хранящий имена друзей

Вооружившись новыми знаниями, перепишем наш код так, чтобы все характеристики пользователей хранились в одном словаре.

```
users = {}
users['Ким'] = attributes
```

Записываем в переменную users пустой словарь, после чего добавляем словарь атрибутов по ключу 'Ким'

Не спешите переходить дальше и проанализируйте этот код. Мы делаем значением ключа 'Ким' другой словарь

Давайте также добавим Джона и Джоша.

```
users['Джон'] = {'Эл. почта' : 'john@abc.com', 'Пол': 'м', 'Возраст': 24, 'Друзья': ['Ким', 'Джош']}
users['Джош'] = {'Эл. почта' : 'josh@wickedlysmart.com', 'Пол': 'м', 'Возраст': 32, 'Друзья': ['Ким']}
```

То же самое: не спешите идти дальше и проанализируйте этот код

Мы назначаем словари ключам 'Джон' и 'Джош'. Значения словарей указываются в виде литералов

Возьмите карандаш

Словарь в словаре — это популярное решение. Рассмотрим, насколько оно эффективно в нашем кинематографическом примере. Постарайтесь предугадать результаты работы приведенного ниже кода.

```
movies = {}
movie = {}

movie['Название'] = 'Forbidden Planet'
movie['Год'] = 1957
movie['Рейтинг'] = '*****'
movie['Год'] = 1956

movies['Forbidden Planet'] = movie

movie2 = {'Название': 'I Was a Teenage Werewolf',
          'Год': 1957, 'Рейтинг': '****'}
movie2['Рейтинг'] = '***'
movies[movie2['Название']] = movie2

movies['Viking Woman and the Sea Serpent'] = {'Название': 'Viking Women and the Sea Serpent',
                                              'Год': 1957,
                                              'Рейтинг': '**'}
movies['Vertigo'] = {'Название': 'Vertigo',
                     'Год': 1958,
                     'Рейтинг': '*****'}

print('Рекомендации фильмов')
print('--------------------')
for name in movies:
    movie = movies[name]
    if len(movie['Рейтинг']) >= 4:
        print(movie['Название'], '(' + movie['Рейтинг'] + ')', movie['Год'])

print('Выбор редакции')
print('--------------------------')
movie = movies['I Was a Teenage Werewolf']
print(movie['Название'], '(' + movie['Рейтинг'] + ')', movie['Год'])
```

Тренируем мозг

Теперь, когда мы определились со способом хранения данных о пользователях антисоциальной сети, можно приступать к написанию кода. Давайте напишем функцию `average_age()`, которая будет запрашивать имя пользователя и возвращать средний возраст его друзей. Попробуйте сделать это сами; в конце концов, вам уже многое по силам. Только начните с составления псевдокода или блок-схемы алгоритма, чтобы все правильно спланировать.

```
users = {}
users['Ким'] = {'Эл. почта' : 'kim@oreilly.com', 'Пол': 'ж', 'Возраст': 27, 'Друзья': ['Джон', 'Джош']}
users['Джон'] = {'Эл. почта' : 'john@abc.com', 'Пол': 'м', 'Возраст': 24, 'Друзья': ['Ким', 'Джош']}
users['Джош'] = {'Эл. почта' : 'josh@wickedlysmart.com', 'Пол': 'м', 'Возраст': 32, 'Друзья': ['Ким']}
```

Запишите здесь свою функцию `average_age()`

```
average_age('Ким')
average_age('Джон')
average_age('Джош')
```

Вот что должно быть получено с помощью тестового кода

Оболочка Python

```
Ким, средний возраст друзей: 28.0
Джон, средний возраст друзей: 29.5
Джош, средний возраст друзей: 27.0
>>>
```

Ключевая особенность антисоциальной сети

Вы ведь не забыли о ней, правда? Это главный элемент демоверсии программы, который позволит нам привлечь инвесторов к проекту. В антисоциальной сети можно в любой момент определить самого антисоциального пользователя, у которого меньше всего друзей. Осталось написать соответствующий код.

Итак, у нас есть словарь с данными о пользователях, каждый элемент которого состоит из ключа, представленного именем пользователя, и значения — вложенного словаря, хранящего все атрибуты пользователя. Одним из атрибутов является список друзей пользователя. Нам нужно просмотреть атрибуты всех пользователей и определить, у кого из них меньше всего друзей. Приступим!

Кто из них?

Определение самого антисоциального пользователя

Код поиска пользователя с наименьшим числом друзей будет не таким уж сложным, но все равно давайте сначала составим псевдокод, чтобы получить общее представление об алгоритме решения задачи.

1. Задать переменную `max` равной большому числу.
   Создаем переменную max для хранения текущего счетчика наименьшего числа друзей. Для начала инициализируем переменную каким-нибудь большим числом
2. Для каждого имени в словаре `users`:
   Проходим по каждому ключу в словаре users
   - А. получить словарь атрибутов пользователя;
     Для каждого пользователя получаем словарь атрибутов, связанный с его именем
   - Б. получить список друзей из словаря атрибутов;
     Используем ключ 'friends' для получения списка друзей
   - В. если число друзей меньше, чем `max`:
     Если число друзей меньше, чем текущее значение max, получаем нового кандидата
     - i. записать в переменную `most_anti_social` имя;
       Задаем переменную most_anti_social равной имени пользователя
     - ii. задать переменную `max` равной числу друзей.
       Задаем переменную max равной новому числу друзей
3. Вывести имя пользователя с ключом `most_anti_social`.
   Пройдя по всем именам в словаре users, находим самого антисоциального пользователя и выводим его имя

Возьмите карандаш

Имея перед глазами псевдокод, используйте свои знания о словарях, чтобы заполнить пропуски в приведенном ниже коде.

```
max = 1000
for name in __________:
    user = ______[________]
    friends = user[_________]
    if len(_________) < max:
        most_anti_social = _______
        max = len(________)

print('Меньше всего друзей имеет', ________________)
```

Что-то мы давно ничего не тестировали! Введите приведенный ниже код, сохраните его в файле *antisocial.py* и проверьте результат работы программы.

```
users = {}
users['Ким'] = {'Эл. почта' : 'kim@oreilly.com', 'Пол': 'ж', 'Возраст': 27, 'Друзья': ['Джон', 'Джош']}
users['Джон'] = {'Эл. почта' : 'john@abc.com', 'Пол': 'м', 'Возраст': 24, 'Друзья': ['Ким', 'Джош']}
users['Джош'] = {'Эл. почта' : 'josh@wickedlysmart.com', 'Пол': 'м', 'Возраст': 32, 'Друзья': ['Ким']}

max = 1000
for name in users:
   user = users[name]
   friends = user['Друзья']
   if len(friends) < max:
       most_anti_social = name
       max = len(friends)

print('Меньше всего друзей имеет', most_anti_social)
```

Оболочка Python

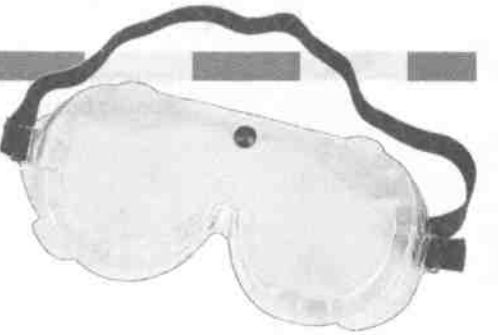
```
Меньше всего друзей имеет Джош
>>>
```

Это Джош!

Слово за Вами!

Это все, о чем мы собирались рассказать вам в рамках проекта антисоциальной сети. Дальнейшую работу вам предстоит выполнить самостоятельно. Идею вы поняли; берите имеющийся код – и вперед к богатству и славе! Пришлете нам открытку, когда разбогатеете (хотя мы не обидимся, если вы этого не сделаете, ведь у основателя антисоциальной сети наверняка будет не слишком много друзей).

Впрочем, тема словарей еще не закончена, так как нам предстоит завершить другой важный проект.

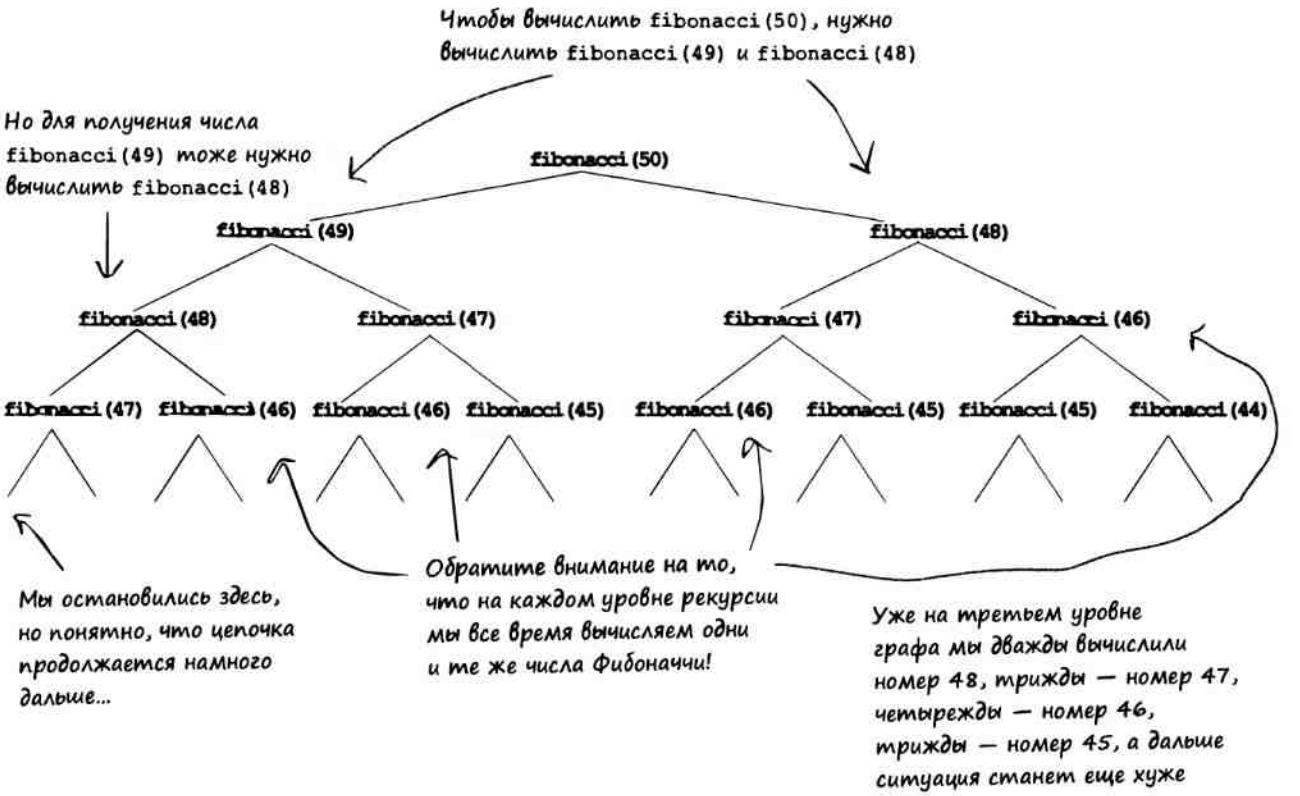


А тем временем...

ЛАБОРАТОРИЯ РЕКУРСИИ

В прошлый раз, когда мы покинули лабораторию рекурсии, ее сотрудники пребывали в расстроенных чувствах. Попытавшись определить первые 100 чисел Фибоначчи за 5 секунд, мы потерпели фиаско. На вычисление всего лишь 50 чисел ушло больше часа. Неужели все так безнадежно? Вовсе нет!

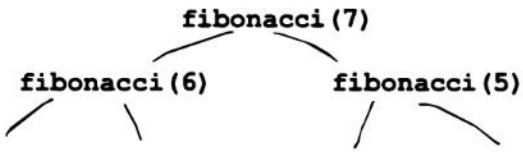
Но мы не сможем продвинуться в решении задачи, пока не поймем, почему первый вариант программы оказался настолько медленным. Рассмотрим цепочку рекурсивных вызовов при вычислении, к примеру, числа `fibonacci(50)`.



Итак, хоть мы имеем логически правильный и понятный код, он оказывается крайне неэффективным. Алгоритм таков, что для получения любого числа программе приходится вычислять все предшествующие ему числа Фибоначчи. Это приводит к огромному объему ненужных расчетов, так как программа постоянно вычисляет одни и те же числа Фибоначчи снова и снова. Например, всякий раз, когда нужно получить число `fibonacci(5)`, нам приходится повторно вычислять значения `fibonacci(4)`, `fibonacci(3)` и `fibonacci(2)`.

Возьмите карандаш

Нарисуйте граф вызовов при получении числа `fibonacci(7)`. В конце подсчитайте, сколько повторных вызовов приходится на каждое из значений от 0 до 7.



Впишите сюда итоговые значения. В таблице уже дан ответ для чисел `fibonacci(7)` *и* `fibonacci(6)`*, которые вычисляются по одному разу*

fibonacci(7): 1 раз	**fibonacci(6):** 1 раз	**fibonacci(5):** ____
fibonacci(4): ____	**fibonacci(3):** ____	**fibonacci(2):** ____
fibonacci(1): ____	**fibonacci(0):** ____	

Предупреждение: ответ может шокировать

И угадайте, сколько раз в функции `fibonacci(50)` придется вычислить `fibonacci(3)`? _____

Сила мысли

Итак, мы многократно вычисляем одни и те же числа Фибоначчи. Подумайте, как можно повысить эффективность программы, не внося радикальных изменений в код. Запишите здесь свои соображения на этот счет.

Ах, если бы существовал способ сохранять результаты предыдущего вызова функции, чтобы при последующем вызове с тем же самым аргументом можно было просто воспользоваться предыдущим результатом, а не вычислять его заново! Знаю, это лишь мои мечты...

Можно ли запомнить результаты работы функции?

Это разумная мысль. Например, если функция `fibonacci()` вызывается с аргументом `49`, то мы сохраняем ее результат, чтобы в следующий раз, когда функция будет вызвана с тем же аргументом, она могла поискать готовый результат, а не вычислять его заново.

С учетом того, как часто вызывается функция `fibonacci()` с одними и теми же аргументами в стеке рекурсивных вызовов, если бы мы могли сохранять ее результаты, а не вычислять их каждый раз заново, это позволило бы сэкономить уйму времени. Но насколько большой была бы такая экономия? Час и больше? Нужно проверить.

Что же нам понадобится для сохранения результатов работы функции `fibonacci()`? Нужен способ сохранить значение аргумента *n* и результат вызова `fibonacci(n)`. Кроме того, необходимо получать быстрый доступ к результатам для любого заданного значения *n*.

Есть идеи, как это сделать? Описанная структура ничего не напоминает?

Хранение чисел Фибоначчи в словаре

Не кажется ли вам, что нам нужен словарь? Рассмотрим, как это можно сделать.

Кешем часто называют место хранения данных, к которым нужно получать быстрый доступ

1. **Создать словарь и назвать его `cache`.**
2. **Всякий раз, когда функция `fibonacci()` вызывается с аргументом *n*:**
   - **А** Проверить, есть ли в словаре `cache` ключ *n*.
     - **i** Если есть, вернуть значение ключа *n*.
   - **Б** В противном случае вычислить число Фибоначчи для *n*.
   - **В** Сохранить результат в словаре `cache` по ключу *n*.
   - **Г** Вернуть число Фибоначчи.

Таким образом, всякий раз, когда функция `fibonacci()` вызывается с аргументом *n*, мы сначала проверяем, есть ли в словаре `cache` ключ *n*. При наличии ключа мы просто возвращаем соответствующее число Фибоначчи как результат вызова функции, вместо того чтобы вычислять его заново.

Если же ключа нет, то число Фибоначчи вычисляется стандартным способом, но перед возвращением результата оно сохраняется в словаре по ключу *n*.

Мемоизация

Если вы думаете, будто мы только что изобрели гениальный способ, позволяющий сохранять результаты работы функции, то ошибаетесь. Это широко известная и достаточно эффективная методика, которая называется *мемоизация*. Ее применяют для оптимизации программ, содержащих затратные вызовы функций. Насколько затратные? У вас все равно нет столько денег. Шутка!

В программировании под затратным понимается вызов, отнимающий много времени и ресурсов (чаще всего ресурсов памяти). Что именно оптимизировать — время выполнения или ресурсы, — зависит от самой программы, но в нашем случае требуется оптимизировать время работы функции.

Как работает мемоизация? Вам это уже известно, поскольку псевдокод, записанный на предыдущей странице, представляет собой реализацию мемоизации.

Чтобы все окончательно прояснить, давайте перепишем программу, опираясь на составленный псевдокод. Это будет несложно.

```
import time

cache = {}

def fibonacci(n):
    global cache
    if n in cache:
        return cache[n]

    if n == 0:
        result = 0
    elif n == 1:
        result = 1
    else:
        result = fibonacci(n-1) + fibonacci(n-2)
    cache[n] = result
    return result

start = time.time()

for i in range(0, 101):
    result = fibonacci(i)
    print(i, result)

finish = time.time()
duration = finish - start
print('Вычислили все 100 чисел за', duration, 'секунды')
```

Это словарь, который будет использоваться в качестве кеша

Первое, что нужно сделать, — это проверить, является ли число n ключом словаря cache. Если да, возвращаем значение ключа

Вместо того чтобы сразу же вернуть результат, если n равно 0 или 1, записываем значение n в локальную переменную result

Если требуется выполнить рекурсивные вычисления, записываем результат в ту же локальную переменную

Прежде чем вернуть результат, сохраняем его в кеше по ключу n. В Python ключом словаря может быть что угодно, даже целое число

Определим длительность вычисления всего списка из 100 чисел Фибоначчи

УСПЕШНЫЙ ОПЫТ С РЕКУРСИЕЙ

Все еще не верите, что несколько строк кода позволяют сократить длительность вычислений до 5 секунд? Внесите изменения в код и проверьте сами!

Оболочка Python

```
46 1836311903
47 2971215073
48 4807526976
49 7778742049
50 12586269025
51 20365011074
52 32951280099
53 53316291173
54 86267571272
55 139583862445
56 225851433717
57 365435296162
58 591286729879
59 956722026041
60 1548008755920
61 2504730781961
62 4052739537881
63 6557470319842
64 10610209857723
65 17167680177565
66 27777890035288
67 44945570212853
68 72723460248141
69 117669030460994
70 190392490709135
71 308061521170129
72 498454011879264
73 806515533049393
74 1304969544928657
75 2111485077978050
76 3416454622906707
77 5527939700884757
78 8944394323791464
79 14472334024676221
80 23416728348467685
81 37889062373143906
82 61305790721611591
83 99194853094755497
84 160500643816367088
85 259695496911122585
86 420196140727489673
87 679891637638612258
88 1100087778366101931
89 1779979416004714189
90 2880067194370816120
91 4660046610375530309
92 7540113804746346429
93 12200160415121876738
94 19740274219868223167
95 31940434634990099905
96 51680708854858323072
97 83621143489848422977
98 135301852344706746049
99 218922995834555169026
100 354224848179261915075
Вычислили все 100 чисел за 0.0005749298095703125 секунды
>>>
```

Оптимизация кода может оказать КОЛОССАЛЬНОЕ влияние на скорость вычислений

Тренируем мозг дальше

Как мы и обещали, в этой главе нам пришлось здорово поменять способ мышления. Но прежде чем завершить главу, нам хотелось бы показать, что рекурсия полезна не только для определения чисел Фибоначчи и палиндромов. Сейчас мы применим ее для построения таких графических объектов, как *фракталы*. Наверняка вы слышали о них, но мало кто знает, что они из себя представляют. Фрактал — это геометрический примитив, повторяющийся в любом масштабе. При увеличении фрактала вы получите ту же фигуру, что и при его уменьшении. Лучше всего познакомиться с фракталами, создав их самостоятельно, чем мы сейчас и займемся.

```
import turtle

def setup(pencil):
    pencil.color('blue')
    pencil.penup()
    pencil.goto(-200, 100)
    pencil.pendown()

def koch(pencil, size, order):
   if order == 0:
       pencil.forward(size)
   else:
       for angle in [60, -120, 60, 0]:
           koch(pencil, size/3, order-1)
           pencil.left(angle)

def main():
    pencil = turtle.Turtle()
    setup(pencil)

    order = 0
    size = 400
    koch(pencil, size, order)

if __name__ == '__main__':
    main()
    turtle.tracer(100)
    turtle.mainloop()
```

И снова черепашки

Задача функции `setup()` — задать цвет черепашки (здесь она называется `pencil`) и переместить ее в позицию, в которой рисование будет более центрированным

Это рекурсивная функция, к которой мы вернемся позже

Функция `main()` создает черепашку (объект `pencil`), определяет две переменные, `order` и `size`, и вызывает рекурсивную функцию, передавая ей три аргумента

Переменная `order` имеет начальное значение 0, а переменная `size` — 400. Вскоре вы узнаете, почему

Мы вызываем функцию `main()` и проверяем, чтобы была запущена функция `mainloop()` модуля `turtle`. Кроме того, мы используем не рассматривавшуюся ранее функцию `tracer()`, которая увеличивает скорость черепашки

Анализ функции koch()

Код, приведенный на предыдущей странице, очень простой: он создает черепашку, окрашивает ее в заданный цвет и изменяет ее положение. Остальные действия сводятся к вызову функции `koch()`. Но что она делает? Давайте разбираться...

Здесь анализируется только функция koch(). Программа целиком была приведена на предыдущей странице

Функция koch() получает аргументы pencil, size и order

```
def koch(pencil, size, order):
    if order == 0:
        pencil.forward(size)
    else:
        for angle in [60, -120, 60, 0]:
            koch(pencil, size/3, order-1)
            pencil.left(angle)
```

Простой случай

Рекурсия

Простейший случай: если аргумент order равен 0, рисуем прямую длиной size

В противном случае вызываем функцию koch() четыре раза, передавая ей аргумент size, деленный на 3, и уменьшая аргумент order на единицу

После каждого вызова функции koch() меняем угол движения черепашки

Нильс Фабиан Хельге фон Кох был шведским математиком, который впервые описал фрактальную кривую, названную "снежинкой Коха". Детали можно узнать по адресу https://ru.wikipedia.org/wiki/Кривая_Коха

Лучший способ понять это — увидеть несколько примеров

Итак, мы получили общее представление о том, как работает функция, но непонятно, что именно она делает. Мы лишь знаем, что работа функции сильно зависит от параметра `order`. Для начала присвоим ему значение `0` и посмотрим, что нарисует функция. Поскольку значение `0` соответствует простейшему случаю, должна быть нарисована прямая.

Введите код и запустите программу. Вот что мы получаем.

В простейшем случае рисуется отрезок длиной 400 пикселей (поскольку переменной size присвоено значение 400)

Если увеличить значение `order` до `1`, то функция будет вызвана рекурсивно. Измените значение локальной переменной `order` на `1` и еще раз запустите программу.

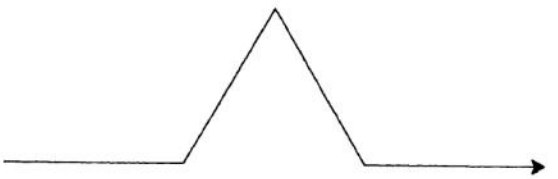
Когда аргумент order равен 1, имеет место рекурсия. В этом случае программа рисует четыре отрезка под разными углами. Рисование происходит при рекурсивных вызовах функции koch(), в каждом из которых выполняется простейший случай и строится отрезок длиной 400/3 пикселей

Возьмите карандаш

Если вы поняли, что нарисует функция `koch()`, когда аргумент `order` равен 0 и 1, то попробуйте предугадать, каким будет результат, когда аргумент `order` равен 2 и 3. Помните: при каждом значении масштаба фрактал выглядит одинаково. Это непростое упражнение, которое как нельзя лучше развивает вычислительное мышление.

Order = 0:

Order = 1:



Нарисуйте здесь свои догадки

Order = 2:

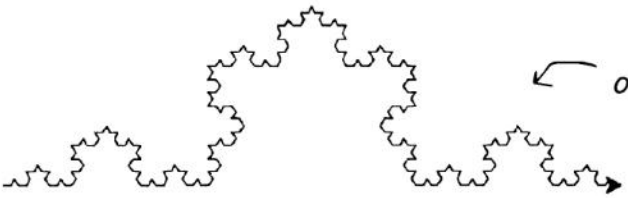
Подсказка: взгляните на то, что произошло на предыдущем уровне. Что, если повторить весь процесс для каждого отрезка предыдущего уровня?

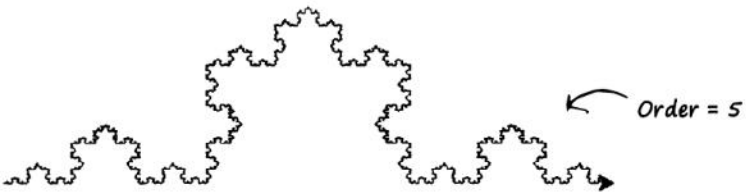
Order = 3:

Подсказка: что, если проделать то же самое на новом уровне?

Фрактал Коха

Надеемся, последнее упражнение помогло вам понять, что такое рекурсия и фракталы. Давайте для закрепления материала выполним программу, присвоив переменной `order` значения 4 и 5.





Снежинка Коха

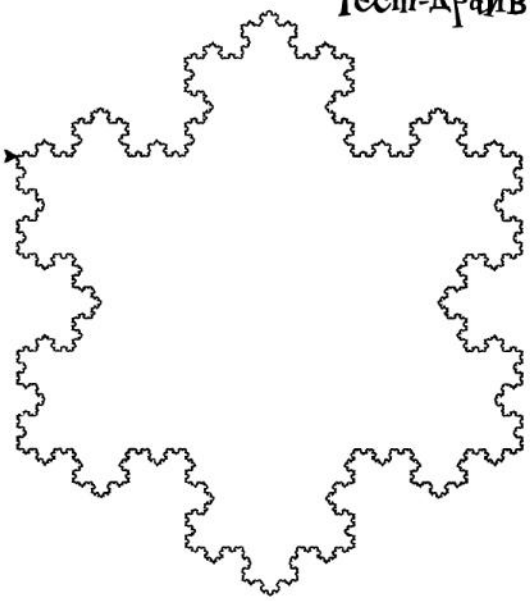
Подробнее о снежинке Коха рассказано в Википедии: `https://ru.wikipedia.org/wiki/Кривая_Коха`

Это ваше последнее задание: измените функцию `main()` так, чтобы функция `koch()` вызывалась в ней трижды, каждый раз с поворотом на 120 градусов.

```
def main():
    pencil = turtle.Turtle()
    setup(pencil)
    turtle.tracer(100)

    order = 5
    size = 400
    koch(pencil, size, order)

    for i in range(3):
        koch(pencil, size, order)
        pencil.right(120)
```

Неплохо для рекурсивной функции из шести строк!

Вызываем функцию `koch()` три раза, каждый раз поворачивая перо на 120 градусов



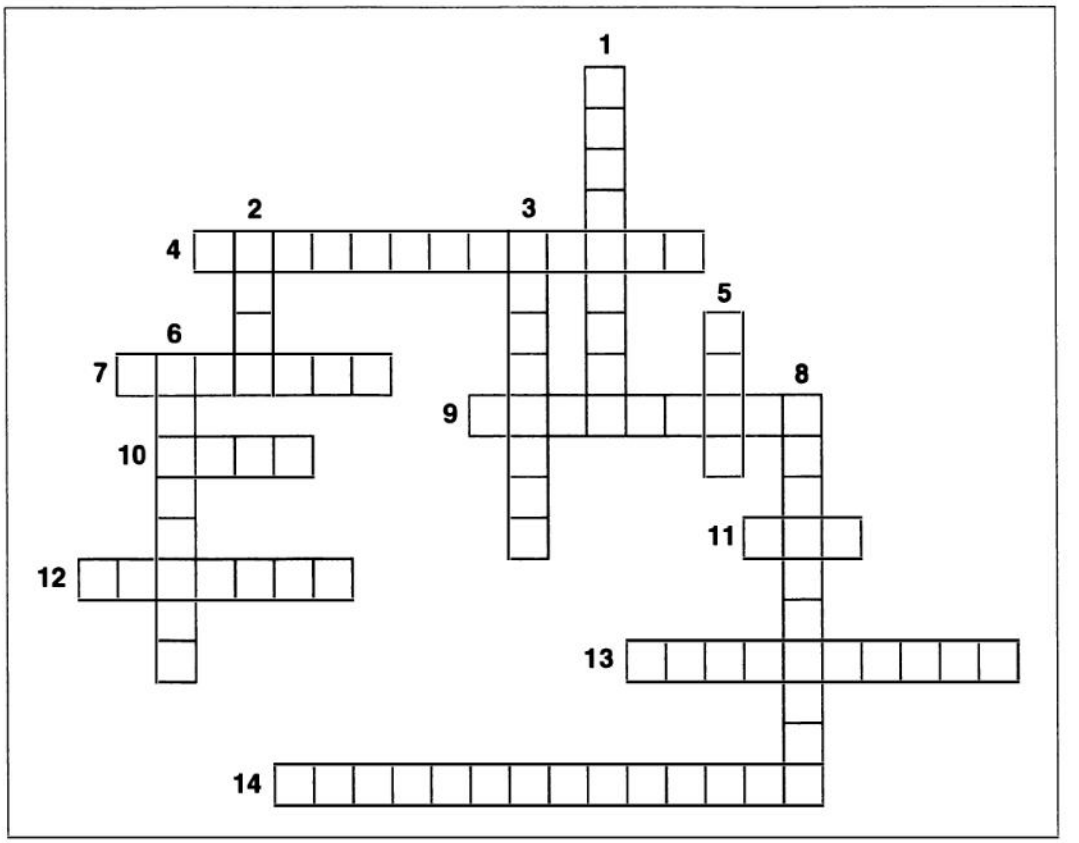
На этом мы завершаем знакомство с рекурсией, фракталами и словарями. Вы проделали серьезную работу, узнав много нового. Пора переключиться на что-то другое, но прежде подведем итоги и решим кроссворд.

САМОЕ ГЛАВНОЕ

- Циклы и рекурсия применяются для решения одних и тех же задач.
- Рекурсивная функция вызывает саму себя.
- Рекурсивное решение требует разделения задачи на простейший и рекурсивный случаи.
- В рекурсивном случае мы упрощаем задачу на один шаг и выполняем рекурсивный вызов функции.
- В языках программирования рекурсия поддерживается за счет сохранения параметров и локальных переменных в стеке вызовов функции.
- Иногда применение рекурсии приводит к неконтролируемому увеличению стека вызовов.
- Некоторые задачи проще решать с помощью рекурсии, а некоторые — с помощью циклов.
- В определенных алгоритмах рекурсия позволяет получить простое и понятное решение.
- Словари Python — это ассоциативные массивы.
- В словарях данные хранятся в виде пар "ключ: значение".
- Ключами словарей Python могут быть числа, строки и булевы значения. Значения, хранимые в словарях, могут иметь любой тип данных.
- Ключи словаря уникальны.
- При задании значения для уже существующего ключа оно заменяет исходное значение.
- Словари можно создавать программным путем или объявлять в явном виде.
- Ключи и связанные с ними значения можно удалять из словаря.
- Поиск ключа в словаре выполняется очень быстро (за константное время).
- В сочетании с мемоизацией словари применяются для оптимизации работы программы.
- Мемоизация заключается в сохранении результатов предыдущих вызовов функции.
- Избежав повторного вызова затратных функций, можно в ряде случаев существенно повысить скорость вычислений.
- Фрактал — это фигура, имеющая одинаковый вид при любом масштабе.

Кроссворд

Не волнуйтесь, это не рекурсивный кроссворд, а самый обычный.



По горизонтали

4. Характеристика словаря как массива
7. Повторяющая в любом масштабе фигура
9. Читается одинаково в обоих направлениях
10. Идентификатор элемента словаря
11. Шведский математик, в честь которого названа фрактальная фигура
12. Степень увеличения или уменьшения изображения
13. Характеристика ключа в словаре
14. Социальная сеть нового типа, которая рассматривалась в этой главе

По вертикали

1. Последовательность чисел, названная в честь итальянского математика
2. В нем хранятся параметры вызовов рекурсивных функций
3. Из них состоит цикл
5. Количество значений, составляющих элемент словаря
6. Вызов функции из самой себя
8. Сохранение результатов предыдущих вызовов функции

Возьмите карандаш
Решение

Отвлекитесь от рекурсии и представьте, что вам нужно написать функцию, которая проверяет, является ли слово палиндромом. Для этого можно использовать любые инструменты Python, изученные в главах 1–7. Чтобы описать ход ваших мыслей, составьте псевдокод решения.

Мы хотим сравнивать внешние символы и продвигаться внутрь, пока не будет найден несовпадающий символ или пока мы не достигнем середины, что означает палиндром

Можно создать индекс i, начинающийся с позиции 0...

...а также индекс j, начинающийся с конца

tacocat

Начинаем с обоих концов слова и сравниваем буквы попарно, продвигаясь снаружи внутрь

```
is_palindrome(слово):
    задать i = 0
    задать j = длина слова (минус 1)
    пока i < j:
        если символы в позициях i и j не равны, вернуть False
        увеличить i
        уменьшить j
    если цикл завершился, вернуть True
```

Если i >= j, значит, мы достигли середины строки и все сравнили

Если в любой точке внешние символы оказываются не равными друг другу, значит, слово не является палиндромом

Осталось написать сам код

```
def is_palindrome(word):
    i = 0
    j = len(word) - 1
    while i < j:
        if word[i] != word[j]:
            return False
        i = i + 1
        j = j - 1
    return True
```

Это должно работать, но приходится все время думать об индексах. Кроме того, код получился не самым понятным. Можно написать код и получше

Изучайте код, пока не поймете, как он работает

Возьмите карандаш
Решение

Попробуйте проанализировать рекурсивный код самостоятельно. Как насчет функции `recursive_compute_sum()`?

```
def recursive_compute_sum(list):
    if len(list) == 0:
        return 0
    else:
        first = list[0]
        rest = list[1:]
        sum = first + recursive_compute_sum(rest)
        return sum

recursive_compute_sum([1, 2, 3])
```

Уже знакомый вам код

Вызов рекурсивной функции

`recursive_compute_sum([1, 2, 3])`

Кадр 1
```
list = [1, 2, 3]
first = 1
rest = [2, 3]
```

Первый шаг сделан за вас. Параметр `list` содержит список `[1, 2, 3]`, также вычислены локальные переменные `first` и `rest`

`recursive_compute_sum([2, 3])`

Кадр 2
```
list = [2, 3]
first = 2
rest = [3]
```

Кадр 1
```
list = [1, 2, 3]
first = 1
rest = [2, 3]
```

Теперь рекурсивно вызываем функцию `recursive_compute_sum()`, чтобы добавить в стек новый кадр и новый список. На этот раз список равен `[2, 3]`

Как и прежде, необходимо вычислить переменные `first` и `rest`

`recursive_compute_sum([3])`

Кадр 3
```
list = [3]
first = 3
rest = []
```

Кадр 2
```
list = [2, 3]
first = 2
rest = [3]
```

Кадр 1
```
list = [1, 2, 3]
first = 1
rest = [2, 3]
```

Опять рекурсивно вызываем функцию `recursive_compute_sum()`, чтобы добавить в стек третий кадр и новый список. На этот раз список равен `[3]`

Как и прежде, необходимо вычислить переменные `first` и `rest`

recursive_compute_sum([])

Кадр4
list = []

Кадр3
list = [3]
first = 3
rest = []

Кадр2
list = [2, 3]
first = 2
rest = [3]

Кадр1
list = [1, 2, 3]
first = 1
rest = [2, 3]

Это последний вызов фукнции recursive_compute_sum(). Мы получили простейший сценарий, поскольку список теперь пуст

Функция сразу же завершается, поэтому переменные first и rest не вычисляются

recursive_compute_sum([3])

Кадр3
list = [3]
first = 3
rest = []
sum = 3

Кадр2
list = [2, 3]
first = 2
rest = [3]

Кадр1
list = [1, 2, 3]
first = 1
rest = [2, 3]

На предыдущем этапе функция завершилась, вернув 0, поэтому выталкиваем кадр из стека

Программа прибавляет 0 к переменной first и сохраняет результат в переменной sum

Функция возвращает значение sum

recursive_compute_sum([2, 3])

Кадр2
list = [2, 3]
first = 2
rest = [3]
sum = 5

Кадр1
list = [1, 2, 3]
first = 1
rest = [2, 3]

На предыдущем этапе функция вернула 3. Это значение прибавляется к переменной first, которая равна 2, и результат сохраняется в переменной sum

Возвращается сумма 5, и кадр выталкивается из стека

recursive_compute_sum([1, 2, 3])

Кадр1
list = [1, 2, 3]
first = 1
rest = [2, 3]
sum = 6

Полученное значение 5 прибавляется к переменной first, которая равна 1. Стек очищается, и результатом вызова функции recursive_compute_sum([1, 2, 3]) становится 6

Возьмите карандаш
Решение

Получив базовые знания о словарях, давайте применим их на практике. Проанализируйте приведенный ниже код и постарайтесь предугадать результаты его работы.

Вы получили такой результат?

Оболочка Python

```
Рекомендации фильмов
--------------------
Forbidden Planet (*****) 1956
Vertigo (*****) 1958
 >>>
```

Возьмите карандаш
Решение

Словарь в словаре — это популярное решение. Рассмотрим, насколько оно эффективно в нашем кинематографическом примере. Постарайтесь предугадать результаты работы приведенного ниже кода.

С двумя словарями получается сложнее?

Оболочка Python

```
Рекомендации фильмов
--------------------
Forbidden Planet (*****) 1956
Vertigo (*****) 1958
Выбор редакции
--------------
I Was a Teenage Werewolf (***) 1957
>>>
```

Тренируем мозг: решение

Теперь, когда мы определились со способом хранения данных о пользователях антисоциальной сети, можно приступать к написанию кода. Давайте напишем функцию `average_age()`, которая будет запрашивать имя пользователя и возвращать средний возраст его друзей.

```
users = {}
users['Ким'] = {'Эл. почта' : 'kim@oreilly.com', 'Пол': 'ж', 'Возраст': 27, 'Друзья': ['Джон', 'Джош']}
users['Джон'] = {'Эл. почта' : 'john@abc.com', 'Пол': 'м', 'Возраст': 24, 'Друзья': ['Ким', 'Джош']}
users['Джош'] = {'Эл. почта' : 'josh@wickedlysmart.com', 'Пол': 'м', 'Возраст': 32, 'Друзья': ['Ким']}

def average_age(username):
    global users

    user = users[username]
    friends = user['Друзья']

    sum = 0

    for name in friends:
        friend = users[name]
        sum = sum + friend['Возраст']

    average = sum/len(friends)
    print(username + ", средний возраст друзей:", average)

average_age('Ким')
average_age('Джон')
average_age('Джош')
```

Это наша функция, которая получает имя пользователя в строковом виде

Извлечем словарь атрибутов пользователя из словаря пользователей

А затем извлечем из словаря атрибутов список друзей

Локальная переменная для хранения суммарного возраста друзей

Пройдем по списку друзей

Необходимо взять имя друга и извлечь словарь его атрибутов из словаря пользователей

Далее извлекаем возраст друга из словаря атрибутов и прибавляем его к переменной sum

После того как возраст просуммирован, вычисляем среднее...

...и выводим результат

Это наш тестовый код

Вот каким должен быть результат работы программы

Оболочка Python

```
Ким, средний возраст друзей: 28.0
Джон, средний возраст друзей: 29.5
Джош, средний возраст друзей: 27.0
>>>
```

Возьмите карандаш

Решение

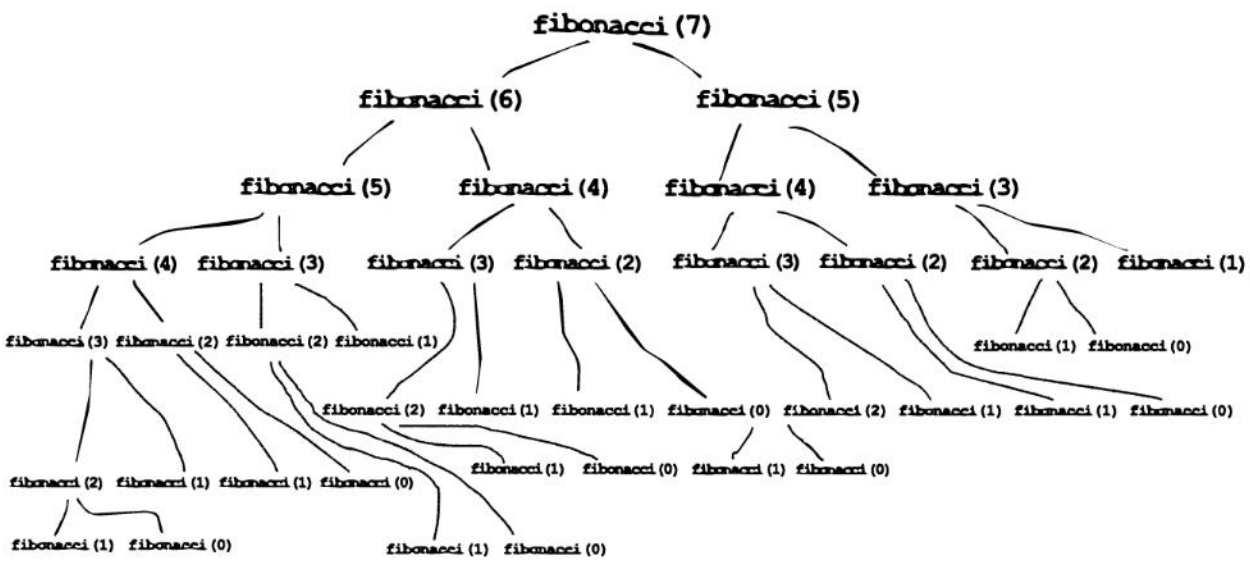
Имея перед глазами псевдокод, используйте свои знания о словарях, чтобы заполнить пропуски в приведенном ниже коде.

```
max = 1000
for name in ___users___:
    user = ___users___[___name___]
    friends = user[___'Друзья'___]
    if len(___friends___) < max:
        most_anti_social = ___name___
        max = len(___friends___)

print('Меньше всего друзей имеет', ___most_anti_social___)
```

Возьмите карандаш

Решение

Нарисуйте граф вызовов при получении числа `fibonacci(7)`. В конце подсчитайте, сколько повторных вызовов приходится на каждое из значений от 0 до 7.



Впишите сюда итоговые значения. В таблице уже дан ответ для чисел `fibonacci(7)` и `fibonacci(6)`, которые вычисляются по одному разу

fibonacci(7): *1 раз*	**fibonacci(6):** *1 раз*	**fibonacci(5):** *2 раза*
fibonacci(4): *3 раза*	**fibonacci(3):** *5 раз*	**fibonacci(2):** *8 раз*
fibonacci(1): *13 раз*	**fibonacci(0):** *8 раз*	

Да, это шокирует!

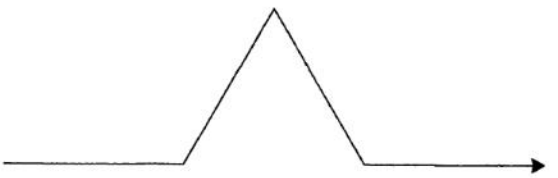
И угадайте, сколько раз в функции `fibonacci(50)` придется вычислить `fibonacci(3)`? *4 807 526 976 раз*

Возьмите карандаш
Решение

Если вы поняли, что нарисует функция `koch()`, когда аргумент `order` равен `0` и `1`, то попробуйте предугадать, каким будет результат, когда аргумент `order` равен `2` и `3`. Помните: при каждом значении масштаба фрактал выглядит одинаково. Это непростое упражнение, которое как нельзя лучше развивает вычислительное мышление.

Order = 0:

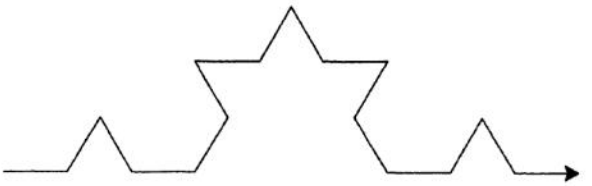
Order = 1:



На данном этапе мы нарисовали фигуру вида

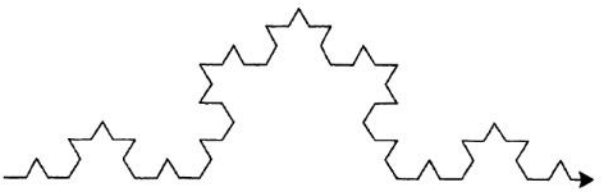
на каждом отрезке (он был только один)

Order = 2:



То же самое: для каждого отрезка предыдущего этапа рисуем фигуру вида

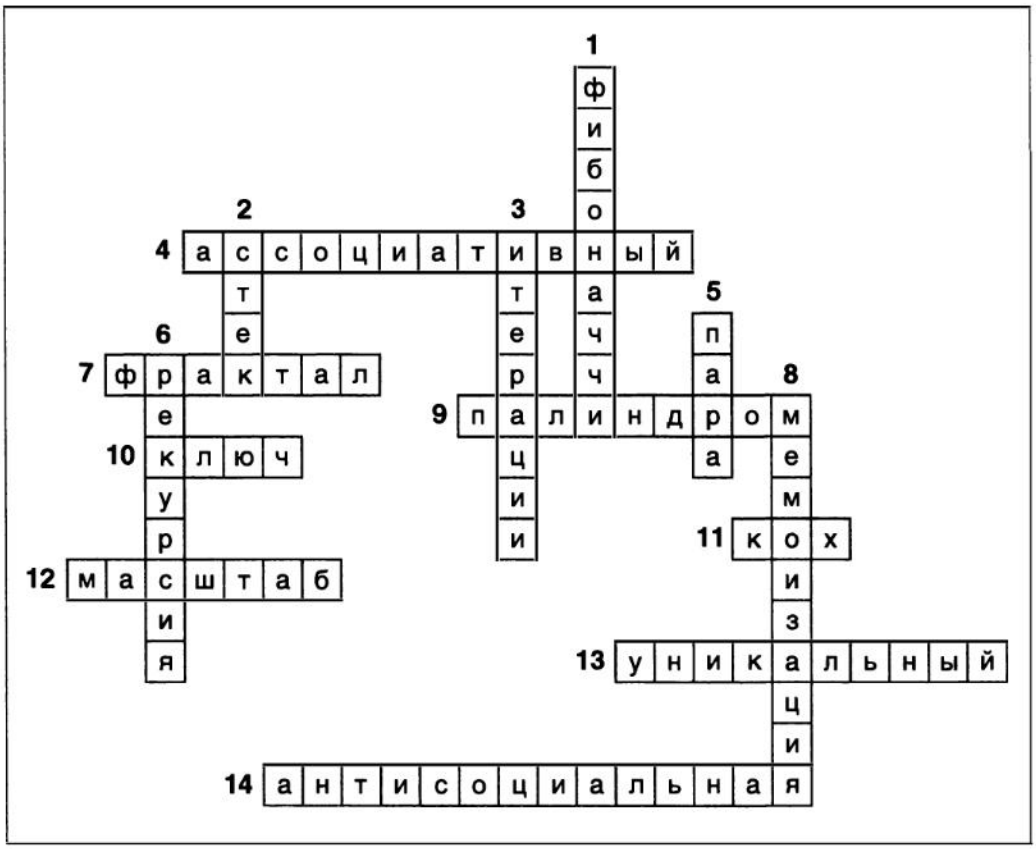
Order = 3:



Отрезки становятся все меньше, но мы продолжаем рисовать...

...уменьшая масштаб

Кроссворд: решение



9 Чтение и запись файлов

Длительное хранение данных

Вы умеете хранить данные в переменных, но как только программа завершается — бах! — они исчезают навсегда. Вот для чего нужно *постоянное* хранилище данных. Большинство компьютерных устройств оснащено такими хранилищами, в частности, жесткими дисками и флеш-накопителями. Кроме того, данные можно хранить в облачной службе. В этой главе вы узнаете, как писать программы, умеющие работать с файлами. Зачем это нужно? Вариантов множество: сохранение конфигурационных настроек, создание файлов отчетов, обработка графических файлов, поиск сообщений электронной почты. Список можно продолжать еще долго, но давайте перейдем к главному.

Игра в слова

Нет, это не каламбур. Нам предстоит создать собственную версию популярной игры Mad Libs, которую мы назовем *Смехослов*. Если вы никогда не играли в подобную игру, то вот ее правила.

"Смехослов" можно создать на основе любого текста; мы взяли готовый текст из книги

Для создания "Смехослова" удалите из текста произвольные слова и замените их прочерками с обозначениями частей речи (существительные, глаголы и пр.)

Правила игры

1. Попросите друга выбрать существительные, прилагательные и глаголы, которые будут подставляться вместо прочерков, но не показывайте ему сам текст.
2. Добавьте эти слова в текст и прочитайте получившуюся историю вслух.
3. Смейтесь!

СМЕХОСЛОВ

Первое, что необходимо сделать, чтобы _____ писать
ГЛАГОЛ

собственный _____ , — научиться _____
СУЩЕСТВИТЕЛЬНОЕ ГЛАГОЛ

стоящие перед вами задачи на _____ действия,
ПРИЛАГАТЕЛЬНОЕ

которые _____ сможет выполнить за вас. Для
СУЩЕСТВИТЕЛЬНОЕ

этого вы и компьютер должны _____ понятный друг
ГЛАГОЛ

другу _____ , но об этом мы вскоре поговорим.
СУЩЕСТВИТЕЛЬНОЕ

Текст первого абзаца главы 1

Ну допустим, я выбираю глагол "уйти", существительное "конспект", глагол "крошить", прилагательное "сумасшедшие", существительное "обезьяна", глагол "съесть" и существительное "помидор".

Получается: "Первое, что необходимо сделать, чтобы уйти писать собственный **конспект**, — научиться **крошить** стоящие перед вами задачи на **сумасшедшие** действия, которые **обезьяна** сможет выполнить за вас. Для этого вы и компьютер должны **съесть** понятный друг другу **помидор**, но об этом мы вскоре поговорим". Супер!

Логика игры

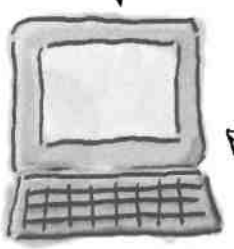
Для создания компьютерной игры "Смехослов" нужно написать код, который будет извлекать историю из текстового файла, запрашивать у пользователя недостающие слова и заполнять ими пропуски в тексте, создавая новый текстовый файл с получившейся историей. Рассмотрим алгоритм игры подробнее.

Под текстовым понимается файл, содержащий текст истории и хранящийся на компьютере

1 Составление шаблона.

Для начала нужно создать шаблон истории с заполнителями для слов, подставляемых пользователем. Нам потребуется как-то обозначить часть речи каждого заполнителя. Для этого мы вставим прямо в текст слова в верхнем регистре, например "СУЩЕСТВИТЕЛЬНОЕ", "ПРИЛАГАТЕЛЬНОЕ" и "ГЛАГОЛ".

Код игры



Интерпретатор Python

Пример шаблона

```
1
Первое, что необходимо сделать, чтобы
ГЛАГОЛ писать собственный СУЩЕСТВИТЕЛЬНОЕ, —
научиться ГЛАГОЛ стоящие перед вами задачи
на ПРИЛАГАТЕЛЬНОЕ действия, которые
СУЩЕСТВИТЕЛЬНОЕ сможет выполнить за вас.
Для этого вы и компьютер должны ГЛАГОЛ
понятный друг другу СУЩЕСТВИТЕЛЬНОЕ, но
об этом мы вскоре поговорим.
```

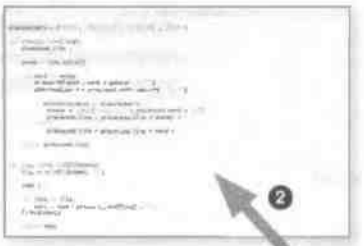
lib.txt

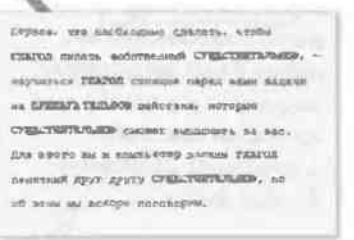
Шаблон — это обычный текстовый файл, содержащий текст с заполнителями для существительных, глаголов, прилагательных и других частей речи

Можете взять собственный текст и создать свой шаблон

❷ Чтение шаблона.

При запуске игры программа должна загрузить шаблон с диска и найти в нем все заполнители, т.е. слова в верхнем регистре ("СУЩЕСТВИТЕЛЬНОЕ", "ПРИЛАГАТЕЛЬНОЕ" и т.п.).





lib.txt

❸ Вывод запроса пользователю.

Для каждого заполнителя, найденного в тексте шаблона, программа должна запросить у пользователя соответствующее слово. Полученное слово нужно подставить в текст вместо заполнителя.

Получив данные от пользователя, программа создает новый файл и записывает в него обновленную версию текста

❹ Запись файла с обновленным текстом на диск.

После получения от пользователя всех слов, подставляемых вместо заполнителей, нужно создать новый файл и сохранить в нем итоговую историю.

Это заполненный "Смехослов", в котором каждый заполнитель заменен данными от пользователя

Первое, что необходимо сделать, чтобы
уйти писать собственный **конспект**, –
научиться **кромсать** стоящие перед вами задачи
на **сумасшедшие** действия, которые
обезьяна сможет выполнить за вас.
Для этого вы и компьютер должны **съесть**
понятный друг другу **поводор**, но
об этом мы вскоре поговорим.

Чтобы получить имя выходного файла, мы добавим префикс "crazy_" к имени исходного файла

crazy_lib.txt

Возьмите карандаш

У нас нет ни малейших сомнений в том, что, добравшись до главы 9, вы сможете самостоятельно написать псевдокод игры, план которой приведен на предыдущих страницах. Составление псевдокода даст вам ясное понимание того, как должна работать программа. Не пренебрегайте этим этапом! Нашу версию псевдокода вы найдете в конце главы.

Запишите здесь псевдокод игры. Помните, что текст истории хранится в файле, поэтому нужно сначала прочитать его, найти в нем заполнители, запросить у пользователя поставляемые слова и сохранить результат в новом файле

P.S. Вы еще не знаете, как работать с файлами, поэтому сконцентрируйтесь на описании общей логики игры, не заботясь о деталях (для этого и нужен псевдокод!)

Считывание истории из файла

Первое, что должна выполнить программа, — прочитать текстовый файл, содержащий исходную историю. В нашем случае он называется *lib.txt* и хранится в папке *ch9* примеров книги. Обязательно просмотрите его. Никто не запрещает использовать в игре собственные истории, но для целей тестирования лучше пока придерживаться шаблона *lib.txt*.

1. Загрузить текст истории из файла.
2. Обработать текст.
   Для каждого слова в тексте:
   - А. Если слово — заполнитель (ГЛАГОЛ, ПРИЛАГАТЕЛЬНОЕ или СУЩЕСТВИТЕЛЬНОЕ):
     1. Попросить пользователя ввести такую часть речи.
     2. Подставить введенное слово вместо заполнителя.
   - Б. В противном случае слово остается в тексте.
3. Сохранить результаты.
   Записать обработанный текст с подставленными заполнителями в файл, имя которого содержит префикс "crazy_".

```
Первое, что необходимо сделать, чтобы
ГЛАГОЛ писать собственный СУЩЕСТВИТЕЛЬНОЕ, —
научиться ГЛАГОЛ стоящие перед вами задачи
на ПРИЛАГАТЕЛЬНОЕ действия, которые
СУЩЕСТВИТЕЛЬНОЕ сможет выполнить за вас.
Для этого вы и компьютер должны ГЛАГОЛ
понятный друг другу СУЩЕСТВИТЕЛЬНОЕ, но
об этом мы вскоре поговорим.
```

lib.txt

Наш тестовый файл с заполнителями

Перед считыванием текстовый файл нужно открыть

Справедливо для всех языков программирования. Перед чтением или записью файл нужно открыть

Если мы собираемся получать данные из файла, нужно сначала открыть его. Что происходит при открытии файла? Интерпретатор Python должен найти целевой файл, убедиться в его существовании и запросить у операционной системы доступ к файлу. В конце концов, может оказаться, что вы не имеете права обращаться к файлу.

Для открытия файла применяется встроенная функция `open()`, которой передается имя файла и флаг *режима доступа*.

Используем функцию `open()` для открытия файла

Функции передается имя открываемого файла...

...и флаг режима доступа

```
my_file = open('lib.txt', 'r')
```

Режим может быть либо `'r'` (чтение из файла), либо `'w'` (запись в файл)

Функция `open()` возвращает файловый объект, который записывается в переменную `my_file`

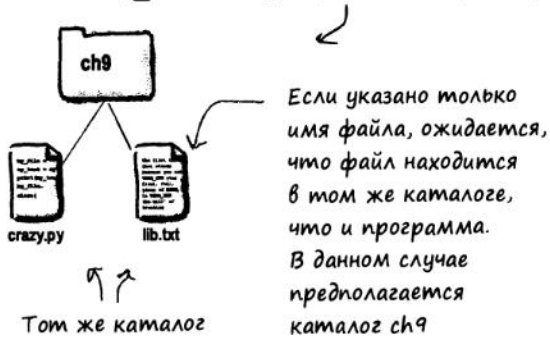
Имя файла может быть простым, если файл находится в том же каталоге, что и программа, или содержать путь. О том, что такое путь к файлу, мы поговорим далее

Термины "папка" и "каталог" являются синонимами, но при обсуждении путей к файлам чаще употребляют термин "каталог"

Путь к файлу

Вызвав функцию `open()` на предыдущей странице, мы передали ей только имя файла `'lib.txt'`, полагая, что он находится в текущем каталоге. Но что, если требуемый файл хранится в другом месте? Как его открыть? В подобных случаях к имени файла нужно добавить *путь*, сообщающий, где искать файл. Существуют два способа указания пути: *относительный* и *абсолютный*. В первом случае расположение файла задается относительно текущего каталога — того, из которого запускался код Python.

```
my_file = open('lib.txt', 'r')
```

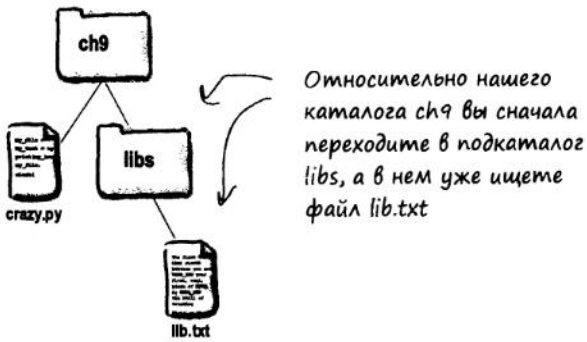


Относительный путь

Сначала познакомимся с относительным путем, который задается по отношению к текущему каталогу программы. Предположим, файл *lib.txt* находится в подкаталоге *libs*. Тогда функции `open()` нужно передать аргумент `'libs\lib.txt'`.

Относительный путь может включать произвольное количество вложенных папок, разделяемых символом обратной косой черты (\).

```
my_file = open('libs/lib.txt', 'r')
```



Разделители пути в Mac и Windows разные

В Mac (и Linux) имена папок разделяются косой чертой (/), а в Windows для этих же целей используется обратная косая черта (\). В Python поддерживается первый вариант, поэтому в Windows нужно будет переписать путь вида

```
C:\Users\eric\code\hfcode\ch9\lib.txt
```

в формате

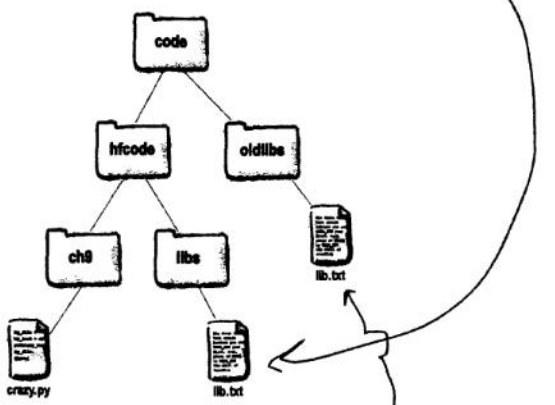
```
C:/Users/eric/code/hfcode/ch9/lib.txt
```

Но как быть, если нужно открыть файл, находящийся в соседнем каталоге или каталоге более высокого уровня? Все очень просто! Чтобы подняться на один уровень выше относительно текущего каталога, следует предварить путь двумя точками. Таким образом, если программа запускается из каталога *ch9* и требуется открыть в ней файл *lib.txt*, находящийся в соседнем каталоге *libs*, необходимо передать функции `open()` аргумент `'../libs/lib.txt'`. Он означает, что нужно подняться в родительский каталог, а затем перейти в подкаталог libs, в котором находится файл *lib.txt*.

Если же файл *lib.txt* находится в каталоге *oldlibs* более высокого уровня, то необходимо задать аргумент `'../../oldlibs/lib.txt'`.

Это означает подняться на два уровня вверх, а затем перейти в подкаталог oldlibs и найти в нем файл lib.txt

```
my_file = open('../libs/lib.txt', 'r')
```



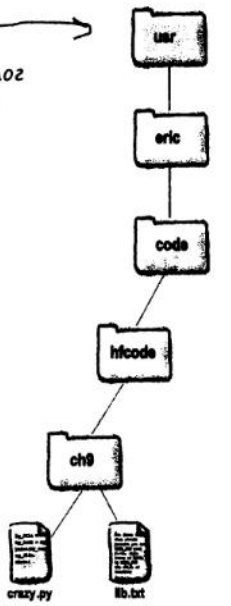

```
my_file = open('../../oldlibs/lib.txt', 'r')
```

Абсолютный путь

Корневым называется каталог, расположенный на самом верхнем уровне файловой системы

Абсолютный путь всегда начинается с корневого каталога файловой системы и задает точное расположение файла. Несмотря на более точную спецификацию файла, абсолютные пути в большинстве случаев менее удобны, ведь если программа переносится на другой компьютер, вам придется вручную менять все заданные в коде абсолютные пути. И все же в отдельных ситуациях абсолютные пути оказываются предпочтительнее относительных.

Это мой каталог верхнего уровня в macOS



Предположим, необходимо указать абсолютный путь к файлу *lib.txt*. На нашем компьютере это выглядит так:

```
my_file = open('/usr/eric/code/hfcode/ch9/lib.txt', 'r')
```

В Mac и Linux корневой каталог никак не обозначается — абсолютный путь начинается с косой черты, после которой указывается остальная часть пути в уже известном вам формате

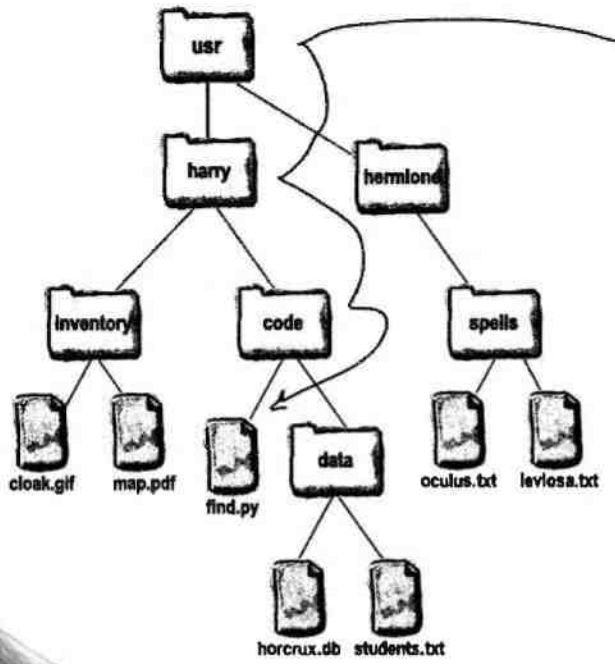
В Windows абсолютный путь выглядит следующим образом (предполагается, что программа запускается с диска `C:`):

```
my_file = open('C:/Users/eric/code/hfcode/ch9/lib.txt', 'r')
```

В Windows абсолютный путь начинается с имени диска, после которого ставится двоеточие и обратная косая черта. Далее указывается остальная часть пути в уже известном вам формате

Возьмите карандаш

Отследите, какому файлу соответствует каждый из приведенных ниже путей. В случае относительного пути начинайте с каталога code. Первое задание мы выполнили за вас.



`/usr/harry/code/find.py`

`../inventory/map.pdf`

`../../hermione/spells/leviosa.txt`

`find.py`

`data/students.txt`

Не забудьте все закрыть в конце

Прежде чем продолжить, следует усвоить важное правило программного этикета: по окончании работы с файлом его необходимо *закрыть*. Зачем? Под открытые файлы в операционной системе выделяются ресурсы, и если длительно выполняющаяся программа все время оставляет файлы открытыми, рано или поздно она может аварийно завершиться из-за нехватки ресурсов. Вот как закрыть файл.

Это переменная `my_file`, в которую записан файловый объект

Чтобы закрыть файл, достаточно вызвать метод `close()` файлового объекта

```
my_file.close()
```

Не забывайте об этом методе, так как он нам неоднократно понадобится

Чтение файла в коде

Для начала разберемся, что означает прочитать файл? В Python содержимое файла заносится в строку, с которой можно работать привычным способом.

Вся строка на странице не помещается, но смысл должен быть вам понятен: строка содержит весь текст из файла lib.txt

'Первое, что необходимо сделать, чтобы\nГЛАГОЛ писать собственный СУЩЕСТВИТЕЛЬНОЕ, —\nнаучиться...'

Это обычная строка Python, хранящая содержимое файла lib.txt

При чтении файла Python заносит все его содержимое в строку

Обратили внимание на странные символы \n в строке? Об этом речь пойдет далее...

Это наш текстовый файл

```
Первое, что необходимо сделать, чтобы
ГЛАГОЛ писать собственный СУЩЕСТВИТЕЛЬНОЕ, —
научиться ГЛАГОЛ стоящие перед вами задачи
на ПРИЛАГАТЕЛЬНОЕ действия, которые
СУЩЕСТВИТЕЛЬНОЕ сможет выполнить за вас.
Для этого вы и компьютер должны ГЛАГОЛ
понятный друг другу СУЩЕСТВИТЕЛЬНОЕ, но
об этом мы вскоре поговорим.
```

Чтение файла с помощью файлового объекта

Прочитать содержимое файла можно двумя способами: построчно или сразу весь целиком. Сначала узнаем, как загрузить файл целиком. Для этого применяется метод `read()` файлового объекта.

Существуют и другие способы чтения файлов, но мы рассмотрим только наиболее распространенные

Используем метод `read()` для чтения всего содержимого файла lib.txt...

...и записываем это содержимое в строку my_text

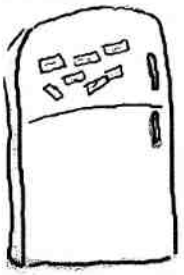
```
my_file = open('lib.txt', 'r')
my_text = my_file.read()
print(my_text)
my_file.close()
```

Не забудьте закрыть файл!

Далее выводим строку на экран, отображая все содержимое файла в оболочке Python

Оболочка Python

```
Первое, что необходимо сделать, чтобы
ГЛАГОЛ писать собственный СУЩЕСТВИТЕЛЬНОЕ, —
научиться ГЛАГОЛ стоящие перед вами задачи
на ПРИЛАГАТЕЛЬНОЕ действия, которые
СУЩЕСТВИТЕЛЬНОЕ сможет выполнить за вас.
Для этого вы и компьютер должны ГЛАГОЛ
понятный друг другу СУЩЕСТВИТЕЛЬНОЕ, но
об этом мы вскоре поговорим.
>>>
```

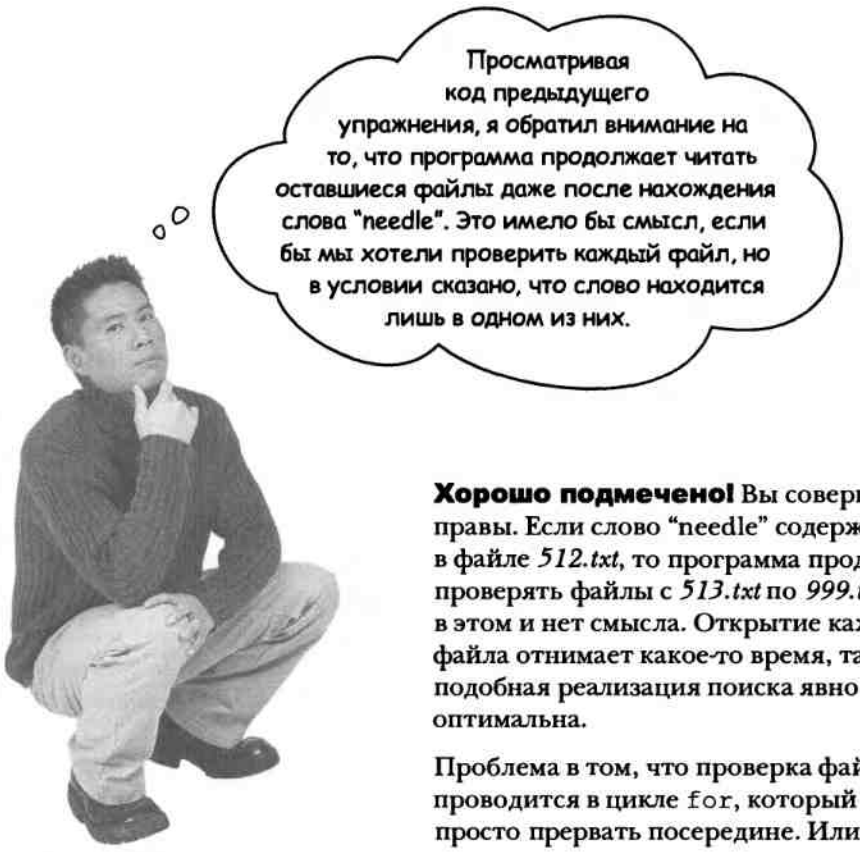


Код на магнитиках

Не поможете найти иголку в стоге сена? В рабочем каталоге находятся 1000 текстовых файлов с последовательными именами от *0.txt* до *999.txt*, и только в одном из них содержится слово `'needle'` (иголка). На магнитиках, прикрепленных к дверце холодильника, был записан код поиска такого файла, но вот незадача: кто-то случайно все перемешал. Сумеете восстановить код в прежнем виде? Учтите, что некоторые магнитики могут оказаться лишними. Сверьтесь с ответом, приведенным в конце главы.

```
for i in range(0, 1000):
file.close()
if 'needle' in text:
print('Нашел иголку в файле ' + str(i) + '.txt')
filename = str(i) + '.txt'
for i in range(0, 999):
text = file.read()
file = open(filename, 'r')
```

Расположите магнитики в нужном порядке



Хорошо подмечено! Вы совершенно правы. Если слово "needle" содержится в файле *512.txt*, то программа продолжит проверять файлы с *513.txt* по *999.txt*, хоть в этом и нет смысла. Открытие каждого файла отнимает какое-то время, так что подобная реализация поиска явно не оптимальна.

Проблема в том, что проверка файлов проводится в цикле `for`, который нельзя просто прервать посередине. Или можно?

Это подводит нас к тому, о чем мы еще не говорили ранее, — возможность прервать цикл, если становится понятно, что нет необходимости выполнять его до конца. Имея в своем распоряжении нечто подобное, мы могли бы легко исправить программу, чтобы она прекращала поиск сразу же после нахождения слова "needle". Разумеется, такая возможность существует: в Python, как и в большинстве языков программирования, имеется инструкция `break`, предназначенная как раз для этого. Познакомимся с ней поближе...

Пора прекратить...

Если мы используем цикл `for` для итерирования по диапазону чисел, то знаем, что цикл последовательно пройдет по каждому числу в диапазоне. Но иногда заранее становится понятно, что дальше продолжать нет смысла. Поиск слова "needle" в папке с файлами – как раз такой случай. Как только мы нашли его в одном из файлов, становится незачем открывать оставшиеся файлы.

Для прекращения цикла предназначена инструкция `break`. Она инициирует немедленный выход из цикла `for`, как показано ниже.

И обо мне не забудьте! Я тоже поддерживаю инструкцию `break`.

Цикл while

```
for i in range(0, 1000):
    filename = str(i) + '.txt'
    file = open(filename, 'r')
    text = file.read()
    if 'needle' in text:
        print('Нашел иголку в файле ' + str(i) + '.txt')
        break
    file.close()
print('Поиск завершен')
```

Это наш код поиска иголки

Как только программа доходит до инструкции `break`, цикл сразу же завершается

Как только иголка найдена, необходимо воспользоваться инструкцией `break` и обойти оставшуюся часть блока, а также выйти из цикла `for`

Сила мысли

Ой-ой-ой! Добавив в приведенный выше код инструкцию `break`, мы внесли небольшую ошибку. Сможете найти ее? Как ее исправить?

Подсказка: все ли файлы были закрыты?

Вернемся к игре

Представьте, что вы набираете текст на печатной машинке. Всякий раз при нажатии клавиши возврата каретки вы переходите на новую строку

Изучать возможности языка программирования всегда интересно, но нам нужно завершить реализацию игры. В последний раз мы занимались тем, что читали содержимое текстового файла целиком. Это несложно сделать, однако есть один недостаток: в случае больших файлов программе потребуются серьезные ресурсы. Представим файл, в котором сотни тысяч строк. Навряд ли стоит загружать в память такой объем данных, если вы не хотите, чтобы программа столкнулась с нехваткой памяти.

Более распространенный подход — считывать содержимое файла *построчно*. Но что в этом случае считается строкой? Весь текст вплоть до символа новой строки. Если вы откроете файл *lib.txt* в текстовом редакторе, где можно включить режим отображения скрытых символов, то вам откроется следующая картина.

```
Первое, что необходимо сделать, чтобы\n
ГЛАГОЛ писать собственный СУЩЕСТВИТЕЛЬНОЕ, -\n
научиться ГЛАГОЛ стоящие перед вами задачи\n
на ПРИЛАГАТЕЛЬНОЕ действия, которые\n
СУЩЕСТВИТЕЛЬНОЕ сможет выполнить за вас.\n
Для этого вы и компьютер должны ГЛАГОЛ\n
понятный друг другу СУЩЕСТВИТЕЛЬНОЕ, но\n
об этом мы вскоре поговорим.
```

lib.txt

Каждая строка завершается символом новой строки, которому соответствует управляющая последовательность \n

Вы не увидите символов новой строки в обычном режиме, потому что текстовые редакторы трактуют их как директиву отобразить следующий блок текста с новой строки

Метод `readline()` файлового объекта

Для построчного чтения файлов применяется метод `readline()` файлового объекта. Давайте используем его для чтения текста из файла *lib.txt*.

```
my_file = open('lib.txt', 'r')
line1 = my_file.readline()
print(line1)
line2 = my_file.readline()
print(line2)
```

На этот раз мы используем метод `readline()` и считываем первые две строки в переменные `line1` и `line2` соответственно

Файловый объект отслеживает текущую позицию при чтении файла, поэтому при каждом вызове метода `readline()` он продолжает чтение с текущей позиции

Оболочка Python

```
Первое, что необходимо сделать, чтобы

ГЛАГОЛ писать собственный СУЩЕСТВИТЕЛЬНОЕ, -
```

Сила мысли

Как вы думаете, почему в возвращаемом на предыдущей странице результате между двумя строками добавлена пустая строка?

Видите дополнительную пустую строку?

```
Оболочка Python
Первое, что необходимо сделать, чтобы

ГЛАГОЛ писать собственный СУЩЕСТВИТЕЛЬНОЕ, -
```

Умный термин, простая концепция.

Не существует способа набрать символ новой строки непосредственно с клавиатуры. Представьте: вы вводите открывающую кавычку, затем – текст строки, после чего нажимаете клавишу <Enter>, чтобы ввести символ новой строки, однако текстовый редактор просто переходит на следующую строку. Вот почему в качестве заменителя используется управляющая последовательность `\n`. Не воспринимайте ее как два символа: обратная косая черта и буква "n". Это способ записи одиночного символа новой строки. Например, если нужно вывести строку, после которой должны идти пять пустых строк, это можно сделать так.

```
print('Сейчас пойдут символы новой строки:\n\n\n\n\n')
```

Иногда последовательность `\n` называют переводом строки

Есть и другие управляющие последовательности. Например, `\t` – символ табуляции, `\b` – возврат на одну позицию, `\v` – вертикальная табуляция.

```
Оболочка Python
Сейчас пойдут символы новой строки:


>>>
```

Сколько символов новой строки стоит после текста? Пять? А вот и нет! Шесть! Пять символов содержатся в самой строке, но, как выясняется, функция `print()` добавляет еще один символ по умолчанию

Вот и ответ на упражнение "Сила мысли"

Как узнать, что мы достигли конца?

Метод `readline()` запоминает позицию последней прочитанной строки в файле. При каждом следующем вызове метод `readline()` переходит в конец последней прочитанной строки и считывает следующую строку. Если достигнут конец файла, метод вернет строку нулевой длины. Вот как организовать построчное чтение всего файла.

```
my_file = open('lib.txt', 'r')
while True:
    line = my_file.readline()
    if line != '':
        print(line)
    else:
        break
my_file.close()
```

Выражение `while True` означает, что цикл выполняется ВЕЧНО

Учтите, что пустая строка, в отличие от строки нулевой длины, будет содержать символ новой строки

На каждом шаге цикла мы считываем очередную строку и, если это не конец файла (нулевая строка), выводим ее

Если достигнут конец файла, используем удобную инструкцию `break` для выхода из цикла `while`. К счастью, вечность оказывается не бесконечной

Более простой способ — с помощью последовательностей Python

Очень хорошо, что вы так быстро освоили инструкцию `break`, но есть более простой и эффективный способ просмотра текстовых файлов в Python. Помните, как мы использовали цикл `for` для просмотра последовательностей (списков и строк)? Аналогичным образом можно поступать и с файлами, представив их в виде последовательности текстовых строк. Вот как будет выглядеть цикл `for`.

```
my_file = open('lib.txt', 'r')
for line in my_file:
    print(line)
my_file.close()
```

Так не только короче, но и понятнее!

По-серьезному

Всякий раз, используя цикл `for` с ключевым словом `in` по отношению к списку, строке, файлу или словарю, мы задействуем специальный механизм: так называемый *итератор*.

В случае итератора предполагается, что соответствующий тип данных поддерживает стандартный способ перебора последовательности значений. Не важно, как именно это реализовано, — цикл `for` все сделает за нас. Мы лишь знаем, что можем пройти по каждому значению, пока они не закончатся.

Итераторы имеются во многих современных языках программирования. Они основаны на высокоуровневых проектных решениях, которые называются *шаблонами проектирования*.

Чтение шаблона игры

Теперь вы знаете, как открывать файлы и читать их построчно. Давайте применим полученные знания для написания первой части игры в слова. Для начала создадим функцию `make_crazy_lib()`, которая получает имя файла и возвращает текстовый шаблон игры, в который вместо заполнителей подставлены все введенные пользователем слова.

Кроме того, нам понадобится создать вспомогательную функцию `process_line()`, отвечающую за обработку строк с заполнителями. Эта функция будет запрашивать все необходимые слова у пользователя и подставлять их в шаблон вместо заполнителей.

Вспомогательной называется функция, которая решает часть задачи, возложенной на основную функцию. В нашем случае функция `process_line()` отвечает за обработку строк с заполнителями в функции `make_crazy_lib()`

Вот наша функция; она получает имя файла и открывает его для чтения

Используем переменную `text` для формирования текста по мере его обработки

```
def make_crazy_lib(filename):
    file = open(filename, 'r')

    text = ''

    for line in file:
        text = text + process_line(line)

    file.close()

    return text

def process_line(line):
    return line

def main():
    lib = make_crazy_lib('lib.txt')
    print(lib)

if __name__ == '__main__':
    main()
```

Каждую строку файла мы будем обрабатывать с помощью функции `process_line()`, добавляя результат в формируемую переменную `text`

После обработки всех строк необходимо закрыть файл и вернуть переменную `text`

В целях тестирования сделаем так, чтобы функция `process_line()` просто возвращала переданную ей строку. Это позволит нам убедиться, что файл читается и конкатенация работает правильно

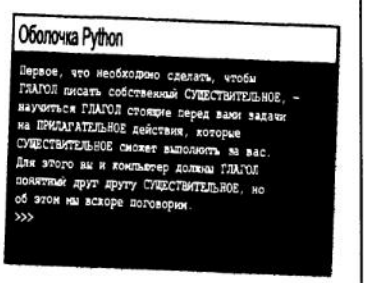
Необходимо вызвать функцию `make_crazy_lib()`. Для этого добавим функцию `main()`

Убедитесь в том, что файл lib.txt находится в каталоге программы

Тест-драйв

Сохраните приведенный выше код в файле *crazy.py*, запустите программу с помощью команды **Run > Run Module** и проверьте полученный результат.

Вы должны увидеть все содержимое файла вместе с заполнителями. Мы лишь занесли каждую строку файла в переменную `text`, не выполнив никакой обработки



Обработка текста шаблона

Итак, согласно написанному ранее псевдокоду на следующем этапе нам нужно обработать текст шаблона, т.е. заставить функцию `process_line()` выполнить возложенные на нее обязанности. Прежде всего (п. 2, А), требуется просмотреть все слова в каждой строке. К счастью, мы уже делали это в главе 6, когда разрабатывали программу, вычислявшую индекс удобочитаемости. В данном случае мы воспользуемся аналогичной методикой. Вот черновой вариант функции `process_line()`.

1. Загрузить текст истории из файла.
2. Обработать текст.
   Для каждого слова в тексте:
   А. Если слово — заполнитель (ГЛАГОЛ, ПРИЛАГАТЕЛЬНОЕ или СУЩЕСТВИТЕЛЬНОЕ):
      1. Попросить пользователя ввести такую часть речи.
      2. Подставить введенное слово вместо заполнителя.
   Б. В противном случае слово остается в тексте.
3. Сохранить результаты.
   Записать обработанный текст с подставленными заполнителями в файл, имя которого содержит префикс "crazy_".

Функция `processed_line()` получает строку текста

```
def process_line(line):
    processed_line = ''

    words = line.split()

    for word in words:


    return processed_line
```

Нам понадобится другая строка для хранения обработанной строки

Разобьем строку на список слов...

...и пройдем по всему списку

Здесь будет находиться код обработки

В конце возвращаем обработанную строку

Построчная обработка текста

Мы написали каркас кода, но что делать дальше? Согласно псевдокоду необходимо распознать заполнители и запросить у пользователя замещающие слова.

```
placeholders = ['СУЩЕСТВИТЕЛЬНОЕ', 'ПРИЛАГАТЕЛЬНОЕ', 'ГЛАГОЛ']

def process_line(line):
    global placeholders
    processed_line = ''

    words = line.split()

    for word in words:
        if word in placeholders:
            answer = input('Введите ' + word + ": ")
            processed_line = processed_line + answer + ' '
        else:
            processed_line = processed_line + word + ' '

    return processed_line + '\n'
```

Сохраним список заполнителей в глобальной переменной на случай, если он понадобится в другой функции

Проверяем каждое слово на предмет того, не является ли оно заполнителем

Если это заполнитель, выдаем запрос пользователю

Добавляем ответ в строку `processed_line`

В противном случае добавляем в строку исходное слово

Нужно также вставить в строку символ новой строки, потому что функция `split()` удаляет его

Пора протестировать игру в слова. Мы реализовали почти весь план, осталось лишь сохранить обработанный текст в файле. Внесите изменения в файл *crazy.py* (ниже приведена его полная версия), выполните команду **Run > Run Module** и проверьте полученный результат.

```
def make_crazy_lib(filename):
    file = open(filename, 'r')

    text = ''

    for line in file:
        text = text + process_line(line) + '\n'

    file.close()

    return text


placeholders = ['СУЩЕСТВИТЕЛЬНОЕ', 'ПРИЛАГАТЕЛЬНОЕ', 'ГЛАГОЛ']

def process_line(line):
    global placeholders
    processed_line = ''

    words = line.split()

    for word in words:
        if word in placeholders:
            answer = input('Введите ' + word + ": ")
            processed_line = processed_line + answer + ' '
        else:
            processed_line = processed_line + word + ' '

    return processed_line + '\n'

def main():
    lib = make_crazy_lib('lib.txt')
    print(lib)

if __name__ == '__main__':
    main()
```

Еще раз проанализируем код. В цикле for программа проходит по каждому слову в строке, проверяя, является ли оно заполнителем. Если да, просим пользователя ввести замещающее слово

Вот что мы получили. Странно! Похоже, несколько заполнителей были пропущены

У нас проблемы!

Оболочка Python

```
Введите ГЛАГОЛ: уйти
Введите ГЛАГОЛ: конспект
Введите ПРИЛАГАТЕЛЬНОЕ: крошить
Введите СУЩЕСТВИТЕЛЬНОЕ: обезьяна
Введите ГЛАГОЛ: съесть
Первое, что необходимо сделать, чтобы
уйти писать собственный СУЩЕСТВИТЕЛЬНОЕ, -
научиться конспект стоящие перед вами задачи
на крошить действия, которые
обезьяна сможет выполнить за вас.
Для этого вы и компьютер должны съесть
понятный друг другу СУЩЕСТВИТЕЛЬНОЕ, но
об этом мы вскоре поговорим.
>>>
```

Возьмите карандаш

Почему программа вдруг пропустила несколько существительных? В чем проблема: в словах или логике программы? Попробуйте выяснить это, сопоставив входные и выходные данные, и запишите свои соображения на этот счет.

У нас три заполнителя-существительных, но программа запросила только один

Заполнители-существительные чем-то отличаются друг от друга?

Или в программе какая-то ошибка?

Оболочка Python

```
Введите ГЛАГОЛ: уйти
Введите ГЛАГОЛ: конспект
Введите ПРИЛАГАТЕЛЬНОЕ: крошить
Введите СУЩЕСТВИТЕЛЬНОЕ: обезьяна
Введите ГЛАГОЛ: съесть
Первое, что необходимо сделать, чтобы
уйти писать собственный СУЩЕСТВИТЕЛЬНОЕ, -
научиться конспект стоящие перед вами задачи
на крошить действия, которые
обезьяна сможет выполнить за вас.
Для этого вы и компьютер должны съесть
понятный друг другу СУЩЕСТВИТЕЛЬНОЕ, но
об этом мы вскоре поговорим.
>>>
```

Исправление ошибки с помощью метода strip()

Мы решим проблему не так, как это делалось в главе 6, а воспользуемся еще неизвестным вам строковым методом `strip()`. Он возвращает переданную ему строку без начальных и конечных символов. Рассмотрим, как это работает.

Создадим несколько строк

```
hello = '!?Как вы там?!'
goodbye = '?Отлично! Все ОК!?!!'

hello = hello.strip('!?')
goodbye = goodbye.strip('!?')

print(hello)
print(goodbye)
```

Метод `strip()` удаляет из строки все указанные символы, если они являются начальными или конечными

Все вхождения восклицательного и вопросительного знаков удалены из начала и конца обеих строк

Оболочка Python

```
Как вы там
Отлично! Все ОК
```

Однако в строке goodbye остался один восклицательный знак, поскольку он находится в середине строки

Сила мысли

Используя метод `strip()`, переработайте функцию `process_line()` так, чтобы она корректно обрабатывала заполнители, заканчивающиеся знаками препинания (точка, запятая, двоеточие, вопросительный знак и пр.). Необходимо не только распознавать заполнители, но и сохранять знаки препинания в тексте. Постарайтесь решить задачу самостоятельно, а потом мы сделаем это вместе. Только чур не подглядывать!

```
def process_line(line):
    global placeholders
    processed_line = ''
    words = line.split()
    for word in words:
        if word in placeholders:
            answer = input('Введите ' + word + ":")
            processed_line = processed_line + answer + ' '
        else:
            processed_line = processed_line + word + ' '
    return processed_line + '\n'
```

Это код, который требуется улучшить

Окончательное исправление ошибки

Для исправления ошибки потребуется внести ряд изменений в программу. Это пример того, как небольшое изменение функциональности, будь то исправление ошибки или добавление новой возможности, вызывает целый каскад изменений в коде. Намного эффективнее было бы предусмотреть все необходимые действия еще на этапе составления псевдокода.

Чем больше изменений вносится в код, тем выше вероятность появления новых ошибок

Согласно предложенному решению мы будем сначала избавлять слова от знаков препинания, а затем сравнивать очищенные версии слов с заполнителями. По окончании обработки необходимо возвращать знаки препинания обратно. Вот как делается.

```
def process_line(line):
    global placeholders
    processed_line = ''
    words = line.split()
    for word in words:
        stripped = word.strip('.,;?!')
        if stripped in placeholders:
            answer = input('Введите ' + stripped + ":")
            processed_line = processed_line + answer
            if word[-1] in '.,;?!':
                processed_line = processed_line + word[-1] + ' '
            else:
                processed_line = processed_line + ' '
        else:
            processed_line = processed_line + word + ' '
    return processed_line + '\n'
```

Сначала очистим слово от всех знаков препинания (точек, запятых и т.п.)

Мы будем сравнивать с заполнителями очищенные версии слов

Отобразим очищенный заполнитель, не содержащий знаков препинания

Если имелся знак препинания, возвращаем его, после чего добавляем пробел, в противном случае просто добавляем пробел

Внесите описанные на предыдущей странице изменения в файл *crazy.py* и выполните команду **Run > Run Module**, чтобы проверить получаемый результат.

Чтобы протестировать собственную историю, вызовите функцию `make_crazy_lib()`, *передав ей свой файл*

Теперь все так, как нам нужно! Опробуйте собственные версии глаголов, существительных и прилагательных

Возьмите свой текст и проверьте на нем работу программы

Оболочка Python

```
Введите ГЛАГОЛ: уйти
Введите СУЩЕСТВИТЕЛЬНОЕ: конспект
Введите ГЛАГОЛ: крошить
Введите ПРИЛАГАТЕЛЬНОЕ: сумасшедшие
Введите СУЩЕСТВИТЕЛЬНОЕ: обезьяна
Введите ГЛАГОЛ: съесть
Введите СУЩЕСТВИТЕЛЬНОЕ: помидор
Первое, что необходимо сделать, чтобы
уйти писать собственный конспект, -
научиться крошить стоящие перед вами задачи
на сумасшедшие действия, которые
обезьяна сможет выполнить за вас.
Для этого вы и компьютер должны съесть
понятный друг другу помидор, но
об этом мы вскоре поговорим.
>>>
```

Остались проблемы посерьезнее

Попробуйте сделать следующее: откройте файл *crazy.py* и поменяйте в нем имя файла `'lib.txt'` на `'lib2.txt'`, после чего запустите программу.

Если попытаться открыть для чтения несуществующий файл, вы получите исключение `FileNotFoundError`

Оболочка Python

```
Traceback (most recent call last):
  File "crazy.py", line 45, in <module>
    main()
  File "crazy.py", line 41, in main
    crazy_lib = make_crazy_lib(filename)
  File "crazy.py", line 2, in make_crazy_lib
    file = open(filename, 'r')
FileNotFoundError: [Errno 2] No such file or directory: 'lib2.txt'
>>>
```

Помогли ли последние исправления избавиться от всех ошибок, связанных с обработкой знаков препинания? Могут ли возникать ситуации, при которых программа по-прежнему работает неправильно?

Подсказка: попробуйте подставить в задном слово 'ГЛАГОЛ!!!'. Сколько восклицательных знаков оставится в этом случае? Как исправить маску ошибку?

Учимся понимать Python

Потратьте время на выполнение небольшого упражнения. Оно позволит увидеть еще несколько распространенных ошибок, о которых часто сообщает интерпретатор Python. Возьмите на себя функции интерпретатора и попытайтесь предугадать его реакцию в процессе синтаксического анализа кода. Запишите предполагаемые результаты или сообщения об ошибках в приведенных ниже окнах.

Не помешает также выполнить код. Узнаете много интересного

```
list = [1, 2, 3, 4]
item = list[5]
```

Оболочка Python

Запишите здесь свои результаты

```
filename = 'документ' + 1 + '.txt'
```

Оболочка Python

```
int('1')
int('2')
int('E')
int('4')
int('5')
int('6')
```

Оболочка Python

```
msg = 'Привет!'

def hi():
    print(msg)
    msg = 'Салют!'

hi()
```

Оболочка Python

```
firstname = 'Бетховен'
print('Имя: ' + name)
```

Оболочка Python

Обработка исключений

Ранее мы уже обсуждали ошибки: синтаксические (чаще всего опечатки), семантические (логические упущения) или выполнения (возникают на этапе запуска кода). Ошибки выполнения мы никак не обрабатывали: интерпретатор Python выдавал соответствующее сообщение, и программа просто прекращала свою работу. Но так не должно быть, особенно если ошибки возникают по естественным причинам, например, при попытке открыть несуществующий файл. Сейчас мы узнаем, как правильно обрабатывать такие ошибки.

Но сначала давайте разберемся, что собой представляет ошибка выполнения, или *исключение*. Это событие, возникающее во время работы программы и означающее ситуацию, с которой не может справиться интерпретатор Python. Столкнувшись с ошибками такого типа, интерпретатор прекращает выполнение программы и создает объект исключения, содержащий сведения о возникшей ошибке. По умолчанию сведения об ошибках выводятся в виде трассировочных сообщений, которые вы уже наблюдали в оболочке Python.

Такие ошибки вполне можно обрабатывать вручную, избавляя интерпретатор от необходимости останавливать работу программы. Фактически вы сообщаете интерпретатору о том, что берете на себя обработку исключений определенного типа. В Python обработка исключений выполняется в блоке инструкций `try/except`.

Начните с ключевого слова `try`...

...после чего введите блок кода, в котором может возникнуть ошибка

```
try:
    filename = 'notthere.txt'
    file = open(filename, 'r')
except:
    print('Возникла ошибка при открытии', filename)
else:
    print('Ура, файл удалось открыть!')
    file.close()
```

Далее укажите ключевое слово `except`...

...и введите блок кода, который будет выполнен в случае ошибки

Далее добавьте условный блок `else`, который выполняется, только если не было ошибок

Можно еще добавить необязательный блок `finally`, который выполняется независимо от того, возникло исключение или нет

Явная обработка исключений

Если после ключевого слова `except` задан блок кода, то это означает универсальную обработку. Другими словами, блок кода будет выполнен для любого исключения, которое может возникнуть в блоке `try`. Но ничто не мешает указать конкретное обрабатываемое исключение, как показано ниже.

```
try:
    filename = 'notthere.txt'
    file = open(filename, 'r')
except FileNotFoundError:
    print('Файл', filename, 'не найден.')
except IsADirectoryError:
    print("Это каталог, а не файл!")
else:
    print("Хорошо, что файл удалось открыть.")
    file.close()
finally:
    print("Я выполняюсь в любой ситуации.")
```

Это снова наш блок `try`

На этот раз указываем конкретное исключение в инструкции `except`

Этот код выполняется, только если возникает исключение `FileNotFoundError`

Можно добавить и другие исключения. Данный блок выполняется, только если попытаться открыть каталог вместо файла

Как и раньше, если все хорошо, выполняется блок `else`

Вот как добавляется блок `finally`. Он выполняется независимо от того, возникло исключение или нет

Не бойтесь задавать вопросы

В: Существует ли ограничение на количество открываемых файлов в программе?

О: В Python нет такого ограничения, но оно устанавливается операционной системой. Это еще одна причина своевременно закрывать любые файлы, больше не используемые программой. Конечно, лимит всегда можно увеличить, изменив системные настройки, но у вас должны быть на то веские причины.

В: Где можно просмотреть список существующих исключений?

О: Такой список доступен по адресу `https://docs.python.org/3/library/exceptions.html`.

В: Можно ли создавать собственные типы исключений?

О: Да. Исключения представляют собой обычные объекты, поэтому можно расширять возможности Python, создавая собственные объекты исключений. Данная тема выходит за рамки главы; многочисленные руководства можно найти в Интернете и на сайте `python.org`.

В: Можно ли обрабатывать в одном блоке инструкций сразу несколько исключений?

О: После ключевого слова `except` допускается указывать несколько исключений. Они должны быть взяты в скобки и разделены запятыми:

```
except (FileNotFoundError, IOError):
```

Можно вообще не указывать имена исключений, в таком случае блок инструкций будет выполнен для любого возникшего исключения.

Возьмите карандаш

Трижды проследите за выполнением приведенного ниже кода. В первом случае введите любое число, не равное нулю, во втором — число 0, а в третьем — строку "ноль". Укажите, какой результат вы ожидаете получить в каждом случае.

```
try:
    num = input('Введите число: ')
    result = 42 / int(num)
except ZeroDivisionError:
    print("Нельзя делить на ноль!")
except ValueError:
    print("Простите, но нужно ввести число.")
else:
    print('Ваш ответ:', result)
finally:
    print('Спасибо, что были с нами!')
```

Введите число, отличное от 0

Оболочка Python

Введите 0

Оболочка Python

Введите строку "ноль"

Оболочка Python

Обработка исключений в игре в слова

Теперь, когда мы познакомились с исключениями, давайте обновим код открытия файла, добавив обработку ряда исключений, связанных с файлами.

```
def make_crazy_lib(filename):
    try:
        file = open(filename, 'r')
        text = ''

        for line in file:
            text = text + process_line(line)
        file.close()

        return text

    except FileNotFoundError:
        print("Не удалось найти файл", filename + '.')
    except IsADirectoryError:
        print("Вообще-то,", filename, '- это каталог.')
    except:
        print("Не удалось прочитать файл", filename)
```

Функция работает с файлом, поэтому имеет смысл включить весь код в блок `try`

Проверяем, не возникла ли ошибка поиска файла и не попытался ли пользователь открыть каталог. (Вдруг он поменял имя файла в коде?)

Перехватываем другие исключения, которые могут произойти при обработке файла

Заметьте, здесь нет блока `else` или `finally`

Данные изменения повышают надежность функции `make_crazy_lib()`, но мы упустили кое-что важное. Какое значение должна возвращать функция при возникновении исключения? Как известно, если инструкция `return` не указана в явном виде, то функция возвращает значение `None`. Не забывайте об этом факте — нам придется его использовать при сохранении файла.

В главе 5 мы говорили о том, что будем время от времени сталкиваться со значением **None**

Добавьте приведенный выше код в файл *crazy.py*, запустите программу и проверьте полученный результат. Попробуйте в качестве шаблона указать несуществующий файл или каталог и посмотрите, что произойдет.

Вы должны получить тот же результат, что и ранее. Можете изменить имя файла шаблона в программе crazy.py, чтобы проверить, как работает перехват исключений

Оболочка Python

```
Введите ГЛАГОЛ: уйти
Введите СУЩЕСТВИТЕЛЬНОЕ: конспект
Введите ГЛАГОЛ: кропить
Введите ПРИЛАГАТЕЛЬНОЕ: сумасшедшие
Введите СУЩЕСТВИТЕЛЬНОЕ: обезьяна
Введите ГЛАГОЛ: съесть
Введите СУЩЕСТВИТЕЛЬНОЕ: помидор
Первое, что необходимо сделать, чтобы
уйти писать собственный конспект, -
научиться кропить стоящие перед вами задачи
на сумасшедшие действия, которые
обезьяна сможет выполнить за вас.
Для этого вы и компьютер должны съесть
понятный друг другу помидор, но
об этом мы вскоре поговорим.
>>>
```

Последний этап: сохранение результата

Чтобы сохранить результаты игры или любой другой текст в файле, достаточно вызвать метод `write()` файлового объекта. Предварительно файл должен быть открыт в *режиме записи*. Давайте напишем функцию `save_crazy_lib()`, получающую в качестве аргументов имя файла и строку. Функция должна будет создать новый файл с указанным именем, сохранить в нем переданную строку и закрыть файл. Код функции показан ниже.

1. Загрузить текст истории из файла.
2. Обработать текст.
   Для каждого слова в тексте:
   A. Если слово — заполнитель (ГЛАГОЛ, ПРИЛАГАТЕЛЬНОЕ или СУЩЕСТВИТЕЛЬНОЕ):
      1. Попросить пользователя ввести такую часть речи.
      2. Подставить введенное слово вместо заполнителя.
   B. В противном случае слово остается в тексте.
3. Сохранить результаты.
   Записать обработанный текст с подставленными заполнителями в файл, имя которого содержит префикс "crazy_".

Функция `save_crazy_lib()` получает имя файла и строку текста

```
def save_crazy_lib(filename, text):
    file = open(filename, "w")

    file.write(text)
    file.close()
```

Сначала открываем файл в режиме 'w', чтобы можно было записывать в него. Python создаст файл, если он не существует

Используем метод `write()` файлового объекта, передавая ему строку текста, которая записывается в файл

Наконец, закрываем файл

Во многих языках программирования, если файл, в который велась запись, не был закрыт, то фактическая запись данных не гарантируется

Будьте осторожны, открывая файл в режиме 'w'

При открытии существующего файла в режиме 'w' его содержимое будет полностью удалено и заменено новыми данными. Учитывайте это!

Обновление остального кода

После того как функция `save_crazy_lib()` создана, ее необходимо вызвать из функции `main()`, но сначала нужно убедиться в том, что функция `make_crazy_lib()` завершила подстановку слов в шаблон (не забывайте, что в случае возникновения ошибки она вернет значение `None`).

Сохраним имя файла в переменной, чтобы с ним было удобнее работать

```
def main():
    filename = 'lib.txt'
    lib = make_crazy_lib(filename)
    print(lib)
    if (lib != None):
        save_crazy_lib('crazy_' + filename, lib)
```

Если при открытии или чтении файла возникает исключение, переменная `lib` будет содержать значение `None`, поэтому необходимо проверить переменную, прежде чем передавать ее функции `save_crazy_lib()`

К оригинальному имени файла добавляется префикс "crazy_". Это будет имя нового файла, в котором сохраняется текст

Пора целиком протестировать нашу игру, поскольку теперь все готово. Нужно только внести последние изменения в файл crazy.py. Ниже приведен весь код целиком, в который мы добавили дополнительную обработку исключений. Обновите файл и попробуйте поиграть.

```
def make_crazy_lib(filename):
    try:
        file = open(filename, 'r')

        text = ''

        for line in file:
            text = text + process_line(line)

        file.close()

        return text

    except FileNotFoundError:
        print("Не удалось найти файл", filename + '.')
    except IsADirectoryError:
        print("Вообще-то,", filename, '- это каталог.')
    except:
        print("Не удалось прочитать файл", filename)

placeholders = ['СУЩЕСТВИТЕЛЬНОЕ', 'ПРИЛАГАТЕЛЬНОЕ', 'ГЛАГОЛ']

def process_line(line):
    global placeholders
    processed_line = ''

    words = line.split()

    for word in words:
        stripped = word.strip('.,;?!')
        if stripped in placeholders:
            answer = input('Введите ' + stripped + ":")
            processed_line = processed_line + answer
            if word[-1] in '.,;?!':
                processed_line = processed_line + word[-1] + ' '
            else:
                processed_line = processed_line + ' '
        else:
            processed_line = processed_line + word + ' '
    return processed_line + '\n'

def save_crazy_lib(filename, text):
    try:
        file = open(filename, 'w')
        file.write(text)
        file.close()
    except:
        print("Не удалось записать файл", filename)

def main():
    filename = 'lib.txt'
    lib = make_crazy_lib(filename)
    print(lib)
    if (lib != None):
        save_crazy_lib('crazy_' + filename, lib)

if __name__ == '__main__':
    main()
```

Код обработки исключения уже добавлен в функцию `save_crazy_lib()` *для вас. Проверьте его*

Мне бы хотелось запускать игру с разными шаблонами, но нет желания открывать исходный файл Python и менять в нем имя файла. А можно передавать имя файла шаблона в командной строке?

Да, можно, и вы должны научиться этому.

Раньше мы не запускали программы из командной строки, но если открыть консоль и перейти в каталог, в котором находится файл *crazy.py*, то можно будет запустить программу с помощью следующей инструкции в Mac:

```
python3 crazy.py
```

Или в Windows:

```
python crazy.py
```

Мы хотим, чтобы с помощью аргумента командной строки можно было указать имя файла шаблона, например:

```
python3 crazy.py lib.txt
```

Запускаем программу crazy.py, используя файл lib.txt в качестве шаблона

Чтобы извлечь аргумент `lib.txt` из командной строки, необходимо подключить встроенный модуль `sys`, в котором поддерживается атрибут `argv`. Этот аргумент хранит список значений, введенных в командной строке (за исключением команды вызова Python). Например, если ввести

```
python3 crazy.py lib.txt
```

то в элемент 0 списка `argv` будет занесено значение `crazy.py`, а в элемент 1 — значение `lib.txt`. Воспользуемся полученными знаниями, чтобы завершить игру в слова.

Обновим в последний раз код игры, чтобы иметь возможность указывать имя файла шаблона в командной строке. Для этого придется внести два простых изменения, которые описаны ниже. Обновите соответствующим образом файл *crazy.py* и проведите финальный тест-драйв игры.

1 Вначале импортируйте модуль `sys`, добавив в начало программы инструкцию `import sys`.

```
import sys
```

Добавьте это в начало файла

2 Внесите изменения в функцию main() файла crazy.py.

```
def main():
    if len(sys.argv) != 2:
        print("crazy.py <имя_файла>")
    else:
        filename = sys.argv[1]
        lib = make_crazy_lib(filename)
        if (lib != None):
            save_crazy_lib('crazy_' + filename, lib)
```

Нам нужно, чтобы аргументов было два, иначе пользователь не предоставил имя файла. В таком случае выводим напоминание

В противном случае имя файла является аргументом командной строки с индексом 1

На этот раз необходимо запустить Python из командной строки. В Windows щелкните на кнопке Пуск, введите в поле поиска "cmd" и щелкните на появившемся имени программы

Консоль

```
$ cd /Users/eric/code/ch9
$ python3 crazy.py lib.txt
Введите ГЛАГОЛ: сбежать
Введите СУЩЕСТВИТЕЛЬНОЕ: хот-дог
Введите ГЛАГОЛ: выпить
Введите ПРИЛАГАТЕЛЬНОЕ: прикольные
Введите СУЩЕСТВИТЕЛЬНОЕ: гамбургер
Введите ГЛАГОЛ: разбить
Введите СУЩЕСТВИТЕЛЬНОЕ: бокал
$ cat lib.txt
Первое, что необходимо сделать, чтобы
сбежать писать собственный хот-дог, -
научиться выпить стоящие перед вами задачи
на прикольные действия, которые
гамбургер сможет выполнить за вас.
Для этого вы и компьютер должны разбить
понятный друг другу бокал, но
об этом мы вскоре поговорим.
$
```

Необходимо перейти в каталог, в котором содержится файл crazy.py, с помощью системной команды cd

Это командная строка в Mac

В Windows нужно ввести python, а не python3

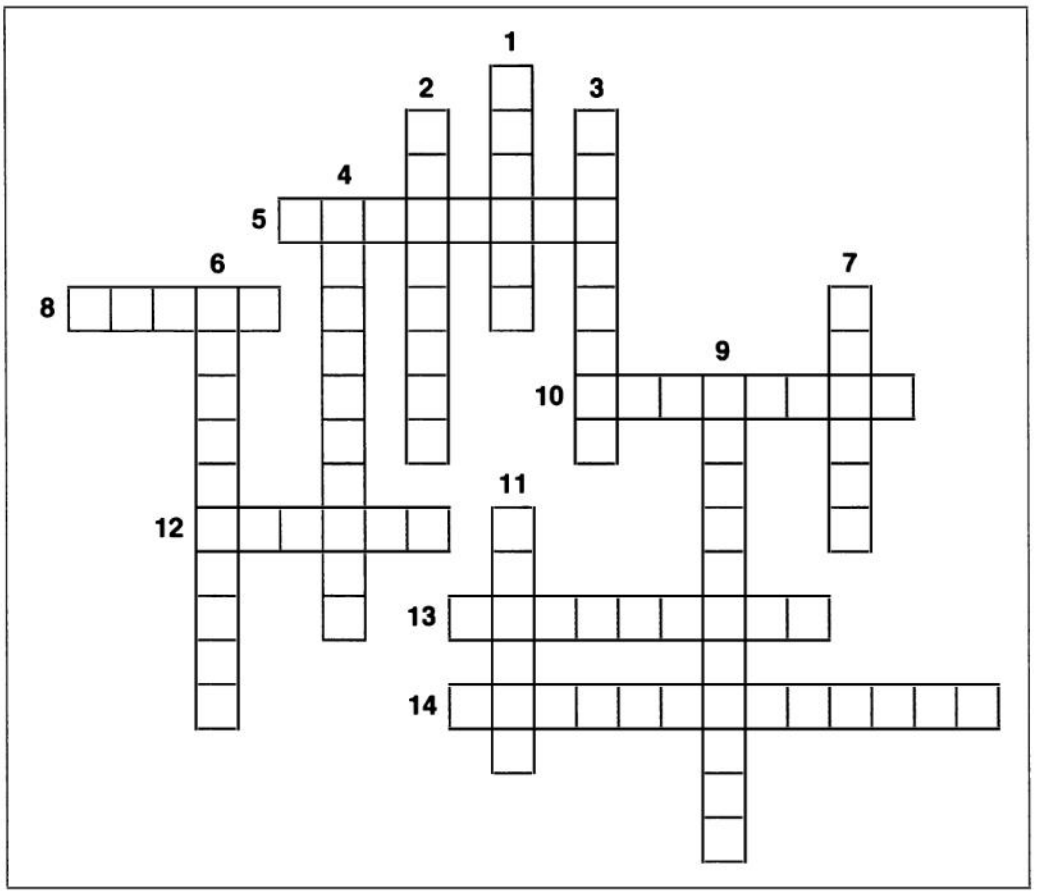
Это содержимое файла crazy_lib.txt

САМОЕ ГЛАВНОЕ

- Чтобы в программе получить доступ к файлу, его нужно сначала открыть.
- Для открытия файла предназначена встроенная функция `open()`.
- Файл можно открыть в режиме чтения (`'r'`) или записи (`'w'`).
- В качестве аргумента функция `open()` получает либо относительный, либо абсолютный путь к файлу.
- По окончании работы с файлом его нужно закрыть, вызвав метод `close()` файлового объекта.
- Функция `open()` поддерживает несколько способов чтения стандартных текстовых файлов.
- Для чтения всего содержимого файла применяется метод `read()`.
- Попытка прочитать файл целиком может оказаться слишком ресурсоемкой операцией в случае больших файлов.
- С помощью метода `readline()` можно прочитать файл построчно.
- По достижении конца файла метод `readline()` возвращает пустую строку.
- Текстовый файл является последовательностью строк, которую можно просматривать в цикле `for` (`for line in file:`).
- Просмотр последовательности строк в инструкции `for/in` становится возможным благодаря итератору.
- Инструкция `break` позволяет досрочно прервать выполнение цикла `for` или `while`.
- В большинстве текстовых файлов строки разделяются символом новой строки.
- Символу новой строки соответствует управляющая последовательность `\n`.
- Метод `strip()` удаляет ноль или больше экземпляров заданных символов в начале и конце строки. Если конкретные символы не указаны, то по умолчанию удаляются пробелы.
- Для перехвата исключений применяется инструкция `try/except`. В блок `try` помещается код, который может вызвать исключение, а в блок `except` — код обработки ошибок.
- Если после ключевого слова `except` не указаны перехватываемые исключения, то блок обработки выполняется для любого исключения.
- Блок `finally` выполняется всегда, независимо от того, возникло исключение или нет.
- В модуле `sys` имеется атрибут `argv`, в котором хранится список аргументов командной строки.
- В списке `argv` перечислены все слова, введенные в командной строке при вызове программы.

Кроссворд

Отдохните от работы с файлами, решив очередной кроссворд.



По горизонтали

5. Операция, предваряющая завершение работы с файлом
8. Определяет способ открытия файла
10. Операция, предваряющая чтение файла
12. Операция получения данных из файла
13. Веселая игра в слова
14. Путь, записанный в сокращенном виде

По вертикали

1. Строка, не содержащая символов
2. Обработка исключений в программе
3. Шаблон проектирования, позволяющий автоматически просматривать последовательности
4. Путь с полным указанием расположения файла
6. Ошибочная ситуация при выполнении кода
7. Тип данных, с помощью которого хранятся аргументы командной строки
9. Символ, вставляемый между именами папок в записи пути
11. Место, где хранятся файлы

Возьмите карандаш
Решение

У нас нет ни малейших сомнений в том, что, добравшись до главы 9, вы сможете самостоятельно написать псевдокод игры, план которой приведен на предыдущих страницах. Составление псевдокода даст вам ясное понимание того, как должна работать программа.

Вот что мы получили. Ваш псевдокод может отличаться, если вы выбрали другой уровень детализации, но главное, чтобы общая логика программы совпадала

1. Загрузить текст истории из файла.

Необходимо каким-то образом загрузить содержимое в Python

После этого начинаем обрабатывать текст, проверяя каждое слово

2. Обработать текст.

   Для каждого слова в тексте:

   А. Если слово — заполнитель (ГЛАГОЛ, ПРИЛАГАТЕЛЬНОЕ или СУЩЕСТВИТЕЛЬНОЕ):

Если найдено слово-заполнитель, необходимо попросить пользователя ввести слово для замены

      1. Попросить пользователя ввести такую часть речи.
      2. Подставить введенное слово вместо заполнителя.

Далее необходимо подставить это слова вместо заполнителя в тексте

   Б. В противном случае слово остается в тексте.

Если слово не является заполнителем, оставляем его

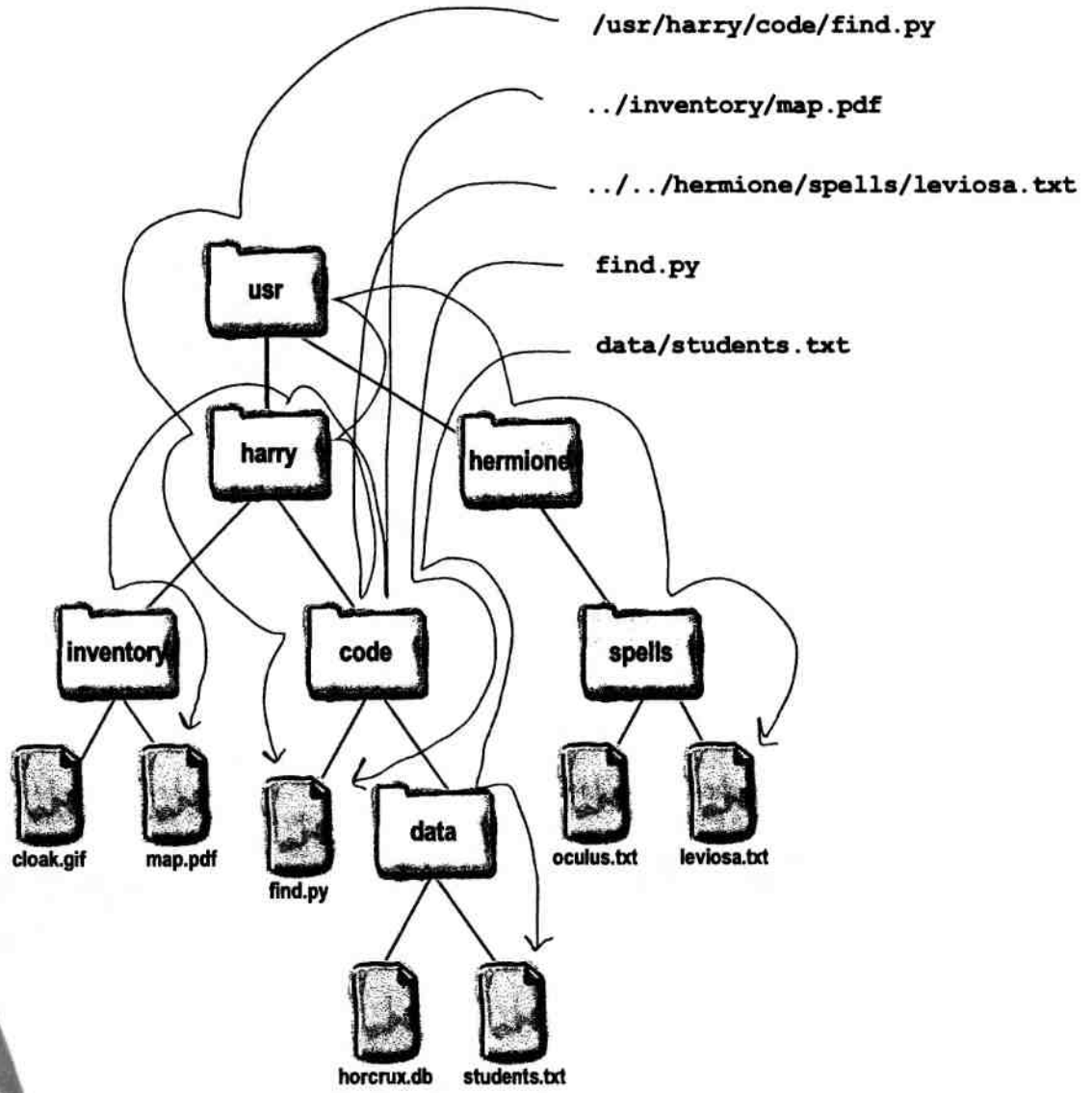
3. Сохранить результаты.

   Записать обработанный текст с подставленными заполнителями в файл, имя которого содержит префикс "crazy_".

Наконец, когда все слова обработаны, записываем текст в новый файл

Возьмите карандаш
Решение

Отследите, какому файлу соответствует каждый из приведенных ниже путей. В случае относительного пути начинайте с каталога code. Первое задание мы выполнили за вас.



Код на магнитиках: решение

Не поможете найти иголку в стоге сена? В рабочем каталоге находятся 1000 текстовых файлов с последовательными именами от *0.txt* до *999.txt*, и только в одном из них содержится слово `'needle'` (иголка). На магнитиках, прикрепленных к дверце холодильника, был записан код поиска такого файла, но вот незадача: кто-то случайно все перемешал. Сумеете восстановить код в прежнем виде? Учтите, что некоторые магнитики могут оказаться лишними.

Расположите магнитики в нужном порядке

```
for i in range(0, 1000):
    filename = str(i) + '.txt'
    file = open(filename, 'r')
    text = file.read()
    if 'need e' in text:
        print('Нашел иголку в файле ' + str(i) + '.txt')
    file.close()
```

Проверяем 1000 файлов, формируя имена вида 0.txt, 1.txt, 2.txt...

Открываем файл для чтения

Считываем весь файл, получая строку, которая записывается в переменную text

Проверяем, содержится ли в строке слово "needle" (иголка)

Если слово найдено, сообщаем имя файла

И не забудьте закрыть каждый файл

Убедитесь сами, проведя поиск в подкаталоге ch9/needle, где содержатся файлы с именами от 0.txt до 999.txt. В каком из них находится слово "needle"?

Возьмите карандаш
Решение

Почему программа вдруг пропустила несколько существительных? В чем проблема: в словах или логике программы? Попробуйте выяснить это, сопоставив входные и выходные данные, и запишите свои соображения на этот счет.

Похоже, пропущено первое существительное и последнее

```
Первое, что необходимо сделать, чтобы
ГЛАГОЛ писать собственный СУЩЕСТВИТЕЛЬНОЕ, -
научиться ГЛАГОЛ стоящие перед вами задачи
на ПРИЛАГАТЕЛЬНОЕ действия, которые
СУЩЕСТВИТЕЛЬНОЕ сможет выполнить за вас.
Для этого вы и компьютер должны ГЛАГОЛ
понятный друг другу СУЩЕСТВИТЕЛЬНОЕ, но
об этом мы вскоре поговорим.
```

Оболочка Python

```
Введите ГЛАГОЛ: уйти
Введите ГЛАГОЛ: конспект
Введите ПРИЛАГАТЕЛЬНОЕ: крошить
Введите СУЩЕСТВИТЕЛЬНОЕ: обезьяна
Введите ГЛАГОЛ: съесть
Первое, что необходимо сделать, чтобы
уйти писать собственный СУЩЕСТВИТЕЛЬНОЕ, -
научиться конспект стоящие перед вами задачи
на крошить действия, которые
обезьяна сможет выполнить за вас.
Для этого вы и компьютер должны съесть
понятный друг другу СУЩЕСТВИТЕЛЬНОЕ, но
об этом мы вскоре поговорим.
>>>
```

Если присмотреться к пропущенным заполнителям, то можно заметить ключевое отличие: в конце каждого из них стоит запятая

Ну надо же! Это та же самая ошибка, с которой мы столкнулись в главе 6 при анализе текста. Необходимо учитывать наличие запятой (или точки) при сравнении слова с заполнителем

Учимся понимать Python Решение

Потратьте время на выполнение небольшого упражнения. Оно позволит увидеть еще несколько распространенных ошибок, о которых часто сообщает интерпретатор Python. Возьмите на себя функции интерпретатора и попытайтесь предугадать его реакцию в процессе синтаксического анализа кода. Запишите предполагаемые результаты или сообщения об ошибках в приведенных ниже окнах.

```
list = [1, 2, 3, 4]
item = list[5]
```

Оболочка Python

```
Traceback (most recent call last):
  File "/Users/eric/code/ch8/errors/list.py", line 2, in <module>
    item = list[5]
IndexError: list index out of range
>>>
```

```
filename = 'документ' + 1 + '.txt'
```

Оболочка Python

```
Traceback (most recent call last):
  File "/Users/eric/code/ch8/errors/filename.py", line 1, in <module>
    filename = "документ" + 1 + ".txt"
TypeError: must be str, not int
>>>
```

```
int('1')
int('2')
int('E')
int('4')
int('5')
int('6')
```

Оболочка Python

```
Traceback (most recent call last):
  File "/Users/eric/code/ch8/errors/ints.py", line 3, in <module>
    int('E')
ValueError: invalid literal for int() with base 10: 'E'
>>>
```

```
msg = 'Привет!'

def hi():
    print(msg)
    msg = 'Салют!'

hi()
```

Оболочка Python

```
Traceback (most recent call last):
  File "/Users/eric/code/ch8/errors/function.py", line 7, in <module>
    hi()
  File "/Users/eric/Documents/code/ch8/errors/function.py", line 4, in hi
    print(msg)
UnboundLocalError: local variable 'msg' referenced before assignment
>>>
```

```
firstname = 'Бетховен'
print('Имя: ' + name)
```

Оболочка Python

```
Traceback (most recent call last):
  File "/Users/eric/code/ch8/errors/print.py", line 2, in <module>
    print('Имя: ' + name)
NameError: name 'name' is not defined
>>>
```

Возьмите карандаш
Решение

Трижды проследите за выполнением приведенного ниже кода. В первом случае введите любое число, не равное нулю, во втором — число 0, а в третьем — строку "ноль". Укажите, какой результат вы ожидаете получить в каждом случае.

```
try:
    num = input('Введите число: ')
    result = 42 / int(num)
except ZeroDivisionError:
    print("Нельзя делить на ноль!")
except ValueError:
    print("Простите, но нужно ввести число.")
else:
    print('Ваш ответ:', result)
finally:
    print('Спасибо, что были с нами!')
```

Введите число, отличное от 0

Оболочка Python

```
Введите число: 2
Ваш ответ: 21.0
Спасибо, что были с нами!
>>>
```

Введите 0

Оболочка Python

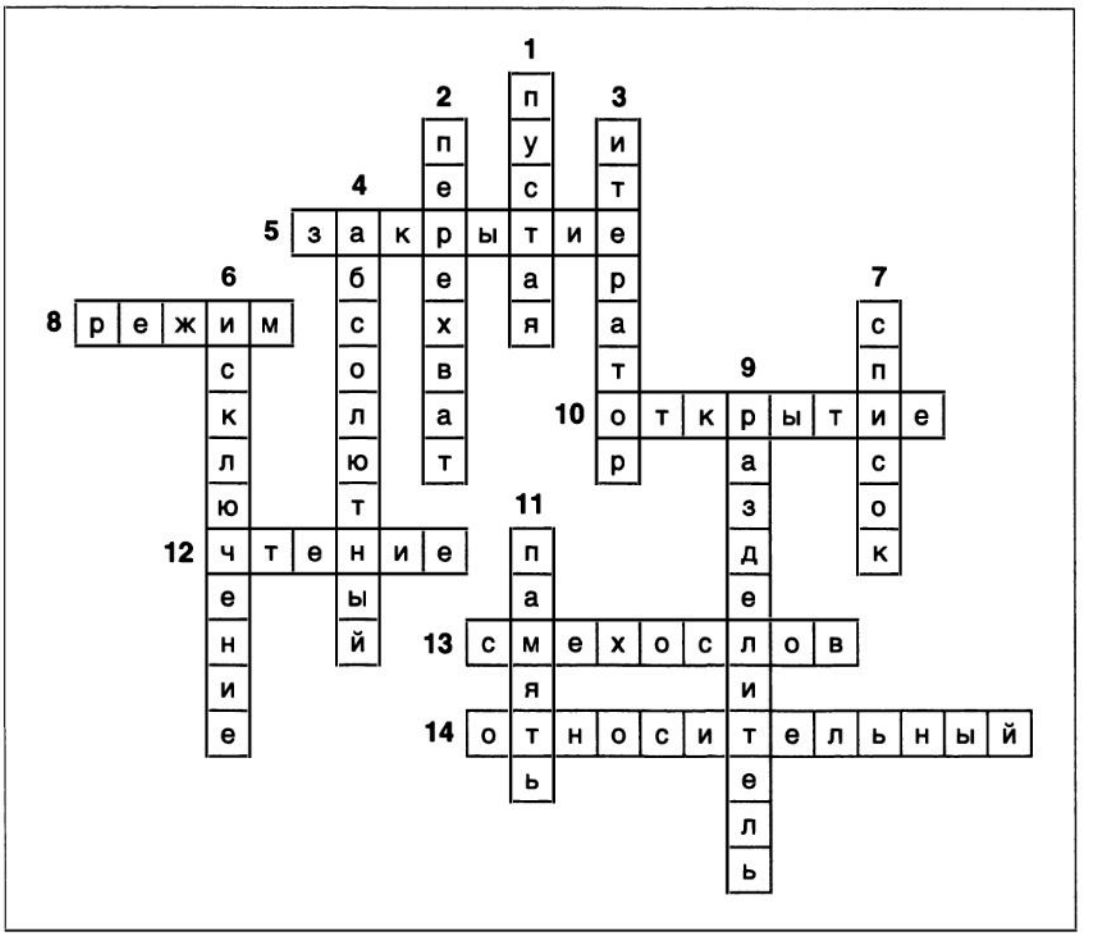
```
Введите число: 0
Нельзя делить на ноль!
Спасибо, что были с нами!
>>>
```

Введите строку "ноль"

Оболочка Python

```
Введите число: ноль
Простите, но нужно ввести число.
Спасибо, что были с нами!
>>>
```

Кроссворд: решение



10 Веб-службы

Так хочется большего

Вы научились писать неплохие программы, но всегда хочется большего. В Интернете хранится просто невероятное количество **данных**, которые так и ждут, чтобы с ними поработали. Нужен прогноз погоды? Как насчет базы данных рецептов? Или вас интересуют спортивные результаты? А может, ваш интерес — музыкальная база данных с информацией о музыкантах, альбомах и композициях? Всю эту информацию можно получить с помощью **веб-служб**. Для работы с ними нужно знать, как организуется передача данных в Интернете, а также познакомиться с парочкой модулей Python: `requests` и `json`. В этой главе вы изучите возможности веб-служб и повысите свой уровень знаний Python до заоблачных высот. Так что приготовьтесь к путешествию в космос.

И это не шутка!

Расширение возможностей благодаря веб-службам

В книге мы постоянно придерживались одного и того же шаблона: мы брали нужный нам код и абстрагировали его в виде функций. Это позволяло нам применять готовые функции, не заботясь о деталях того, как именно они реализованы. Такой подход дает возможность анализировать программу на более высоком уровне.

Далее мы научились упаковывать переменные и функции в модули. Помните модуль, который мы передали Кори Доктороу в главе 7? Он смог быстро изучить документацию к нему, увидеть, какие функции доступны, и начать применять их. Такого рода модули называются *API* (Application Programming Interface – интерфейс прикладного программирования). API можно рассматривать как набор задокументированных функций, доступных для свободного использования в программах.

В объектно-ориентированном программировании имеется еще один уровень абстракции: объекты. Подробнее об этом мы поговорим в главе 12, в которой вы научитесь создавать собственные объекты и классы

В Python существует множество готовых модулей, созданных сторонними разработчиками. Математические функции, случайные числа, графика, черепашки и многое другое – все это в значительной степени расширяет базовые возможности языка.

В этой главе мы пойдем еще дальше и узнаем, как работать с кодом, доступным в Интернете. Такой код не обязательно должен быть написан на Python, но он всегда выполняется на стороне веб-сервера, а для доступа к нему используется программный интерфейс веб-служб (Web API).

Вы спросите, какого рода веб-службы доступны в Интернете? Самые разные, начиная от погодных сводок и заканчивая музыкальными базами данных. А как насчет веб-службы, которая возвращает координаты космических объектов?

Это лишь несколько примеров веб-служб, доступных для использования в программах.

Сила мысли

Представьте, что вас только что вызвал к себе босс и попросил быстро написать приложение, отображающее текущую погодную сводку для вашего населенного пункта. Причем демоверсия нужна на завтра. Насколько упростится задача, если метеосводки будут доступны с помощью веб-службы? Можно ли написать приложение без использования веб-службы? Сколько времени, на ваш взгляд, займет написание приложения в каждом из случаев?

Как работает Веб-служба

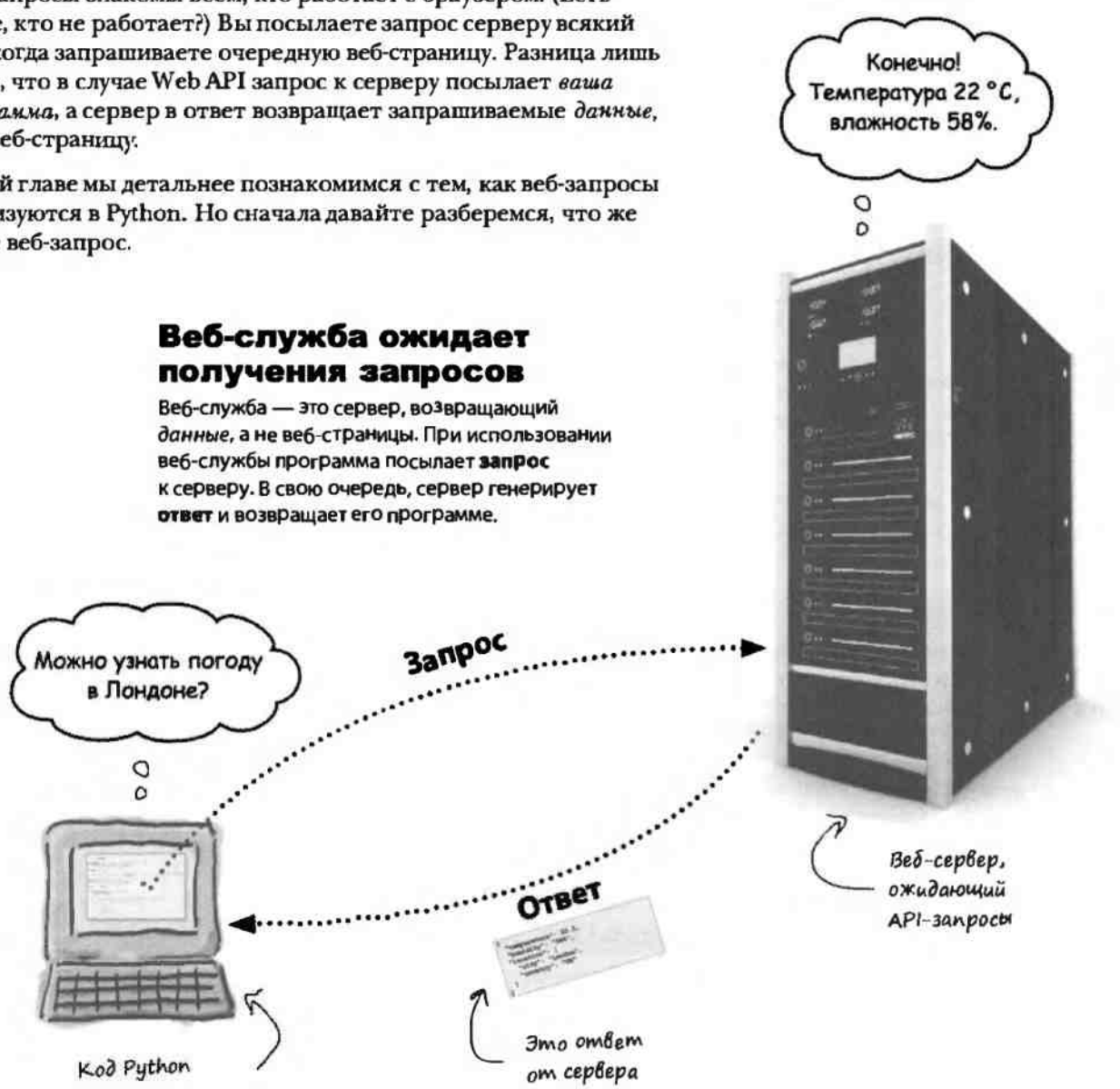
В случае веб-службы мы не вызываем функции модуля или библиотеки, а отправляем запросы к веб-серверу. Поэтому нас будут интересовать не функции, реализуемые веб-сервером, а *веб-запросы*, которые он принимает. Именно они и составляют программный интерфейс веб-сервера (Web API).

Веб-запросы знакомы всем, кто работает с браузером. (Есть такие, кто не работает?) Вы посылаете запрос серверу всякий раз, когда запрашиваете очередную веб-страницу. Разница лишь в том, что в случае Web API запрос к серверу посылает *ваша программа*, а сервер в ответ возвращает запрашиваемые *данные*, а не веб-страницу.

В этой главе мы детальнее познакомимся с тем, как веб-запросы реализуются в Python. Но сначала давайте разберемся, что же такое веб-запрос.

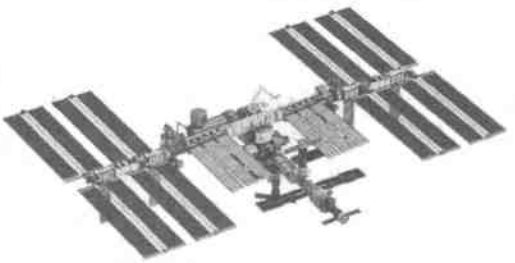
Веб-служба ожидает получения запросов

Веб-служба — это сервер, возвращающий *данные*, а не веб-страницы. При использовании веб-службы программа посылает **запрос** к серверу. В свою очередь, сервер генерирует **ответ** и возвращает его программе.



Адрес Веб-службы

Для обращения к веб-службе необходимо указать ее адрес, подобный вводимому в браузере. Он состоит из имени сервера и описания ресурса. В случае браузера ресурсом обычно служит веб-страница. Но при работе с веб-службой целевым ресурсом будут данные самого разного вида. Способ задания ресурса также различается в зависимости от веб-службы. Для наглядности рассмотрим несколько примеров.



Веб-адреса называются URL. Это аббревиатура от "Uniform Resource Locator" (унифицированный указатель ресурса)

```
http://api.open-notify.org/iss-now.json
```

Веб-адреса всегда начинаются с указания протокола браузера, в данном случае HTTP

Далее указывается имя сервера в Интернете

В конце указывается запрашиваемый ресурс на сервере. В данном случае нас интересует текущее положение Международной космической станции (МКС)

HTTP означает "Hyptertext Transfer Protocol" (протокол передачи гипертекста). См. https://ru.wikipedia.org/wiki/HTTP

А вот еще один пример, в котором с сайта организации Open Weather Map запрашивается метеорологическая сводка для Лондона.

Это URL-адрес службы прогноза погоды

Указываем имя веб-службы сервера

В данном случае описание запрашиваемого ресурса выглядит совсем по-другому

```
http://api.openweathermap.org/data/2.5/weather?q=London
```

Сначала сообщаем веб-службе, что нам нужны данные...

...и что мы хотим воспользоваться версией 2.5 API...

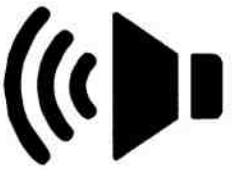
...и мы хотим узнать погоду в Лондоне

Что, если попробовать запросить данные об исполнителе в базе данных Spotify?

Здесь мы используем версию 1 серверного API

Запрашиваем данные о конкретном музыканте

Это идентификатор нужного нам музыканта



http://api.spotify.com/v1/artists/43ZHCT0cAZBISjO8DG9PnE/related-artists

Как всегда, сначала указываем имя сервера

Догадываетесь, кто этот музыкант? Подсказка: его называли "королем рок-н-ролла"

Мы хотим получить данные о связанных с ним музыкантах

Изучите документацию к API.

Мы не шутим. Это не настолько страшно, как кажется. Все веб-службы придерживаются стандартного синтаксиса URL-адресов (вы сможете узнать его самостоятельно), но предоставляют разные программные интерфейсы для разных типов данных. Поэтому способ обращения к данным в каждом случае будет разным. Достаточно взглянуть на приведенные выше примеры: запрос к серверу Open Notify очень простой, чего не скажешь о запросе к серверу Spotify.

Вы увидите, что большинство веб-служб снабжено вполне адекватной документацией, в которой детально расписаны форматы запросов для всех поддерживаемых типов данных. Нам предстоит поближе познакомиться с веб-службой Open Notify.

Не удивительно.

Отправка веб-запросов из браузера — прекрасный способ изучить интерфейс веб-службы. Но нужно учитывать, что многие службы требуют предварительной регистрации (чаще всего бесплатной). В процессе регистрации вы получаете маркер авторизации или ключ доступа, который передается серверу вместе с запросом. При отсутствии ключа многие службы возвращают ошибку.

Вот почему необходимо предварительно изучить документацию к веб-службе, прежде чем отправлять ей запросы.

В приведенных выше примерах в ключах доступа нуждаются службы Spotify и Open Weather. А вот служба Open Notify в данный момент работает без маркеров авторизации.

УПРАЖНЕНИЕ

Введите следующий URL-адрес в браузере. Что вы получите?

`http://api.open-notify.org/iss-now.json`

Попробуйте ввести запрос несколько раз. Можете также задать широту и долготу в Google Maps, например `"-0.2609, 118.8982"` *(первой указывается широта)*

Небольшое обновление Python

Прежде чем приступать к составлению запросов к веб-службам, нужно добавить в Python *новый пакет*. Что такое "пакет"? Как упоминалось в главе 7, пакет представляет собой набор взаимосвязанных модулей. Его также называют *библиотекой*.

Для добавления пакета мы воспользуемся утилитой `pip`, название которой представляет собой акроним от "Pip Installs Packages". С ее помощью в системе устанавливаются и удаляются различные пакеты.

Это рекурсивный акроним!

Нам понадобится пакет `requests`, позволяющий отправлять запросы к веб-службам. Из всего пакета нам нужен одноименный модуль `requests` (трудно запутаться). Следует отметить, что в Python имеется встроенный модуль создания веб-запросов, но большинство программистов считают модуль `requests` более функциональным и простым. Кроме того, благодаря ему мы получим опыт работы с утилитой `pip`.

Куда можно слать API-запросы? Да куда угодно: Twitter, Spotify, Microsoft, Amazon, BuzzFeed, Reddit и др.

Давайте установим пакет `requests` и познакомимся с его возможностями.

Возможности пакета и модуля `requests` описаны по адресу `http://docs.python-requests.org`

Пакеты, написанные другими разработчиками и организациями (не входящими в рабочую группу Python), называются сторонними

Requests: HTTP for Humans

Release v2.18.1. (Installation)

Requests is the only Non-GMO HTTP library for Python, safe for human consumption.

Warning: Recreational use of the Python standard library for HTTP may result in dangerous side-effects, including: security vulnerabilities, verbose code, reinventing the wheel, constantly reading documentation, depression, headaches, or even death.

Behold, the power of Requests:

```
>>> r = requests.get('https://api.github.com/user', auth=('user', 'pass'))
>>> r.status_code
200
>>> r.headers['content-type']
'application/json; charset=utf8'
>>> r.encoding
'utf-8'
>>> r.text
u'{"type":"User",...'
>>> r.json()
{u'private_gists': 419, u'total_private_repos': 77, ...}
```

See similar code, sans Requests.

Requests allows you to send organic, grass-fed HTTP/1.1 requests, without the need for manual labor. There's no need to manually add query strings to your URLs, or to form-encode your POST data. Keep-alive and HTTP connection pooling are 100% automatic, thanks to urllib3.

User Testimonials

Twitter, Spotify, Microsoft, Amazon, Lyft, BuzzFeed, Reddit, The NSA, Her Majesty's Government, Google, Twilio, Runscope, Mozilla, Heroku, PayPal, NPR, Obama for America, Transifex, Native Instruments, The Washington Post, SoundCloud, Kippt, Sony, and Federal U.S. Institutions that prefer to be unnamed claim to use Requests internally.

Stay Informed

Установка обновления

Утилита `pip` встроена в Python, поэтому ее можно вызвать из командной строки для установки пакета `requests`.

Используйте командную строку операционной системы, как это делалось в главе 9

Консоль

```
$ python3 -m pip install requests
```

Пользователи Windows должны набирать "python", а не "python3"

Введите показанную выше команду, чтобы начать установку. ***Необходимо иметь подключение к Интернету, чтобы утилита `pip` смогла скачать пакеты. Учтите, что установка выполняется в каталог библиотеки Python, а не в ваш текущий каталог. Если возникнут ошибки доступа, проверьте, есть ли у вас полномочия для установки пакетов***

Консоль

```
$ python3 -m pip install requests
Collecting requests
  Downloading requests-2.18.1-py2.py3-none-any.whl (88kB)
    100% |                                | 92kB 1.4MB/s
Collecting id a<2.6,>=2.5 (from requests)
  Downloading idna-2.5-py2.py3-none-any.whl (55kB)
    100% |                                | 61kB 2.5MB/s
Collecting urllib3<1.22,>=1.21.1 (from requests)
  Downloading urllib3-1.21.1-py2.py3-no e-a y.whl (131kB)
    100% |                                | 133kB 2.5MB/s
Collecting certifi>=2017.4.17 (from requests)
  Downloading certifi-2017.4.17-py2.py3-none-any.whl (375kB)
    100% |                                | 378kB 2.2MB/s
Collecting chardet<3.1.0,>=3.0.2 (from requests)
  Downloading chardet-3.0.4-py2.py3-none-any.whl (133kB)
    100% |                                | 143kB 4.1MB/s
Installing collected packages: id a, urllib3, certifi, chardet, requests
$
```

Утилита `pip` скачивает модуль `requests`, а также все зависимые пакеты. В данном случае установка ведется в macOS, поэтому в вашей системе окно будет другим

Не бойтесь задавать вопросы

В: Чем модуль `requests` лучше встроенных инструментов Python? Неужели встроенного модуля недостаточно для работы с веб-запросами?

О: Разработчики пакета `requests` создали библиотеку для работы с веб-запросами, которая намного проще и функциональнее, чем встроенный модуль Python, причем настолько, что именно она применяется в большинстве веб-служб. Впрочем, никто не запрещает вам использовать встроенный модуль, если возникает такая необходимость (он называется `urllib` или `urllib2`).

Что хорошо в возможности расширения Python: разработчики создают собственные пакеты и делятся ими со всеми желающими.

В: Это хорошо, что в Python можно добавлять новые пакеты, но где их искать?

О: Новые пакеты можно искать непосредственно в командной строке, например:

```
python3 -m pip search hue
```

В данном случае мы ищем пакеты по ключевому слову `'hue'`. Другой вариант — выполнить поиск в Google, например "python3 requests module" или "python3 hue lighting". Также проверьте репозиторий программного обеспечения для Python по адресу `http://pypi.org`.

В: Я работаю в Python 3, но не могу найти утилиту `pip`.

О: Менеджер пакетов `pip` был добавлен в Python версии 3.4. Проверьте, какая у вас версия, и обновитесь до самой последней версии, которую можно найти на сайте `www.python.org`.

Нам нужна интересная Веб-служба

После установки пакета `requests` можно приступать к составлению первого веб-запроса на Python. Но сначала нужно найти интересную веб-службу, к которой этот запрос можно будет адресовать. Как вы помните, в начале главы мы анонсировали путешествие в космос, а для этого мы воспользуемся веб-службой Open Notify, предоставляющей текущие координаты Международной космической станции (МКС). Соответствующий Web API доступен на сайте `open-notify.org`. Рассмотрим его возможности.

Документация к Open Notify доступна на сайте `open-notify.org`. Здесь мы рассмотрим только самое основное

Open Notify предоставляет три программных интерфейса

Возможности Open Notify

Как сообщается на начальной странице сайта `open-notify.org`, в вашем распоряжении имеются сразу три программных интерфейса: для определения текущих координат МКС, вычисления времени ее прохождения над текущей точкой и определения количества людей, находящихся в данный момент в космосе. Нас интересует первый из них, поэтому щелкните на ссылке "Current Location of..."

Это документация к веб-службе, возвращающей текущие координаты МКС

Вы, вероятно, уже поняли, что эта веб-служба возвращает текущую широту и долготу Международной космической станции (МКС)

Open Notify

International Space Station Current Location

Overview

JSON

Вот URL-адрес запроса

http://api.open-notify.org/iss-now.json

Вот так формируется URL-адрес запроса на получение текущего положения МКС

```
{
  "message": "success",
  "timestamp": UNIX_TIME_STAMP,
  "iss_position": {
    "latitude": CURRENT_LATITUDE,
    "longitude": CURRENT_LONGITUDE
  }
}
```

А это пример ответа от веб-службы

Напоминает пары "ключ/значение". Что-то знакомое, правда?

Нам нужен ключ `iss_position`, в котором хранится еще один набор пар "ключ/значение" с координатами МКС

The `data` payload has a `timestamp` and an `iss_position` object with the latitude and longitude.

Examples

Data Source

Веб-службы Open Notify написаны и сопровождаются Нейтаном Берджи, инженером из Портленда, штат Орегон

Сила мысли

Формат данных, возвращаемых веб-службой Open Notify, должен быть вам знаком. Какой тип данных Python он напоминает?

Произносится как "Джейсон" с ударением на "о"

Передача данных в формате JSON

JSON – это формат, в котором веб-службы возвращают данные пользователям. Он наверняка знаком вам, поскольку синтаксически схож со словарями Python.

Существуют и другие форматы обмена данными в Интернете, но JSON – самый распространенный

Формат JSON можно рассматривать как набор пар "ключ/значение", представленных в текстовом виде. Предположим, к примеру, что вы запрашиваете на сервере погоды текущую метеосводку по Лондону. Соответствующая веб-служба сформирует ответ в формате JSON, который будет выглядеть примерно так.

Каждая пара состоит из строки, которая служит ключом, и значения

В формате JSON пары "ключ/значение" заключаются в фигурные скобки

```
{
  "temperature": 22.0,
  "humidity": "58%",
  "location": {
    "city": "London",
    "country": "UK"
  }
}
```

Значения могут быть как строками, так и числами

В качестве значений допускается использовать вложенные пары "ключ/значение"

Все ключи – это строки JSON

В формате JSON можно также задавать списки, которые здесь называются массивами

Веб-служба вставляет это определение в текстовую строку и возвращает ее в качестве ответа

Получение ответа в формате JSON

Не бойтесь задавать вопросы

В: Получается, что JSON и словари Python — это одно и то же?

О: Эти структуры данных настолько похожи, что их легко перепутать. Но между ними есть и принципиальная разница. Повстречав код объявления словаря, интерпретатор Python преобразует его во внутреннюю структуру данных, с которой далее работает программа. В случае JSON пары "ключ/значение" пересылаются по сети в текстовом виде. Так что JSON и словарь Python — не одно и то же. JSON — универсальный формат, предназначенный для чтения и интерпретации любым языком.

В: Если данные JSON поступают в текстовом формате, то как их обрабатывать в программе?

О: Мы как раз собираемся рассмотреть этот вопрос.

В: Почему формат называется JSON?

О: Несмотря на то что JSON разрабатывался как формат, независимый от языка программирования, он произошел от JavaScript, а его название расшифровывается как "JavaScript Object Notation" (объектная нотация JavaScript).

В: И все же мне непонятно: как узнать, какие есть веб-службы и как с ними работать?

О: Сведения о доступных веб-службах можно найти на различных сайтах, таких как `www.programmableweb.com`. Большинство популярных служб снабжается документацией, например `dev.twitter.com` и `developer.spotify.com`.

Возьмите карандаш

Преобразуйте данные JSON, приведенные на предыдущей странице, в словарь Python и завершите приведенный ниже фрагмент программы, заполнив прочерки. В качестве подсказки сверяйтесь с результатами работы программы.

```
current = ____________________________________
          ____________________________________
          ____________________________________
          ____________________________________
          ____________________________________
          ____________________________________
          ____________________________________

loc = ________________________
print('Город:',
      loc['city'] + ', ' + ____________ +
      '. Температура:',
      ________________, '°C.')
```

В будущем программа сама будет запрашивать эти данные

Оболочка Python

```
Город: London, UK. Температура: 22.0 °C.
>>>
```

Вернемся к модулю requests

Для создания веб-запроса мы воспользуемся функцией `get()` модуля `requests`.

1 Вызовите функцию get() для создания запроса.

Сначала мы вызываем функцию `get()` модуля `requests` для создания запроса к удаленному веб-серверу.

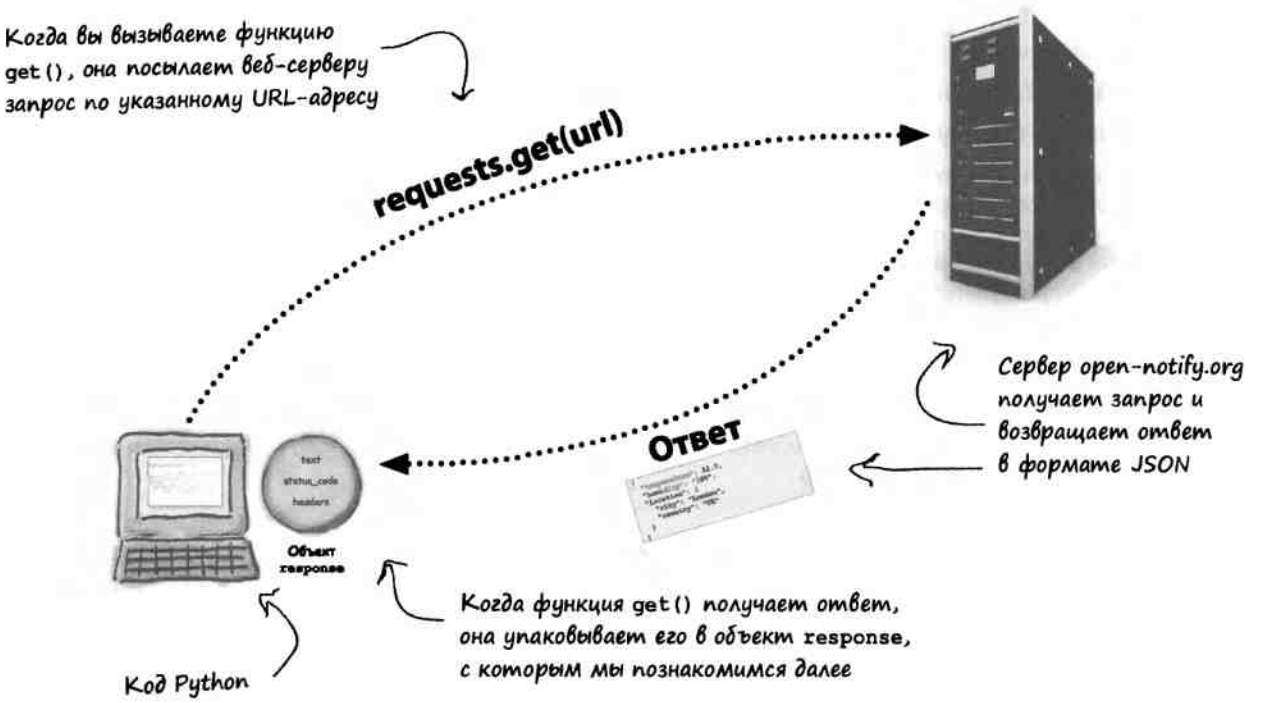
```
url = 'http://api.open-notify.org/iss-now.json'
```

Это URL-адрес, который мы взяли в документации Open Notify

```
response = requests.get(url)
```

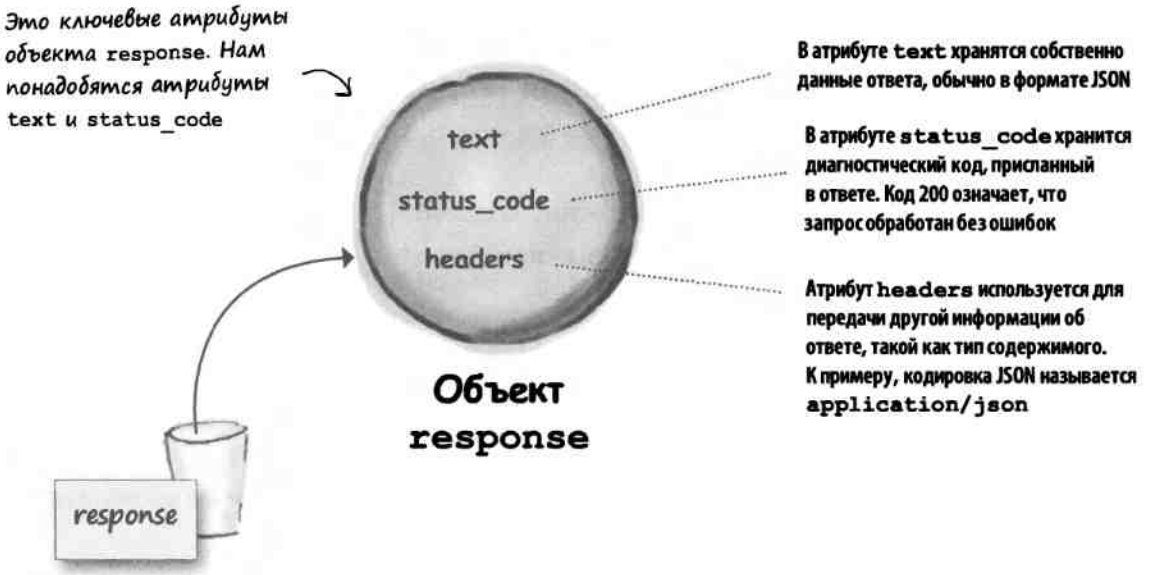
Передаем URL-адрес в функцию `get()`, после чего функция посылает запрос удаленному серверу

Ответ, полученный от удаленного сервера, упаковывается в удобный объект `response`



2 Изучите объект response.

Как вы уже знаете, получив ответ от сервера, функция `get()` представляет его в виде объекта `response`. Давайте познакомимся с ним поближе.



3 Проверьте атрибут status_code и получите данные атрибута text.

После получения ответа от сервера первым делом нужно проанализировать код состояния. Код 200 указывает на то, что запрос обработан без ошибок. Конечно, может быть получен и другой код состояния, например код ошибки. В случае кода 200 можно приступать к извлечению данных, переданных веб-службой, — они содержатся в атрибуте `text`. Ниже показано, как это сделать. Внимательно изучите данный фрагмент, а программа целиком приведена на следующей странице.

Используем объект response, полученный в п. 1

Убедимся, что код состояния равен 200 (запрос выполнен успешно)

```
if (response.status_code == 200):
    print(response.text)
else:
    print("Хьюстон, у нас проблема:", response.status_code)
```

Если код состояния равен 200, значит, в атрибуте text что-то содержится

В противном случае произошла какая-то ошибка, и нужно сообщить об этом пользователю. Полный список кодов состояния доступен по адресу https://ru.wikipedia.org/wiki/Список_кодов_состояния_HTTP

Собираем все вместе: создание запроса к службе Open Notify

Мы располагаем всем необходимым для создания запроса, поэтому не будем мешкать. Вы уже знаете, что для отправки веб-запроса вызывается функция `get()`, которой передается целевой URL, а благодаря сайту `open-notify.org` мы знаем, как должен выглядеть этот URL. Нам также известно, что после получения ответа нужно проверить значение атрибута `status_code` объекта `response`. Если оно равно 200, значит, можно прочитать значение атрибута `text`.

Итак, давайте создадим запрос к веб-службе Open Notify.

Сначала импортируем модуль `requests`

Запишем в переменную `url` *URL-адрес веб-службы*

```
import requests

url = 'http://api.open-notify.org/iss-now.json'

response = requests.get(url)

if (response.status_code == 200):
    print(response.text)
else:
    print("Хьюстон, у нас проблема:", response.status_code)
```

Это объект `response`

Используем функцию `get()`*, передав ей URL, по которому мы хотим получить данные*

Проверим, все ли в порядке (получен ли код состояния 200)

Если код состояния не равен 200, значит, возникла какая-то ошибка, поэтому выводим ее код. Полный список кодов состояния доступен по адресу https://ru.wikipedia.org/wiki/Список_кодов_состояния_HTTP

Выводим текст ответа. Помните: в атрибуте `text` *объекта* `response` *хранятся данные, полученные от веб-службы*

Проверьте доступность, прежде чем продолжать

На момент написания книги служба ISS Current Location была полностью работоспособна, но на будущее никто гарантий не дает. Если по какой-то причине служба перестанет функционировать, у нас есть запасной план: посетите сайт `http://wickedlysmart.com/hflearntocode`, чтобы узнать, нужно ли будет что-то менять в описанном далее коде. Скорее всего, ничего делать не придется, но лучше перестраховаться.

Тестовый полет

Сохраните приведенный выше код в файле *iss.py* и выполните команду **Run > Run Module**, чтобы проверить работу программы.

Внимание!

Возникли проблемы?

Если тест-драйв закончился неудачей, в первую очередь проверьте код состояния. Любое значение, отличающееся от 200, означает ошибку (узнайте в Интернете, что она означает). Не можете определить код состояния? Вставьте URL в адресную строку браузера, чтобы проверить доступность веб-службы. Далее проверьте программу на наличие синтаксических ошибок и убедитесь в наличии пакета `requests`. Убедитесь, что в оболочке не выводятся сообщения об исключениях. Если возникает подозрение в неполадках со службой Open Notify, прочитайте врезку "Внимание!" на предыдущей странице.

Вот что мы получили

Для этого тест-драйва требуется подключение к Интернету

Оболочка Python

```
{"iss_position": {"longitude": "-146.2862", "latitude":
"-51.0667"}, "message": "success", "timestamp": 1507904011}
>>>
```

Есть идеи, что это за место?

Работа с данными JSON в Python

Итак, мы получили ответ на запрос о местонахождении МКС, представленный в формате JSON, но пока что это всего лишь *текстовая строка*, от которой мало пользы. Ее можно разве что вывести на экран. Для работы с такими данными нам понадобится модуль `json`. Он содержит удобную функцию `load()`, которая преобразует строку JSON в словарь Python. Рассмотрим пример.

Сначала необходимо импортировать модуль `json`

Строка, содержащая данные в формате JSON. Учтите, что для Python это обычная текстовая строка

```
import json

json_string = '{"first": "Emmett", "last": "Brown", "prefix": "Dr."}'

name = json.loads(json_string)

print(name['prefix'], name['first'], name['last'])
```

Применяем к строке функцию `json.loads()` *и заносим полученный словарь в переменную* `name`

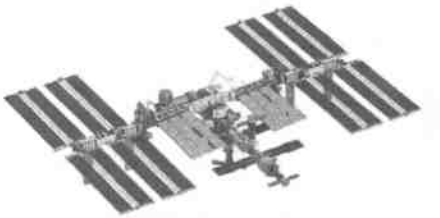
Используем словарь Python для доступа к атрибутам `prefix`, `first` *и* `last`

Напоминание: если собираетесь тестировать этот код, не называйте файл json.py (так называется файл модуля `json`*). Данный вопрос обсуждался в главе 7*

Оболочка Python

```
Dr. Emmett Brown
>>>
```

Применение модуля json



Давайте теперь воспользуемся модулем `json` для получения координат Международной космической станции. Сделать это достаточно просто. У нас уже есть строка JSON, полученная от сервера Open Notify и сохраненная в объекте `response`. Нужно лишь использовать функцию `load()` модуля `json`, чтобы преобразовать строку в словарь Python.

Дополнительно импортируем модуль `json`. Имена модулей разделяются запятыми, чтобы их можно было указывать в одной строке

```
import requests, json

url = 'http://api.open-notify.org/iss-now.json'

response = requests.get(url)

if (response.status_code == 200):
    response_dictionary = json.loads(response.text)
    ~~print(response.text)~~
    position = response_dictionary['iss_position']
    print('Координаты МКС: ' +
        position['latitude'] + ', ' + position['longitude'])
else:
    print("Хьюстон, у нас проблема:", response.status_code)
```

Используем функцию `json.loads()`, чтобы получить ответ в формате строки JSON и преобразовать его в словарь Python

Используем ключ `iss_position`, который содержит другой словарь

Выводим значения широты и долготы из словаря `position`

Тестовый полет

Внесите изменения в файл *iss.py* и протестируйте полученный результат.

Вот что мы получили. Неплохо!

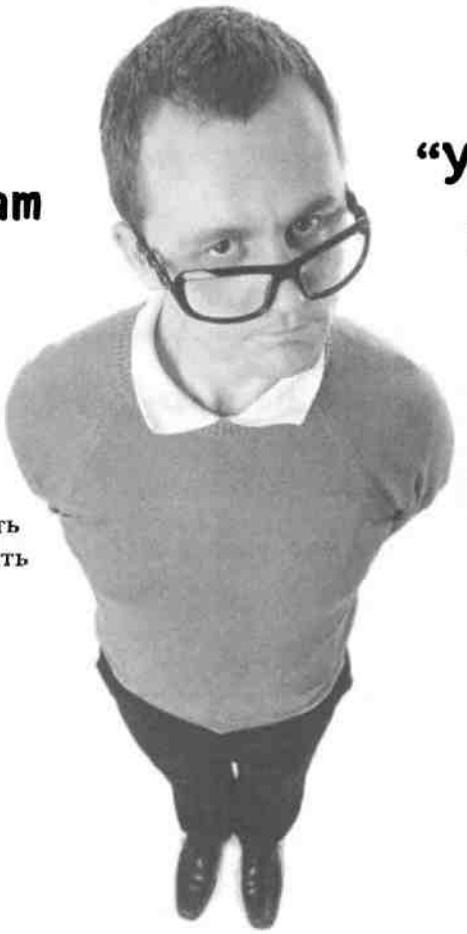
```
Оболочка Python
Координаты МКС: -51.5770, -108.2028
>>>
```

Что насчет этих координат? Знаете, где это?

Маловероятно, чтобы МКС находилась в том же месте, когда вы запустите свою программу, так что вы должны получить другие координаты

Представим результат в графическом виде

Отображение координат МКС в текстовом виде — не лучшее решение. В конце концов, это уже глава 10, хочется отобразить положение МКС на карте или что-то в таком духе. К тому же у нас есть готовая технология — пора вспомнить про черепашек и заставить их побегать вокруг земного шара. *Это не шутка!*



"Увижу еще одно консольное приложение Python — уволю!"

Знакомство с объектом screen

Нам снова на помощь придут черепашки. На этом раз мы используем их для более детального изучения возможностей объекта `screen`, с которым мы впервые столкнулись в главе 7. Этот объект позволяет настраивать свойства окна, в котором существуют черепашки.

По отношению к данному объекту лучше подходит термин "окно", а не "экран", но разработчики не спрашивали нашего мнения

Давайте импортируем модуль `turtle` и поработаем с объектом `screen`. Внесите соответствующие изменения в файл *iss.py*.

Добавляем модуль turtle

```
import requests, json, turtle

screen = turtle.Screen()
screen.setup(1000, 500)
screen.bgpic('earth.gif')
screen.setworldcoordinates(-180, -90, 180, 90)

url = 'http://api.open-notify.org/iss-now.json'

response = requests.get(url)

if (response.status_code == 200):
    response_dictionary = json.loads(response.text)
    position = response_dictionary['iss_position']
    print('Координаты МКС: ' +
          position['latitude'] + ', ' + position['longitude'])
else:
    print("Хьюстон, у нас проблема:", response.status_code)

turtle.mainloop()
```

Сначала получаем ссылку на объект screen модуля turtle

Увеличим размер окна до 1000x500 пикселей, что соответствует размеру добавляемого изображения

Изображение занимает весь фон окна

Файл earth.gif содержится в папке ch10 каталога примеров

Сбрасываем систему координат окна, о чем будет рассказано на следующей странице

Стандартная процедура запуска окна, как в главе 7

Этот файл можно найти в папке ch10 каталога примеров; скопируйте его в свою собственную папку

earth.gif

Координаты черепашек и географические координаты

Необходимо поближе познакомиться с методом `setworldcoordinates()`, который играет ключевую роль в определении положения МКС на нашей графической карте.

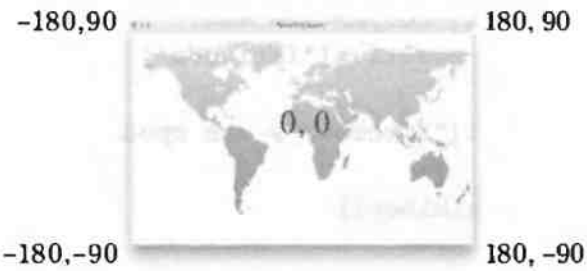
Как вы уже знаете, черепашки существуют на сетке, центр которой совмещен с началом координат (0, 0). Таким образом, у сетки размером 1000×500 координаты изменяются в диапазоне от (–500, –250) в левом нижнем углу до (500, 250) в правом верхнем углу.

Сетка географических координат состоит из *параллелей* (линий, проходящих с запада на восток) и *меридианов* (линий, проходящих с севера на юг). На координатной сетке меридианы (*долгота*) рисуются от точки –180 до точки 180 градусов (через точку 0 проходит *нулевой меридиан*), а параллели (*широта*) — от точки 90 градусов (северный полюс) до точки –90 градусов (южный полюс). Нулевая параллель называется *экватором*.

Можно легко совместить обе системы координат, воспользовавшись следующей инструкцией:

screen.setworldcoordinates(–180, –90, 180, 90)

В результате левый нижний угол окна получит координаты (–180, –90), а правый верхний угол — координаты (180, 90). Форма и размер окна останутся прежними, зато теперь перемещения черепашек будут происходить согласно географическим координатам.



Это означает, что для позиционирования черепашек на экране можно использовать привычные широту и долготу

С помощью метода `setworldcoordinates()` координатная сетка черепашек накладывается на систему координат Земли

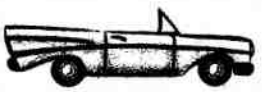
Черепашка-космонавт

Еще не догадались, как мы собираемся использовать черепашек для отслеживания положения МКС на карте? Сейчас вы все поймете. Создайте черепашку и внесите показанные ниже изменения в файл *iss.py*.

```
import requests, json, turtle

screen = turtle.Screen()
screen.setup(1000, 500)
screen.bgpic('earth.gif')
screen.setworldcoordinates(-180, -90, 180, 90)

iss = turtle.Turtle()
iss.shape('circle')
iss.color('red')

url = 'http://api.open-notify.org/iss-now.json'

response = requests.get(url)

if (response.status_code == 200):
    response_dictionary = json.loads(response.text)
    position = response_dictionary['iss_position']
    print('Координаты МКС: ' +
        position['latitude'] + ', ' + position['longitude'])
else:
    print("Хьюстон, у нас проблема:", response.status_code)

turtle.mainloop()
```

Создаем новый объект turtle, делая его форму круглой, а цвет — красным

На карте этот кружок будет (временно) представлять положение МКС над Землей



Тестовый полет

Прежде чем продолжить, протестируйте имеющийся код. Внесите все необходимые изменения и убедитесь в том, что файл *earth.gif* находится в каталоге программы (вы найдете его в папке *ch10* примеров книги). Запустите программу и проверьте результат.

И это все?

Если увидите такое изображение, значит, все в порядке. На данный момент мы всего лишь поменяли разрешение окна, задали фоновое изображение, изменили форму и цвет черепашки и поменяли систему координат

Вы должны увидеть красный кружок, соответствующий черепашке. По умолчанию он находится в точке (0, 0), только теперь это широта и долгота

Черепашка может выглядеть как космическая станция

Хоть это и не обязательный шаг для приложения, получающего данные от веб-службы, но придание черепашке сходства с космической станцией добавит программе реалистичности. К тому же так просто забавнее. Вот как это сделать.

```
import requests, json, turtle

screen = turtle.Screen()
screen.setup(1000, 500)
screen.bgpic('earth.gif')
screen.setworldcoordinates(-180, -90, 180, 90)

iss = turtle.Turtle()
turtle.register_shape("iss.gif")
iss.shape("iss.gif")
iss.shape('circle')
iss.color('red')

url = 'http://api.open-notify.org/iss-now.json'

response = requests.get(url)

if (response.status_code == 200):
    response_dictionary = json.loads(response.text)
    position = response_dictionary['iss_position']
    print('Координаты МКС: ' +
        position['latitude'] + ', ' + position['longitude'])
else:
    print("Хьюстон, у нас проблема:", response.status_code)

turtle.mainloop()
```

Файл iss.gif тоже содержится в папке ch10 каталога примеров

Необходимость регистрировать форму является особенностью модуля turtle, которую следует принять как факт. Такое требуется далеко не во всех графических библиотеках

Сообщаем модулю turtle о том, что в качестве формы используется изображение

Формой черепашки становится файл iss.gif

Тестовый полет

Как и ранее, внесите указанные изменения в код и убедитесь в том, что файл *earth.gif* находится в каталоге программы (вы найдете его в папке *ch10* примеров книги).

Отлично, теперь у нас есть изображение МКС!

Забудьте об МКС. Где находимся мы?

Конец уже близок! У нас есть все необходимое, включая координаты МКС, получаемые от службы Open Notify. Осталось только собрать все воедино. Для начала сохраним долготу и широту в отдельных переменных, чтобы можно было перемещать черепашку по экрану. Вы ведь помните, что передаваемые веб-службой данные представлены в текстовом формате JSON, а не как числовые значения? Необходимо преобразовать строки в числа.

Это лишь фрагмент кода, в котором проверяется ответ от сервера и определяются широта и долгота

```
if (response.status_code == 200):
    response_dictionary = json.loads(response.text)
    position = response_dictionary['iss_position']
    print('Координаты МКС: ' +
        position['latitude'] + ', ' + position['longitude'])
    lat = float(position['latitude'])
    long = float(position['longitude'])
else:
    print("Хьюстон, у нас проблема:", response.status_code)
```

Получаем широту и долготу из словаря `position`. *Поскольку оба значения являются строками, приводим их к типу* `float`

Вводить этот код пока что не нужно, так как вскоре мы введем всю программу целиком

Следующим этапом будет перемещение МКС по экрану. Вы уже умеете управлять черепашками — справитесь и с МКС. Все, что нам нужно, — простая функция `move_iss()`, которая перемещает черепашку в точку с заданными координатами. Перед перемещением следует поднять перо черепашки, чтобы она не оставляла след на экране.

В функцию `move_iss()` *передаются широта и долгота*

```
def move_iss(lat, long):
    global iss

    iss.penup()
    iss.goto(long, lat)
    iss.pendown()
```

Если не поднять перо, то на экране будет нарисована линия

Перемещаемся в точку с заданными координатами; сначала указывается долгота, так как это координата x

Опускаем перо обратно, делая черепашку видимой

С технической точки зрения мы не обязаны опускать перо обратно, но это правило хорошего тона, ведь по умолчанию перо опущено. Другой разработчик может быть сбит с толку, если не ожидает, что функция меняет свойства пера

Завершение программы

Добавим в программу функцию `move_iss()`, вставим инструкции для извлечения широты и долготы из словаря `position` и, наконец, вызовем функцию `move_iss()`, передав ей полученные координаты.

```
import requests, json, turtle

def move_iss(lat, long):
    global iss

    iss.penup()
    iss.goto(long, lat)
    iss.pendown()

screen = turtle.Screen()
screen.setup(1000, 500)
screen.bgpic('earth.gif')
screen.setworldcoordinates(-180, -90, 180, 90)

iss = turtle.Turtle()
turtle.register_shape("iss.gif")
iss.shape("iss.gif")

url = 'http://api.open-notify.org/iss-now.json'

response = requests.get(url)

if (response.status_code == 200):
    response_dictionary = json.loads(response.text)
    position = response_dictionary['iss_position']
    print('Координаты МКС: ' +
        position['latitude'] + ', ' + position['longitude'])
    lat = float(position['latitude'])
    long = float(position['longitude'])
    move_iss(lat, long)
else:
    print("Хьюстон, у нас проблема:", response.status_code)

turtle.mainloop()
```

Добавляем функцию `move_iss()` *в начало файла*

Нам больше не нужна функция `print()` *для вывода координат МКС: теперь они представлены графически!*

Преобразуем строки `latitude` *и* `longitude` *в значения типа* `float` *и передаем их в функцию* `move_iss()`

Тестовый полет

Замечательно, приложение готово к использованию. Теперь вы должны увидеть положение МКС на карте.

МКС движется со скоростью 7,6 км/с

МКС облетает Землю каждые 92 минуты, поэтому запустите программу несколько раз, чтобы увидеть, как меняется положение МКС

Поздравляем! Миссия выполнена!

Тренируем мозг

Просто невероятно: с помощью столь простого кода нам удалось определить положение МКС и отобразить его на карте мира. Чтобы сделать приложение еще более впечатляющим, нужно модифицировать программу так, чтобы положение отслеживалось *постоянно*, а не *разово*. Этим мы сейчас и займемся, но сначала немного подчистим имеющийся код, упаковав его в удобные функции. Зря, что ли, мы их так долго изучали? Вот переработанный вариант программы, который стал намного более понятным.

```
import requests, json, turtle

iss = turtle.Turtle()

def setup(window):
    global iss

    window.setup(1000, 500)
    window.bgpic('earth.gif')
    window.setworldcoordinates(-180, -90, 180, 90)
    turtle.register_shape("iss.gif")
    iss.shape("iss.gif")

def move_iss(lat, long):
    global iss

    iss.penup()
    iss.goto(long, lat)
    iss.pendown()


def track_iss():
    url = 'http://api.open-notify.org/iss-now.json'
    response = requests.get(url)
    if (response.status_code == 200):
        response_dictionary = json.loads(response.text)
        position = response_dictionary['iss_position']
        lat = float(position['latitude'])
        long = float(position['longitude'])
        move_iss(lat, long)
    else:
        print("Хьюстон, у нас проблема:", response.status_code)


def main():
    global iss
    screen = turtle.Screen()
    setup(screen)
    track_iss()

if __name__ == "__main__":
    main()
    turtle.mainloop()
```

Включим весь код настройки окна и черепашки в функцию `setup()`

Здесь имя параметра указано как `window`*, тогда как в действительности в функцию передается объект* `screen`*. Возможно, это не лучшее имя для параметра, и нам стоило бы добавить в код комментарий по этому поводу*

В функцию `move_iss()` *не было внесено никаких изменений*

Мы вставили в функцию `track_iss()` *код, который взаимодействует с веб-службой*

Наконец, правилом хорошего тона является создание функции `main()`

Итак, ниже приведен код отслеживания положения МКС в режиме реального времени. Запустив программу, вы сможете получать координаты Международной космической станции каждые 5 секунд. В этом упражнении вашей задачей будет понять, как работает данный код. Внимательно изучите его и постарайтесь догадаться, как все происходит. Объяснения будут даны в следующей главе.

```
import requests, json, turtle

iss = turtle.Turtle()

def setup(window):
    global iss

    window.setup(1000, 500)
    window.bgpic('earth.gif')
    window.setworldcoordinates(-180, -90, 180, 90)
    turtle.register_shape("iss.gif")
    iss.shape("iss.gif")

def move_iss(lat, long):
    global iss

    iss.penup()
    iss.goto(long, lat)
    iss.pendown()

def track_iss():
    url = 'http://api.open-notify.org/iss-now.json'
    response = requests.get(url)
    if (response.status_code == 200):
        response_dictionary = json.loads(response.text)
        position = response_dictionary['iss_position']
        lat = float(position['latitude'])
        long = float(position['longitude'])
        move_iss(lat, long)
    else:
        print("Хьюстон, у нас проблема:", response.status_code)
    widget = turtle.getcanvas()
    widget.after(5000, track_iss)

def main():
    global iss
    screen = turtle.Screen()
    setup(screen)
    track_iss()

if __name__ == "__main__":
    main()
    turtle.mainloop()
```

Мы добавили всего две строки кода

С помощью этого кода можно видеть, как меняется положение МКС каждые 5 секунд

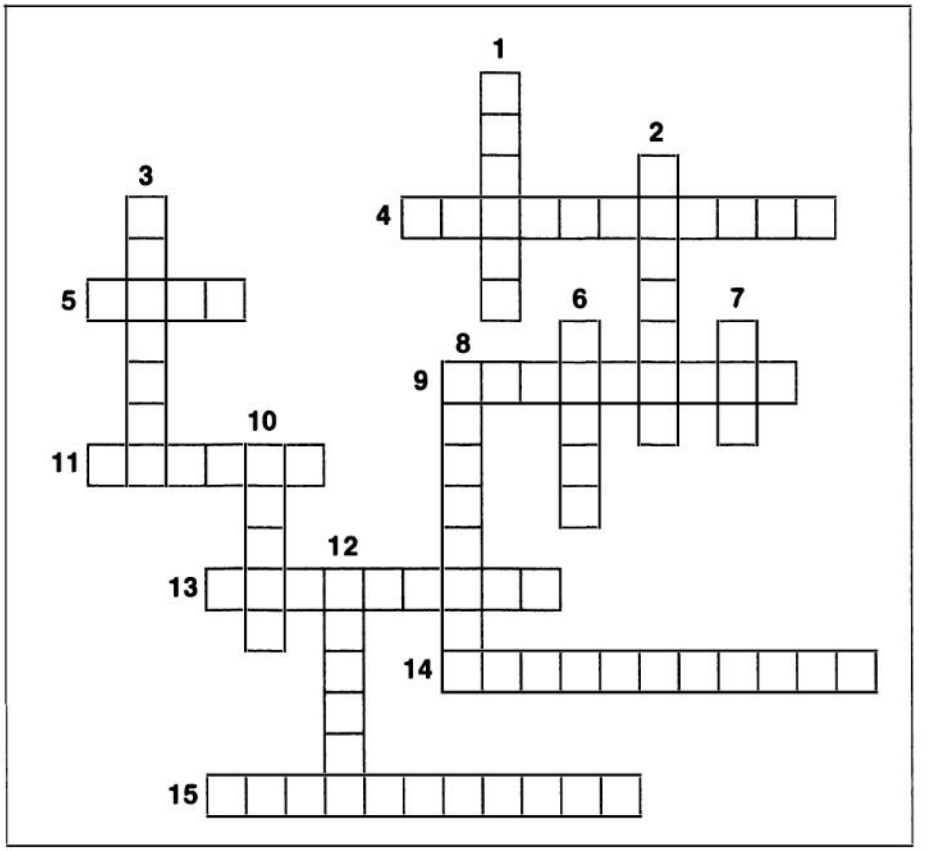
САМОЕ ГЛАВНОЕ

- С помощью Python можно взаимодействовать с веб-службами, обрабатывая получаемые от них данные в приложении.
- Веб-служба представляет собой задокументированный программный интерфейс (Web API), возвращающий данные определенного вида.
- Для доступа к веб-службе обычно требуется зарегистрироваться и получить специальный ключ или маркер авторизации.
- Взаимодействие с веб-службами в коде Python осуществляется путем составления запросов по протоколу HTTP, как при работе в браузере.
- Веб-службы реагируют на запросы, возвращая запрашиваемые данные, обычно в формате JSON.
- JSON расшифровывается как "JavaScript Object Notation". Это стандартный формат обмена данными между разными языками программирования.
- Синтаксис JSON напоминает словари Python.
- Существует бесплатный пакет `requests`, упрощающий работу с веб-запросами в Python.
- Пакет — это набор модулей Python.
- Пакет `requests` можно установить с помощью утилиты `pip`.
- Название `pip` представляет собой рекурсивный акроним "`pip` installs packages".
- Для отправки веб-запроса используется функция `get()` модуля `requests`.
- Функция `get()` возвращает объект `response`, в котором содержится код состояния, текст ответа (в формате JSON) и заголовки.
- Код состояния 200 означает, что запрос обработан успешно.
- Встроенный модуль `json` содержит методы для преобразования строки JSON в словарь или список Python.
- Объект `screen` модуля `turtle` позволяет изменить фон и систему координат окна черепашек.

Кроссворд

Добро пожаловать обратно на Землю! Дайте передышку своему мозгу, загрузив правое полушарие чем-то увлекательным.

Как и раньше, перед вами кроссворд, состоящий из слов, которые встречались на протяжении данной главы.



По горизонтали

4. Графический объект
5. Разрешает доступ к веб-службе
9. Добавление пакета в систему
11. Обращение к веб-службе
13. Формат данных в JSON
14. Получение разрешения на доступ к веб-службе
15. Редактирование чужого кода

По вертикали

1. Координата, определяющая расстояние до экватора
2. JSON в русской транскрипции
3. Координата, определяющая расстояние до нулевого меридиана
6. Набор взаимосвязанных и совместно распространяемых модулей
7. Международная космическая станция
8. Формирование ответа на запрос
10. Реакция веб-службы на запрос
12. Код, выполняемый на стороне сервера

УПРАЖНЕНИЕ (РЕШЕНИЕ)

Введите следующий URL-адрес в браузере. Что вы получите?

http://api.open-notify.org/iss-now.json

Отлично! Мы только что определили текущее местоположение МКС. Словарь содержит несколько ключей, причем ключ `iss_position` *содержит свой набор пар "ключ/значение" (широта и долгота)*

```
{
 "message": "success",
 "timestamp": 1500664795,
 "iss_position":
      {
         "longitude": "-110.6066",
         "latitude": "-50.4185"
      }
}
```

Вот что мы получили

Форматирование зависит от браузера. Некоторые браузеры могут даже загружать данные в виде текстового файла

Учтите, что широта и долгота представлены в строковом виде. Их нужно будет преобразовать в значения типа `float` *перед использованием*

Возьмите карандаш
Решение

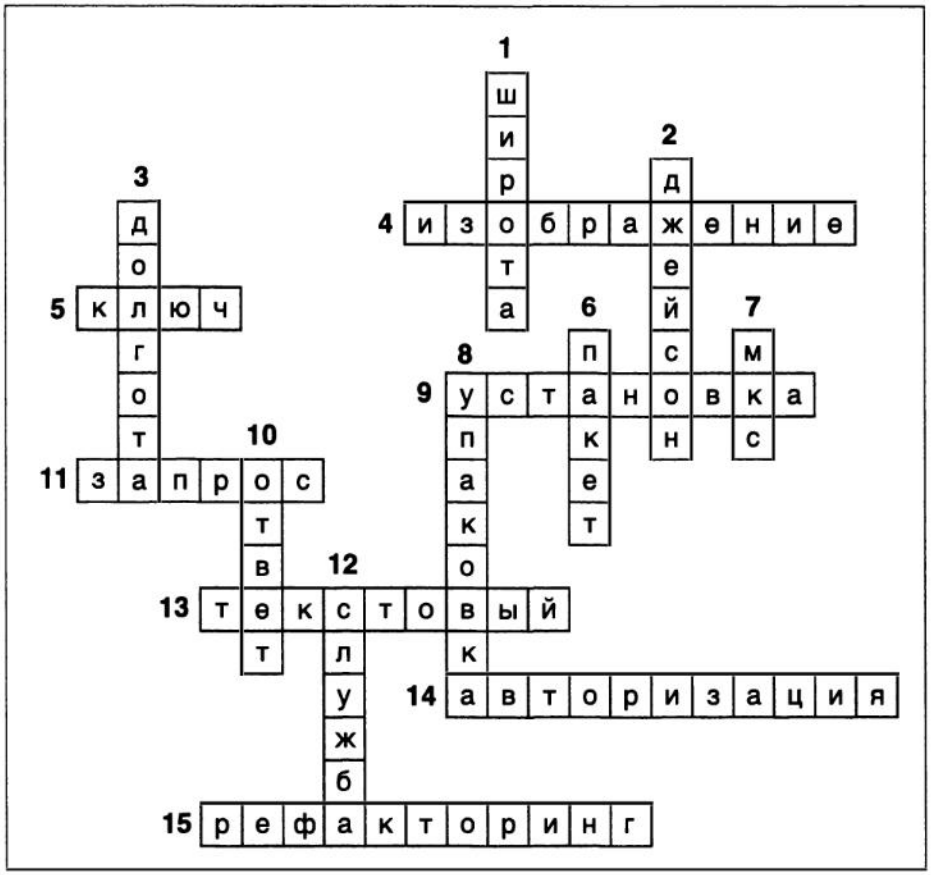
Преобразуйте данные JSON, приведенные на предыдущей странице, в словарь Python и завершите приведенный ниже фрагмент программы, заполнив прочерки. В качестве подсказки сверяйтесь с результатами работы программы.

```
current = {'temperature': 22.0,
           'humidity': '58%',
           'location': {
                  'city': 'London',
                  'country': 'UK'
            }
          }
loc = current['location']
print('Город:',
      loc['city'] + ', ' + loc['country'] +
      '. Температура:',
      current['temperature'], '°C.')
```

Оболочка Python

```
Город: London, UK. Температура: 22.0 °C.
```

Кроссворд: решение



11 Виджеты, события и непредсказуемое поведение

Интерактивные возможности

Это глава для сильных духом. Будьте упорны и трудолюбивы, проявите терпение и настойчивость — и вам обязательно удастся выйти на новый уровень программирования!

Вам уже приходилось писать простые графические приложения, но у них не было полноценного графического интерфейса. Другими словами, приложения не позволяли пользователям взаимодействовать с экранными элементами. Чтобы это стало возможным, необходимо применить иную модель выполнения программы, в рамках которой приложение **реагирует** на действия пользователя. Пользователь щелкнул на кнопке? Программа должна знать, как реагировать на такое событие, и быть готовым к нему. Процесс создания графического интерфейса сильно отличается от привычного процедурного стиля программирования, который мы применяли до сих пор. Нам придется совершенно по-иному подойти к решению задач. В этой главе вы создадите свой первый графический интерфейс, причем не какое-то примитивное экранное приложение, а нечто намного более интересное. Мы напишем игровой симулятор искусственной жизни с непредсказуемым поведением. Что это значит? Сейчас узнаете...

Откройте для себя удивительный мир игры «ИСКУССТВЕННАЯ ЖИЗНЬ»

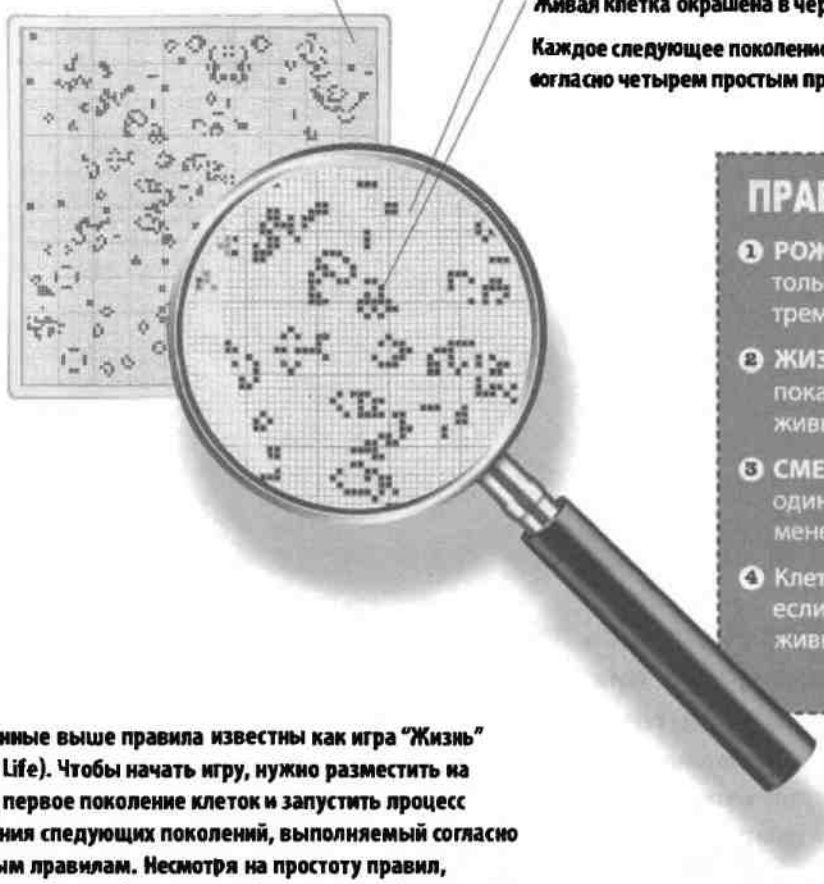
Станьте творцом НОВОЙ ЖИЗНИ, просто написав код! Воспользовавшись **невероятно простым алгоритмом**, основанным всего на **четырех правилах**, вы создадите искусственную жизнь с **непредсказуемым поведением**. Вот как это работает.

Наша искусственная жизнь состоит из набора клеток, формирующих решетку.

Каждая клетка может быть либо живой, либо мёртвой.

Живая клетка окрашена в черный цвет

Каждое следующее поколение клеток рассчитывается согласно четырем простым правилам.



ПРАВИЛА ИГРЫ

1. **РОЖДЕНИЕ:** новая клетка рождается, только если она окружена ровно тремя живыми клетками.
2. **ЖИЗНЬ:** клетка живет до тех пор, пока с ней граничат две или три живые клетки.
3. **СМЕРТЬ:** клетка умирает от одиночества, если с ней граничит менее двух живых клеток.
4. Клетка умирает от перенаселенности, если с ней граничат четыре и более живых клеток.

Приведенные выше правила известны как игра "Жизнь" (Game of Life). Чтобы начать игру, нужно разместить на решетке первое поколение клеток и запустить процесс вычисления следующих поколений, выполняемый согласно указанным правилам. Несмотря на простоту правил, единственный способ узнать, как будет вести себя популяция клеток, — это *понаблюдать за ней*. Поведение клеток настолько непредсказуемо, что никогда заранее нельзя определить, прекратит ли популяция свое существование. Для этого придется выполнять моделирование, чем мы сейчас и займемся, написав свой собственный симулятор игры "Жизнь"!

Игра "Жизнь" придумана английским математиком Джоном Конвеем; см. описание игры в Википедии:

https://ru.wikipedia.org/wiki/Игра_«Жизнь»

Правила игры „Жизнь"

Итак, вы знаете четыре простых правила. Давайте попробуем понять, как играть в игру и какова ее специфика. Как известно, все клетки размещаются в ячейках решетки. Каждая ячейка содержит либо живую, либо мертвую клетку. Если клетка жива, она окрашивается, чтобы ее было видно. Мертвая клетка остается прозрачной.

Во "вселенной" игры "Жизнь" новые поколения клеток рассчитываются последовательно. На каждом шаге эволюции все четыре правила применяются к каждой из клеток текущего поколения, после чего все клетки разом обновляются. Процесс повторяется снова и снова, поколение за поколением. Рассмотрим, как применяются правила в типичных случаях.

Сфокусируемся на центральной клетке в каждом примере

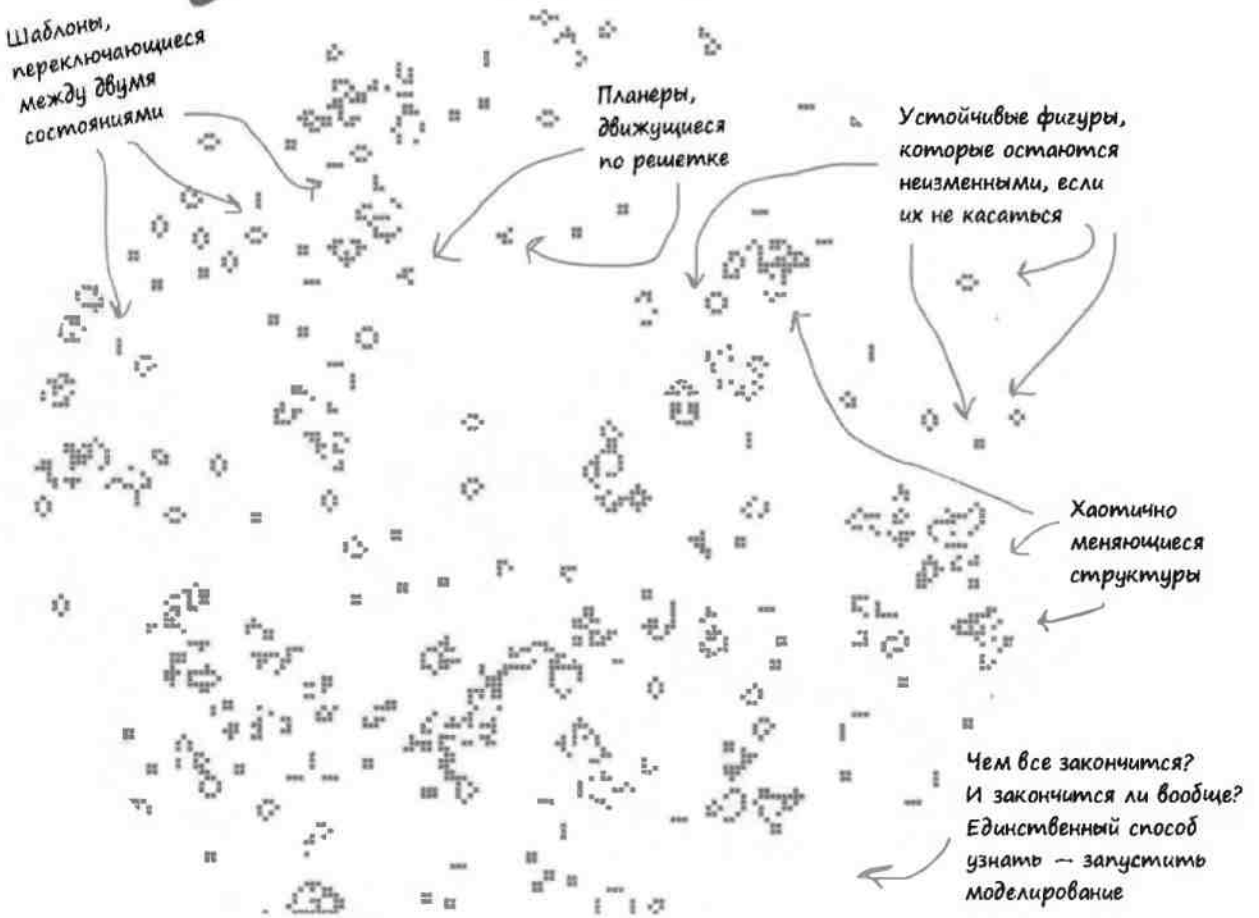
Для каждой клетки мы считаем число ее соседей

В данном случае центральная клетка мертва, в отличие от других примеров

число соседей = 0	число соседей = 1	число соседей = 2	число соседей = 3	число соседей = 4	число соседей = 3
Клетка, у которой нет соседей, умирает от одиночества	Клетка, у которой один сосед, умирает от одиночества	При двух соседях клетка продолжает жить	При трех соседях клетка продолжает жить	Если у клетки 4 (5, 6, 7...) соседей, она умирает от перенаселенности	В пустой клетке, рядом с которой три живые клетки, зарождается жизнь!

Вы не поверите...

Игра "Жизнь" является примером **порождающей системы**. Подобные системы строятся на предопределенном, часто простом, наборе правил, но при этом демонстрируют непредсказуемое поведение, которое невозможно предугадать. Порождающие системы находят применение в искусстве, музыке, машинном обучении и являются частым предметом философских дискуссий о происхождении Вселенной. Но лучшего примера порождающих систем, чем игра "Жизнь "Джона Конвея, пожалуй, не найти. На основе четырех простых правил рождается множество шаблонов поведения, с которыми вы столкнетесь, когда запустите разработанный нами симулятор.



Возьмите карандаш

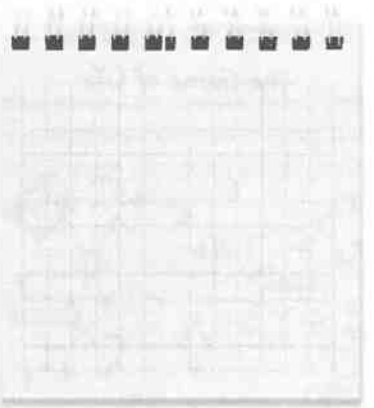
Прежде чем приступать к написанию кода, попробуйте вручную проследить за судьбой нескольких поколений клеток. В качестве начального используйте поколение 1 и вычислите для него поколения 2–6, применяя правила игры "Жизнь". Повторим их еще раз:

- мертвая клетка, у которой три живые соседние клетки, оживает в следующем поколении;
- живая клетка, у которой две или три живые соседние клетки, продолжает жить в следующем поколении;
- живая клетка, у которой менее двух живых соседних клеток, умирает в следующем поколении;
- живая клетка, у которой более трех живых соседних клеток, умирает в следующем поколении.

Это первое поколение

Не забывайте проверять пустые ячейки, окруженные живыми клетками: возможно, в них зарождается жизнь!

Поколение 1



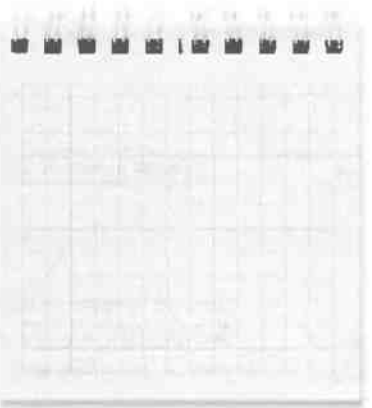
Поколение 2

Поколение 3

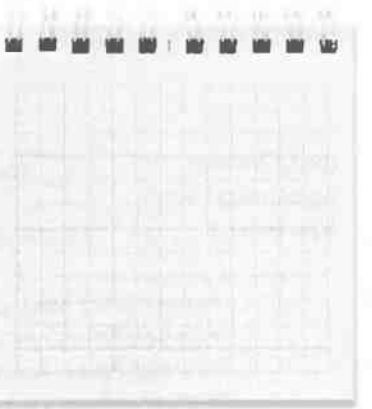
Примените правила игры "Жизнь" и пометьте результаты для поколения 2

Продолжайте применять правила к последующим поколениям

Поколение 4



Поколение 5

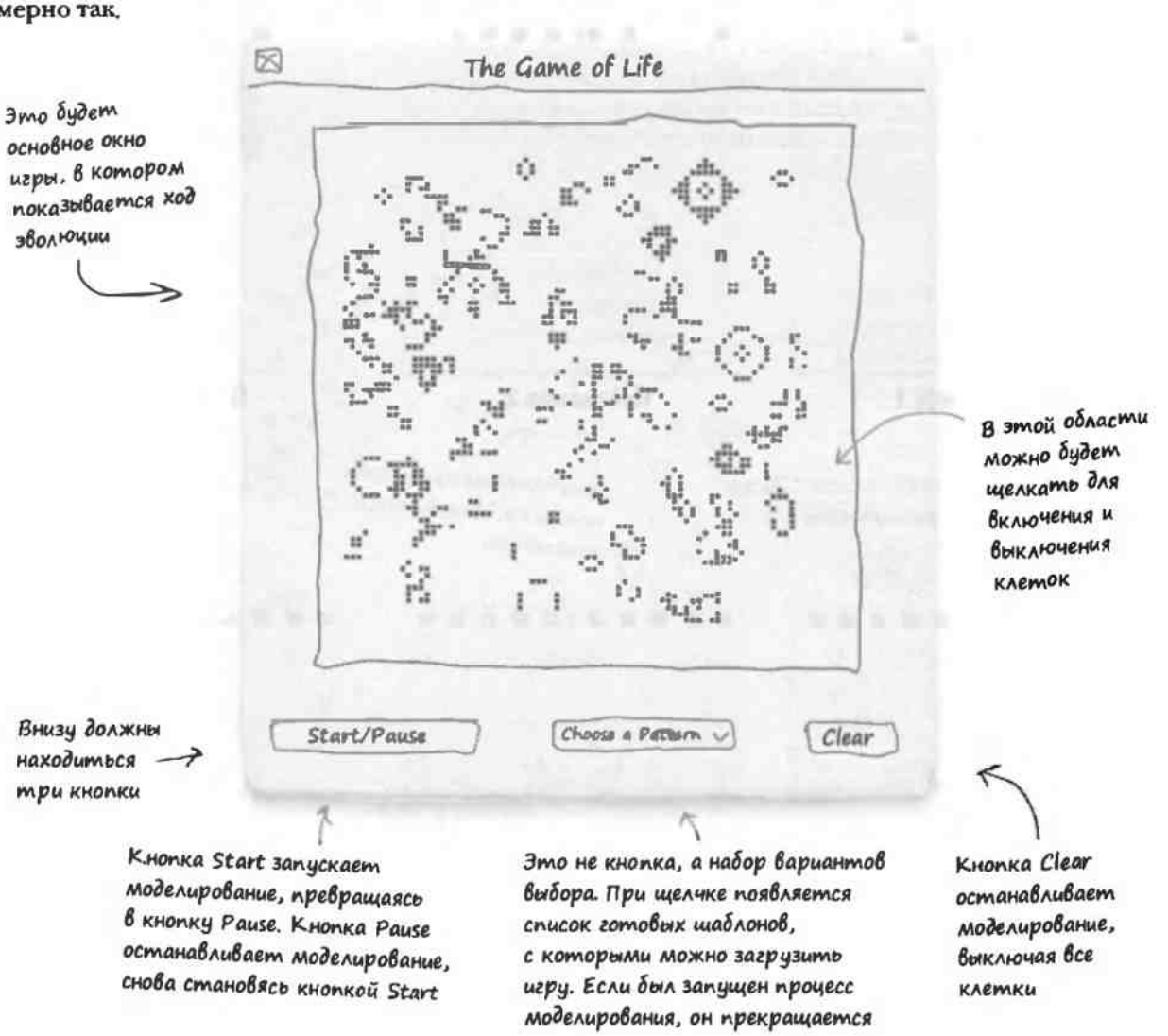


Поколение 6

Что нам предстоит создать

Как вы уже поняли из предыдущего упражнения, для анализа порождающей системы с непредсказуемым поведением, каковой является игра "Жизнь", без компьютера просто не обойтись. Нам предстоит разработать *симулятор* игры. Он будет отображать решетку, с которой пользователь сможет взаимодействовать, щелкая на ячейках для создания клеток. Под игровым полем будет находиться несколько управляющих кнопок. Прежде всего, симулятор нужно снабдить кнопками запуска и остановки игры. Также мы добавим кнопку очистки игрового поля от имеющихся клеток. Кроме того, нам понадобится инструмент выбора готовых фигур, или конфигураций клеток. Другими словами, нам предстоит создать полноценный графический интерфейс.

Прежде чем заняться построением графического интерфейса, нужно набросать эскиз на листе бумаги. Не пренебрегайте этим этапом — он чрезвычайно полезен для предварительной визуализации графических идей. В нашем случае эскиз выглядит примерно так.



Дизайнерские решения

Отдельным этапом создания симулятора будет проектирование и тестирование графического интерфейса. На практике этим занимаются профессиональные дизайнеры, которые берут деньги за свою работу. Но даже если в нашем распоряжении нет многотысячного бюджета на тестирование удобства использования, существуют простые и эффективные методики, которые помогут нам улучшить создаваемый интерфейс. Одна из таких методик — *бумажное прототипирование*, которое заключается в рисовании макета графического интерфейса на бумаге (уже сделано!) с последующим моделированием *сценариев использования*, когда тестовая группа пользователей пытается работать с макетом так, как если бы это был полноценный графический интерфейс. Применение такой методики позволяет понаблюдать за тем, как работают реальные пользователи и какие ошибки они допускают.

Тестирование удобства использования предполагает привлечение конечных пользователей в качестве тестировщиков

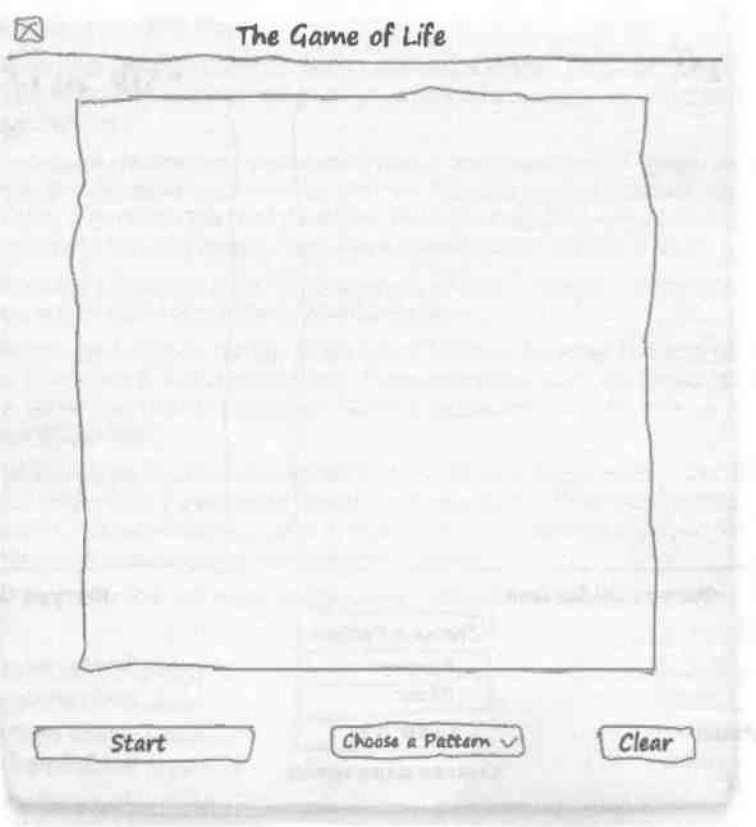
Сценарии использования — это набор операций или действий, выполняемых типичным пользователем

По мнению специалистов, эта методика позволяет выявить до 85% ошибок в работе графического интерфейса

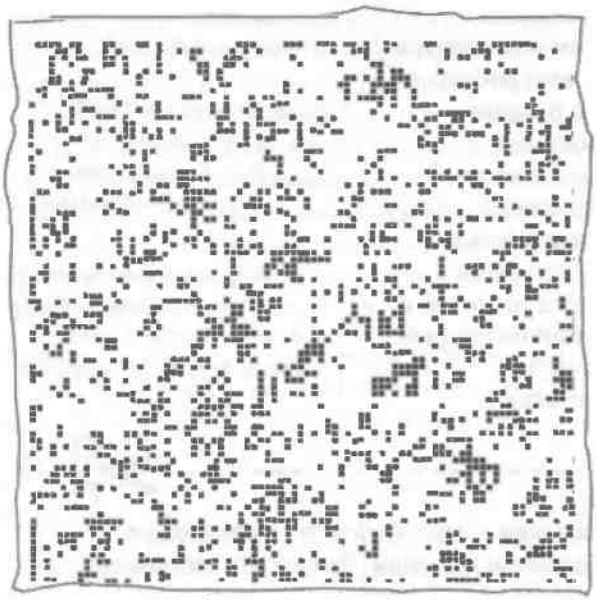
УПРАЖНЕНИЕ

Пора заняться бумажным прототипированием. Отсканируйте эту и следующую страницу и обрежьте рисунки по обозначенным границам. Пригласите нескольких друзей, положите перед ними прототип и попросите ответить на несколько вопросов (их список будет приведен далее).

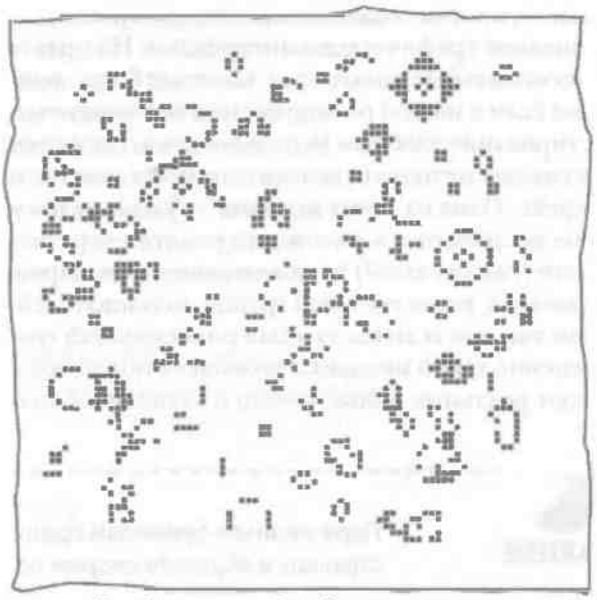
Можете вырезать этот шаблон в качестве образца интерфейса



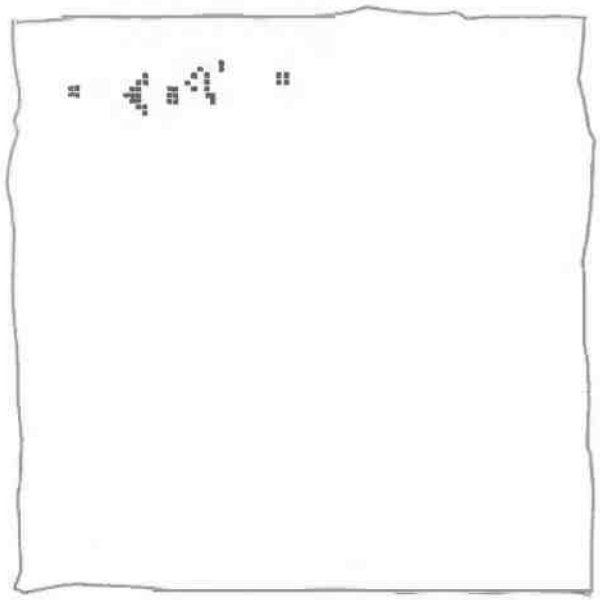
Основной интерфейс



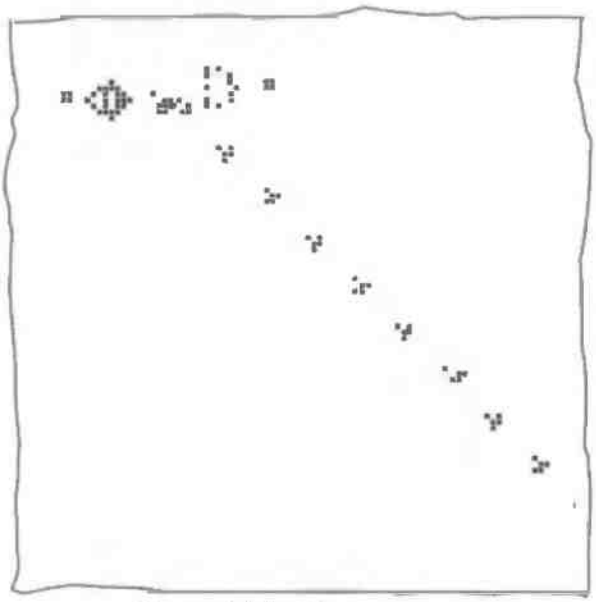
Конфигурация Random



Конфигурация Random в движении



Фигура Glider Gun



Фигура Glider Gun в движении

Pause

КнопкаPause

Choose a Pattern
Random
Glider
Glider Gun

Список вариантоа

Вырежьте каждый элемент по отдельности. Всего у вас будет шесть элементов (семь с учетом предыдущей страницы)

УПРАЖНЕНИЕ

Ниже описан сценарий, которого должен придерживаться каждый пользователь-тестировщик. Попросите друзей комментировать вслух все свои действия по мере выполнения упражнения. Фиксируйте все совершаемые ими ошибки и случаи неправильного использования элементов интерфейса. Можно, конечно, отмечать и те блоки, которые работают правильно.

1. Положите прототип **"Основной интерфейс"** перед тестировщиком.
2. Объясните тестировщику правила игры "Жизнь" и принципы работы симулятора.
3. Скажите тестировщику, что самый простой способ начать игру — загрузить конфигурацию *Random* (Произвольный шаблон), и спросите, понимает ли он, как это сделать.

   Если пользователь ответит, что нужно щелкнуть на кнопке *Choose a Pattern* (Выбрать шаблон), поместите поверх нее прототип **"Список вариантов"**.

   Если далее пользователь скажет, что нужно щелкнуть на варианте *Random* (Произвольный шаблон), уберите прототип **"Список вариантов"** и поместите на игровое поле прототип **"Конфигурация Random"**.

   Скажите пользователю, что он только что загрузил шаблон с произвольным расположением клеток.
4. Спросите у тестировщика, понимает ли он, как начать моделировать следующие поколения клеток.

   Дождитесь, пока пользователь выберет кнопку *Start* (Начать). Если этого не произошло, дайте пользователю несколько подсказок. После выбора кнопки *Start* положите на игровое поле прототип **"Конфигурация Random в движении"**. Также поместите прототип **"Кнопка Pause"** поверх кнопки *Start*. Объясните пользователю, что он сейчас наблюдает процесс генерации новых и гибели существующих клеток.
5. Попросите тестировщика очистить игровое поле. Если он не может найти кнопку *Clear* (Очистить), дайте ему несколько подсказок. После выбора кнопки *Clear* уберите прототипы **"Конфигурация Random в движении"** и **"Кнопка Pause"**.
6. Попросите тестировщика вручную добавить несколько живых клеток. При необходимости подсказывайте ему, пока он не додумается щелкнуть мышью на игровом поле. Нарисуйте ручкой клетки прямо на прототипе.
7. Попросите тестировщика сгенерировать несколько новых поколений клеток. Дождитесь, пока он выберет кнопку *Start*, после чего поместите прототип **"Конфигурация Random в движении"** поверх игрового поля и прототип **"Кнопка Pause"** — поверх кнопки *Start*. Объясните пользователю, что он сейчас наблюдает процесс генерации новых и гибели существующих клеток.
8. Попросите тестировщика приостановить генерирование клеток. Подождите, пока он выберет кнопку *Pause*, после чего уберите прототип **"Кнопка Pause"**.
9. Попросите тестировщика загрузить фигуру *Glider Gun* (Планерное ружье). Когда он выберет кнопку *Choose a Pattern*, поместите поверх нее прототип **"Список вариантов"**. Как только пользователь выберет вариант *Glider Gun*, уберите прототип **"Список вариантов"** и поместите на игровое поле прототип **"Фигура Glider Gun"**.
10. Попросите тестировщика начать моделирование. Как только он выберет кнопку *Start*, поместите прототип **"Фигура Glider Gun в движении"** поверх игрового поля и прототип **"Кнопка Pause"** — поверх кнопки *Start*. Объясните пользователю, что он сейчас наблюдает генерацию бесконечного потока фигур в виде планеров, перемещающихся по решетке.
11. Сообщите пользователю о том, что этап тестирования завершен, и поблагодарите его за участие.

ПРИМЕЧАНИЕ: *если вы обнаружили ошибки в ходе тестирования, то нам очень хотелось бы исправить их, но, к сожалению, мы уже не сможем этого сделать, так как книга давно напечатана. Тем не менее держите их в голове, когда будете читать главу, и подумайте о том, как их можно было бы исправить.*

Не помешает иметь под рукой печеньки в качестве угощения

Создание симулятора игры „Жизнь"

Разработав дизайн графического интерфейса и выполнив бумажное прототипирование, мы можем приступать к программной реализации. Но сначала нужно разобраться с тем, как создаются такого рода интерфейсы.

Мы обратимся к проверенной методике, которая повсеместно применяется при разработке пользовательских интерфейсов коммерческого уровня. Она предполагает разделение кода на три принципиальные части: *модель* данных, графическое *представление* (пользовательский интерфейс) и *контроллер* (управляющая логика). Такая схема разделения приложения называется MVC (Model-View-Controller).

Разделение кода на составляющие — модель, представление и контроллер — основано на приемах объектно-ориентированного программирования, с которыми нам предстоит познакомиться в следующей главе. Поэтому мы не будем следовать методике MVC буквально, а лишь воспользуемся ею как концептуальной схемой, постаравшись все максимально упростить.

В схеме MVC структура приложения определяется следующим образом.

Представление

Код **представления** управляет элементами интерфейса. Он не связан с процессом моделирования, занимаясь лишь отображением данных в игровом поле

Контроллер

Контроллер получает запросы от пользователя (например, начать игру) и преобразует их в запросы к представлению или модели

Модель

Код **модели** занимается только вычислением данных для игрового поля и клеток, а также расчетом следующих поколений. Он не связан с представлением

Существует целое направление разработки программного обеспечения — шаблоны проектирования, которые обобщают опыт создания приложений

Итак, у нас будут три программные составляющие: 1) представление, которое управляет только отображением экранных элементов; 2) модель, которая занимается только вычислением клеток и ничего не знает о том, как они будут отображаться на экране; 3) контроллер, который управляет взаимодействием с пользователем, в случае необходимости передавая команды представлению и модели.

Почему так сложно? Зачем нужна схема MVC? Опыт разработки показывает, что приложения, снабженные графическим интерфейсом, быстро становятся слишком запутанными, превращаясь в плохо управляемый спагетти-код (это устоявшийся технический термин). Методика MVC позволяет избежать этого, фокусируясь в каждом компоненте на одной конкретной задаче.

На данном этапе от вас не требуется глубокое понимание MVC. Главное, чтобы вы представляли общую идею, которая поможет нам в построении симулятора игры "Жизнь". А теперь к делу!

Построение модели данных

Несмотря на то что концепция пользовательского интерфейса в целом ясна, мы отложим написание его кода на потом, а пока займемся созданием модели данных для нашего симулятора. Как уже было сказано, под моделью данных мы понимаем код, отвечающий за вычисление клеток и генерацию всех последующих поколений.

Формирование решетки

Для построение решетки мы будем хранить массив целых чисел, в котором значение `0` обозначает мертвую клетку, а значение `1` — живую.

Вы уже знаете, как в Python создаются линейные списки, но сейчас нам нужно получить двухмерную структуру, характеризующуюся шириной и высотой. Как создать двухмерный список? Вложив один список в другой. Предположим, необходимо получить решетку размером четыре элемента в ширину и три элемента в высоту. Это можно сделать следующим образом.

```
my_grid = [[0, 1, 2, 3],
           [4, 5, 6, 7],
           [8, 9, 10, 11]]
```

Смоделировать двухмерную решетку можно с помощью списка списков

Это двухмерная решетка размером 3x4

В Python существует специальный синтаксис для доступа к элементам двухмерных списков.

```
value = my_grid[2][3]
print(value)
```

Получить значение в ячейке на пересечении строки с номером 2 и столбца с номером 3

Помните: индексы считаются от 0!

Оболочка Python

```
11
>>>
```

Итак, давайте создадим список для хранения клеток на решетке.

Изучите этот код и убедитесь, что он действительно создает список списков размером 100x100

```
height = 100
width = 100

grid_model = [0] * height

for i in range(height):
    grid_model[i] = [0] * width
```

В этих двух глобальных переменных хранятся ширина и высота сетки

Помните правила умножения списков из главы 4? Здесь создается список, число нулей в котором равно `height`

Можно было бы предположить, что значения списка должны быть булевыми (т.к. клетка либо жива, либо мертва). Такой выбор был бы оправдан, однако использование целых чисел снижает сложность кода, упрощая подсчет живых соседей клетки

Проходим по каждому элементу списка `grid_model` и заменяем его списком, число нулей в котором равно `width`

Вычисление поколений клеток

После создания списка, в котором будут храниться клетки, нужно написать код для вычисления их последующих поколений. Для этого следует проанализировать окружение каждой клетки и выяснить, продолжит ли она жить, умрет или родится в следующем поколении. Напишем код просмотра всех клеток, хранящихся в списке.

Создаем функцию `next_gen()` *для расчета следующего поколения*

```
def next_gen():
    global grid_model

    for i in range(0, height):
        for j in range(0, width):
```

Проходим по каждой ячейке двухмерного списка...

Это вложенный список: для каждого значения `i` *мы проходим по каждому значению* `j`

...и определяем, будет ли клетка `grid_model[i][j]` *жить, умирать или рождаться в следующем раунде*

Расчеты проводятся для каждой клетки

Определение судьбы клеток

Поведение клеток в каждом следующем поколении определяется правилами игры "Жизнь". Рассмотрим их еще раз:

- мертвая клетка, у которой три живые соседние клетки, оживает в следующем поколении;
- живая клетка, у которой две или три живые соседние клетки, продолжает жить в следующем поколении;
- живая клетка, у которой менее двух живых соседних клеток, умирает в следующем поколении;
- живая клетка, у которой более трех живых соседних клеток, умирает в следующем поколении.

Эти правила определяют логику игры и применяются к каждой просматриваемой клетке. Если клетка мертва, нужно проверить, есть ли у нее три живые соседние клетки. Если это так, то в следующем поколении в текущей ячейке рождается новая клетка. Если клетка живая и у нее две или три живые соседние клетки, то она продолжает жить дальше. Во всех остальных случаях клетка умирает в следующем поколении. Следовательно, несмотря на наличие четырех правил, логику игры можно представить всего двумя условиями, описывающими случаи, когда клетка жива или мертва. Соответствующий код будет рассмотрен далее, но сначала нам нужно подумать над тем, как определить количество живых клеток по соседству с текущей. Самый очевидный способ — изучить все соседние клетки и подсчитать, сколько из них живых.

Рассмотрим, как это делается.

Функция count_neighbors() получает решетку, а также номер строки и столбца и возвращает число живых соседей для заданной ячейки

```
def count_neighbors(grid, row, col):
    count = 0
    if row-1 >= 0:
        count = count + grid[row-1][col]
    if (row-1 >= 0) and (col-1 >= 0):
        count = count + grid[row-1][col-1]
    if (row-1 >= 0) and (col+1 < width):
        count = count + grid[row-1][col+1]
    if col-1 >= 0:
        count = count + grid[row][col-1]
    if col+1 < width:
        count = count + grid[row][col+1]
    if row+1 < height:
        count = count + grid[row+1][col]
    if (row+1 < height) and (col-1 >= 0):
        count = count + grid[row+1][col-1]
    if (row+1 < height) and (col+1 < width):
        count = count + grid[row+1][col+1]
    return count
```

Фактически эта функция добавляет число живых клеток к счетчику соседей заданной клетки

Код усложняется, потому что необходимо учитывать положение клеток на краях игрового поля

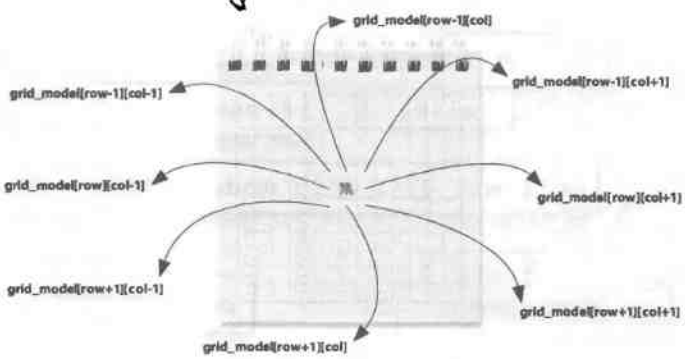
Готовый рецепт

Здесь приведен готовый код. Вам нужно просто ввести его (или скопировать из папки с исходными файлами). Наверняка вы легко поймете, как он работает (в нем нет ничего нового или сложного), поэтому детальный анализ кода не входит в наши задачи.

Тем не менее анализ рецепта будет хорошей тренировкой для мозга. Самое сложное здесь — не определить количество живых соседних клеток, а учесть, что клетки, находящиеся у края решетки, имеют меньше соседей, чем внутренние клетки. Так что потратьте время, чтобы разобраться в работе кода.

Приятного изучения!

Вот как получить доступ к соседним клеткам



Код на магнитиках

Вам повезло! Пока вы изучали код готового рецепта, мы не стали терять время и создали функцию `next_gen()`, код которой записали на магнитиках, прикрепленных к дверце холодильника. Но, как всегда, кто-то решил усложнить нам задачу и все перемешал. Не поможете восстановить исходный код функции? Только будьте внимательны: некоторые магнитики могут оказаться лишними.

Напомним правила игры

ПРАВИЛА ИГРЫ

РОЖДЕНИЕ: новая клетка рождается, только если она окружена ровно тремя живыми клетками.

ЖИЗНЬ: клетка живет до тех пор, пока с ней граничат две или три живые клетки.

СМЕРТЬ: клетка умирает от одиночества, если с ней граничит менее двух живых клеток.

Клетка умирает от перенаселенности, если с ней граничат четыре и более живых клеток.

Этот фрагмент мы собрали за вас

```
def next_gen():
    global grid_model
    for i in range(0, height):
        for j in range(0, width):
            cell = 0
```

```
cell = 1
    else:
cell = 1
cell = 0
```

```
and
or
if count == 3:
if grid_model[i][j] == 0:
```

```
if count == 2 or count == 3:
elif grid_model[i][j] == 1:
count < 2:
if count > 4
count = count_neighbors(grid_model, i, j)
```

Прежде чем двигаться дальше, давайте упорядочим имеющийся код и протестируем его, чтобы потом не было неприятных сюрпризов. Скопируйте приведенный ниже код в файл *model.py* и не забудьте включить в него функцию, описанную в разделе "Готовый рецепт". Пока что программа не делает ничего особенного, но проверить ее работоспособность не помешает.

```
height = 100
width = 100

grid_model = [0] * height
for i in range(height):
   grid_model[i] = [0] * width

def next_gen():
   global grid_model

   for i in range(0, height):
       for j in range(0, width):
           cell = 0
           print('Проверяем клетку', i, j)
           count = count_neighbors(grid_model, i, j)

           if grid_model[i][j] == 0:
               if count == 3:
                   cell = 1
           elif grid_model[i][j] == 1:
               if count == 2 or count == 3:
                   cell = 1

def count_neighbors(grid, row, col):

if __name__ == '__main__':
    next_gen()
```

Соединим код, приведенный на предыдущих страницах

Не пропустите эту строку, которая вставлена для целей тестирования

Готовый рецепт

Вставьте целиком функцию из раздела "Готовый рецепт"

Если вывод занимает слишком много времени, вы всегда можете остановить программу, выполнив команду Shell > Interrupt Execution

Если программа работает корректно, то она должна проверить все клетки

Вот что мы получили. Не слишком много, но это только начало

Оболочка Python

```
Проверяем клетку 99 76
Проверяем клетку 99 77
Проверяем клетку 99 78
Проверяем клетку 99 79
Проверяем клетку 99 80
Проверяем клетку 99 81
Проверяем клетку 99 82
Проверяем клетку 99 83
Проверяем клетку 99 84
Проверяем клетку 99 85
Проверяем клетку 99 86
Проверяем клетку 99 87
Проверяем клетку 99 88
Проверяем клетку 99 89
Проверяем клетку 99 90
Проверяем клетку 99 91
Проверяем клетку 99 92
Проверяем клетку 99 93
Проверяем клетку 99 94
Проверяем клетку 99 95
Проверяем клетку 99 96
Проверяем клетку 99 97
Проверяем клетку 99 98
Проверяем клетку 99 99
>>>
```

Завершение кода модели

Код файла *model.py* еще не завершен. Мы выстроили логический каркас, но на данный момент программа только вычисляет клетки следующего поколения, ничего не обновляя. Здесь возникает очевидная проблема: если сохранять значения клеток следующего поколения в текущей решетке (по мере их вычисления), то результаты, возвращаемые функцией `count_neighbors()`, потеряют смысл, так как будут базироваться на смешанных данных текущего и следующего поколений. Для устранения проблемы нам понадобятся *две решетки*: одна – для хранения текущих значения, другая – для следующего поколения. Когда вычисление следующего поколения будет завершено, его нужно будет сделать текущим. Вот как это реализуется.

```
grid_model = [0] * height
next_grid_model = [0] * height

for i in range(height):
   grid_model[i] = [0] * width
   next_grid_model[i] = [0] * width

def next_gen():
   global grid_model, next_grid_model

   for i in range(0, height):
       for j in range(0, width):
           cell = 0
           print('Проверяем клетку', i, j)
           count = count_neighbors(grid_model, i, j)

           if grid_model[i][j] == 0:
               if count == 3:
                   cell = 1
           elif grid_model[i][j] == 1:
               if count == 2 or count == 3:
                   cell = 1
           next_grid_model[i][j] = cell

   temp = grid_model
   grid_model = next_grid_model
   next_grid_model = temp
```

Вместо того чтобы показывать весь файл, мы приводим лишь код, в котором есть изменения. Внесите их в свою программу

Создаем вторую решетку, `next_grid_model`

Добавляем глобальное объявление

После вычисления клетки сохраняем ее в соответствующей позиции решетки `next_grid_model`

Когда решетка `next_grid_model` *полностью вычислена, необходимо превратить ее в решетку* `grid_model`. *Для этого меняем их местами*

Не бойтесь задавать вопросы

В: Зачем менять местами решетки `next_grid_model` и `grid_model`? Почему нельзя просто присвоить переменной `grid_model` значение `next_grid_model`?

О: Такой вариант подходит, только если вычисляется одно-единственное поколение. Но если вычисления продолжатся, то вы получите две переменные, `next_grid_model` и `grid_model`, ссылающиеся на один и тот же список, и воспроизведете указанную выше проблему. Чтобы избежать этого, мы перед вычислением каждого следующего поколения меняем переменные местами. В результате переменная `grid_model` будет хранить текущее поколение, а в переменную `next_grid_model` будут записываться значения следующего поколения.

Возьмите карандаш

Мы не можем считать код функции `next_gen()` завершенным, если он тестируется на решетке из одних нулей. Напишите функцию `randomize()`, которая, получив решетку заданной ширины и высоты, случайным образом заполняет ее нулями и единицами.

```
import random

def randomize(grid, width, height):
```

Напишите здесь свой код

Тест-драйв

Выполним еще один тест. У нас пока нет возможности визуализировать модель (для этого придется создавать пользовательский интерфейс), поэтому все приходится тщательно тестировать. Приведем имеющийся код целиком.

```
import random

height = 100
width = 100

def randomize(grid, width, height):
   for i in range(0, height):
       for j in range(0, width):
           grid[i][j] = random.randint(0, 1)

grid_model = [0] * height
next_grid_model = [0] * height
for i in range(height):
   grid_model[i] = [0] * width
   next_grid_model[i] = [0] * width

randomize(grid_model, width, height)
```

Мы будем использовать модуль `random`

Добавляем функцию `randomize()` *в начало программы...*

...и вызываем ее после создания решетки

Продолжение следует...

```
def next_gen():
   global grid_model, next_grid_model

   for i in range(0, height):
       for j in range(0, width):
           cell = 0
           print('Проверяем клетку', i, j)
           count = count_neighbors(grid_model, i, j)

           if grid_model[i][j] == 0:
               if count == 3:
                   cell = 1
           elif grid_model[i][j] == 1:
               if count == 2 or count == 3:
                   cell = 1
           next_grid_model[i][j] = cell
           print('Новое значение:', next_grid_model[i][j])
   temp = grid_model
   grid_model = next_grid_model
   next_grid_model = temp

def count_neighbors(grid, row, col):

if __name__ == '__main__':
    next_gen()
```

Если вывод занимает слишком много времени, вы всегда можете остановить программу, выполнив команду Shell -> Interrupt Execution

Вот что мы получили. В вашем случае результаты будут другими, поскольку значения клеток выбираются случайным образом

Оболочка Python

```
Новое значение: 1
Проверяем клетку 99 91
Новое значение: 0
Проверяем клетку 99 92
Новое значение: 0
Проверяем клетку 99 93
Новое значение: 0
Проверяем клетку 99 94
Новое значение: 1
Проверяем клетку 99 95
Новое значение: 0
Проверяем клетку 99 96
Новое значение: 0
Проверяем клетку 99 97
Новое значение: 1
Проверяем клетку 99 98
Новое значение: 1
Проверяем клетку 99 99
Новое значение: 1
>>>
```

Добавьте еще один вызов `print()` для тестирования

Готовый рецепт

Что дальше?

Мы проделали большую работу, почти завершив создание модели данных для нашего симулятора. И хотя он еще не умеет отображать решетку с живыми и мертвыми клетками, он рассчитывает и сохраняет их для каждого следующего поколения.

Как бы там ни было, файл *model.py* уже можно рассматривать как готовый модуль, а значит, пора переходить к разработке пользовательского интерфейса, с помощью которого мы будем визуализировать модель данных и управлять ею.

Нас ждут новые задачи!

УПРАЖНЕНИЕ

Прежде чем двигаться дальше, удалите из функции `next_gen()` оба вызова функции `print()` — он больше не понадобятся.

Не забудьте об этом!

Экранное представление

Готовы наблюдать эволюцию на экране? Пора увидеть все своими глазами. В этом разделе мы займемся созданием экранного представления для нашего симулятора. Нам понадобится встроенный модуль `Tkinter`, позволяющий создавать графические интерфейсы с использованием типовых элементов управления, таких как кнопки, текстовые поля, меню и пр. В модуле `Tkinter` они называются *виджетами*. Ниже показаны виджеты, которые мы задействуем в приложении.

Основное окно — это виджет, включающий остальные виджеты

Решетка симулятора — это виджет-холст, позволяющий рисовать геометрические фигуры (мы будем рисовать множество квадратиков, представляющих живые клетки)

Clear — это виджет-кнопка

Start — это виджет-кнопка

Это виджет списка выбора; он позволяет выбирать из набора доступных вариантов

Это лишь малая часть виджетов, доступных в модуле `Tkinter`. Подробное описание его возможностей приведено в Википедии:

`https://ru.wikipedia.org/wiki/Tkinter`

Создание первого виджета

Мы понимаем, как виджеты будут выглядеть на экране, но как работать с ними в коде Python? С точки зрения программирования виджеты — обычные объекты. Как правило, сначала создается оконный виджет (он тут же отображается на экране), после чего создаются и добавляются в окно все остальные виджеты. Именно так мы и поступим: начнем с оконного виджета, а затем добавим в него кнопку Start. В модуле `Tkinter` класс, представляющий окно, называется `Tk`.

По-серьезному

Существует еще один, ранее не рассматривавшийся способ импорта модуля, который заключается в использовании инструкции `from`:

```
from tkinter import *
```

Благодаря такому объявлению нам больше не нужно предварять каждое имя функции, переменной или класса именем модуля. В частности, теперь вместо `tkinter.Tk()` можно писать просто `Tk()`.

Следите за регистром символов: в коде модуль называется `tkinter`, а класс — `Tk`

Начинаем с импорта модуля `Tkinter`, но иным способом

```
from tkinter import *

root = Tk()
root.title('The Game of Life')

start_button = Button(root, text='Start', width=12)

start_button.pack()

mainloop()
```

Создаем новое окно с помощью класса `Tk`

Окно верхнего уровня называют корневым, поэтому мы создали переменную `root`

Задаем строку заголовка, отображаемую вверху окна

При создании виджета указываем окно `root`, частью которого станет виджет

Далее создаем объект `Button`, передавая ему несколько аргументов: объект окна, надпись на кнопке и размер кнопки в символах

Эта команда просит модуль `Tkinter` поместить кнопку в окно там, где это возможно. Такая функция называется менеджером компоновки, о чем мы вскоре поговорим. Когда в окне отображается несколько виджетов, менеджер компоновки помогает позиционировать их

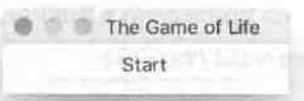
Наконец, как и в случае с черепашками, нужно передать управление модулю `Tkinter`, позволив ему обрабатывать щелчки и другие действия пользователя

Это функция из модуля `Tkinter`, и нам не нужно предварять ее вызов именем модуля, который был целиком импортирован

Тест-драйв

Добавьте приведенный выше код в файл *view.py* и протестируйте его.

Вы должны увидеть примерно такое окно. Но учтите, что его вид зависит от операционной системы и ее версии



Что произойдет, если щелкнуть на кнопке?

Если строка заголовка не отображается целиком, растяните окно вручную

Добавление остальных виджетов

Теперь можно приступать к добавлению в окно всех остальных виджетов. Помимо кнопки Start нам нужны: игровое поле (холст), кнопка Clear и список выбора. В будущем нам могут понадобиться и другие виджеты, поэтому вынесем весь код в функцию `setup()`. Введите приведенный ниже код в файл *view.py*.

```
from tkinter import *
import model

cell_size = 5

def setup():
    global root, grid_view, cell_size, start_button, clear_button, choice

    root = Tk()
    root.title('The Game of Life')

    grid_view = Canvas(root, width=model.width*cell_size,
                             height=model.height*cell_size,
                             borderwidth=0,
                             highlightthickness=0,
                             bg='white')

    start_button = Button(root, text='Start', width=12)
    clear_button = Button(root, text='Clear', width=12)

    choice = StringVar(root)
    choice.set('Choose a Pattern')
    option = OptionMenu(root, choice, 'Choose a Pattern', 'glider',
                        'glider gun', 'random')
    option.config(width=20)

    grid_view.pack()
    start_button.pack()
    option.pack()
    clear_button.pack()

if __name__ == '__main__':
    setup()
    mainloop()
```

Представлению понадобится доступ к модулю `model` (который мы написали), поэтому импортируйте его

Вскоре вы узнаете, для чего это нужно. Клетки на экране имеют размер больше одного пикселя, поэтому необходимо настроить размеры окна

Это наша функция `setup()`. Объявляем глобальные переменные и переходим к виджетам

Обратите внимание на аргументы с ключевыми словами: их много в модуле `Tkinter`

Создаем окно верхнего уровня с помощью класса `Tk`, как и раньше

Сначала необходимо создать холст для рисования клеток. Конструктору передается много аргументов, таких как ширина, высота, толщина рамки и цвет фона

Это знакомая кнопка Start, но нужна еще кнопка Clear

Создаем список выбора, о котором поговорим позже

Это виджет списка выбора, который был задуман изначально. Сам объект `OptionMenu` создается так же, как и другие виджеты, но здесь есть ряд важных аргументов, которые мы вскоре обсудим

Помните: нам нужен менеджер компоновки для позиционирования виджетов в окне, поэтому для каждого виджета вызывается метод `pack()`

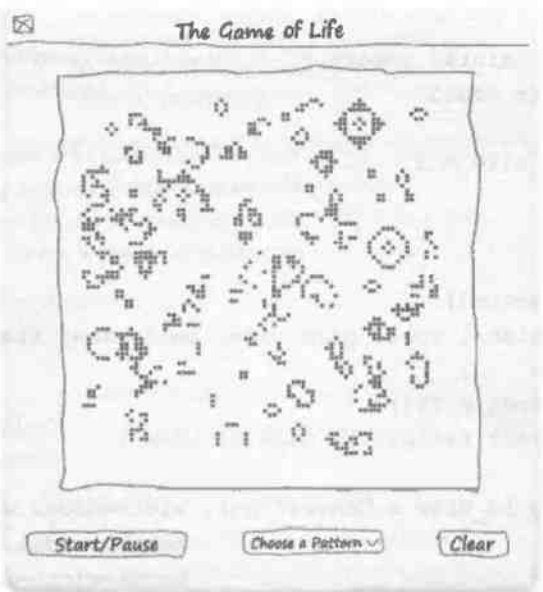
И не забудьте вызвать функцию `setup()`

Сохраните приведенный на предыдущей странице код в файле *view.py* и протестируйте его.

Вот наше окно. Помните, что его вид зависит от операционной системы

Как видите, менеджер компоновки не лучшим образом справился с задачей: нам не нужно, чтобы кнопки располагались в столбик

Для сравнения: вот наш план

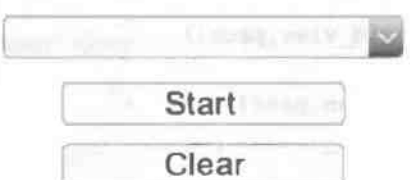


Мы хотим, чтобы кнопки располагались вот так. Граница вокруг холста нас не интересует, так как игровое поле будет хорошо смотреться и без нее

Исправление макета

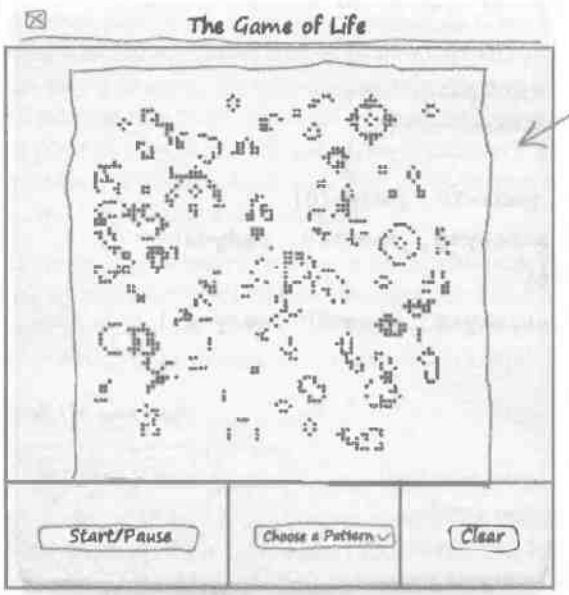
Менеджер компоновки модуля `Tkinter`, размещающий виджеты в основном окне по своему усмотрению, не позволяет получить предусмотренный нами макет графического интерфейса. Можно было бы потратить время на его настройку, но оказывается, что в модуле `Tkinter` есть несколько менеджеров компоновки, и для наших целей лучше всего подходит *менеджер компоновки по сетке*. Несмотря на название, он не имеет никакого отношения к решетке нашего симулятора. Сетка, которой управляет такой менеджер, применяется для позиционирования виджетов в основном окне. Это очень простая задача, если заранее продумать расположение элементов интерфейса.

Расположение в столбик — не то, что нам нужно



Размещение виджетов по сетке

Давайте еще раз изучим проект графического интерфейса и представим, будто все виджеты расположены на невидимой сетке.



Считайте это строкой 0, которая занимает три столбца и содержит виджет холста

Наверное, не стоит напоминать о том, что в программировании счет начинается с 0

Это строка 1, содержащая три столбца, в каждом из которых находится виджет

Кнопка Start: строка 1, столбец 0

Виджет списка: строка 1, столбец 1

Кнопка Clear: строка 1, столбец 2

Кодирование макета

Определившись с расположением каждого виджета, нужно передать менеджеру компоновки их координаты на сетке.

Холст `grid_view` добавляется в строку 0 сетки и занимает три столбца. Вокруг него создаются отступы для наглядности

```
grid_view.grid(row=0, columnspan=3, padx=20, pady=20)
start_button.grid(row=1, column=0, sticky=W, padx=20, pady=20)
option.grid(row=1, column=1, padx=20)
clear_button.grid(row=1, column=2, sticky=E, padx=20, pady=20)
```

Кнопки и список добавляются в соответствующие столбцы строки 1, тоже с отступами

Параметр `sticky` сообщает менеджеру компоновки о том, что вместо центрирования необходимо закреплять кнопки у левого (W) и правого (E) края окна. Это позволяет кнопкам сохранять свое расположение при масштабировании окна

Откройте файл *view.py* и замените в нем обращения к менеджеру компоновки, заданному по умолчанию, обращениями к менеджеру компоновки по сетке. Протестируйте программу и убедитесь в том, что пользовательский интерфейс стал выглядеть лучше.

```
grid_view.pack()
start_button.pack()
clear_button.pack()
option.pack()
```

Найдите этот код в файле view.py и замените следующим кодом

```
grid_view.grid(row=0, columnspan=3, padx=20, pady=20)
start_button.grid(row=1, column=0, sticky=W, padx=20, pady=20)
option.grid(row=1, column=1, padx=20)
clear_button.grid(row=1, column=2, sticky=E, padx=20, pady=20)
```

Так намного лучше! Еще нет игрового поля с клетками, но остальное уже смотрится так, как нам нужно

Не бойтесь задавать вопросы

В: Почему окно верхнего уровня называют корневым? По аналогии с корнем дерева?

О: Это устоявшийся термин, и аналогия с деревом тут не случайна. В корневой системе дерева есть большой главный корень, от которого ветвятся меньшего размера боковые корни. Похожим образом к большому окну верхнего уровня прикрепляются компоненты поменьше. Например, виджет **OptionMenu** включает раскрывающееся меню, которое, в свою очередь, содержит набор опций, а каждая из них снабжена подписью. Аналогии с корнями дерева часто встречаются в компьютерных науках: корень файловой системы, корень иерархической структуры и т.п.

В: Почему наши кнопки не выполняют никаких действий, когда на них щелкают?

О: Потому что мы еще не указали им, что они должны делать. Сейчас мы этим займемся!

Переходим к написанию контроллера

Вот мы и добрались до создания контроллера. Мы подготовили модель данных для хранения клеток и расчета их новых поколений, а также создали графический интерфейс в полном соответствии с бумажным прототипом. Осталось объединить обе части приложения, заставив интерфейс выполнять полагающиеся ему функции. Другими словами, нужно написать управляющий код симулятора, для чего нам придется по-иному взглянуть на процесс вычислений. Не волнуйтесь, все это не настолько сложно для понимания, как рекурсия, просто не похоже на то, чем мы занимались раньше.

Для начала следует объединить модель с представлением. Это первый шаг на пути к построению полноценного симулятора. Далее мы начнем писать код контроллера, последовательно реализуя функциональность каждого элемента интерфейса.

Выдохни

Это длинная глава, в которой много кода и терминов. Сделайте перерыв, поспите и дайте мозгу усвоить новый материал. Отдохнуть никогда не помешает!

Функция update()

Чтобы можно было объединить представление с моделью, мы напишем функцию `update()`, которая будет отвечать за вызов функции `next_gen()` модели и прорисовку клеток на экране, точнее, на виджете холста.

Сначала составим псевдокод функции `update()`.

```
DEF update():                                   ← Вычисляем новое поколение
    CALL next_gen()
    FOR переменная i in range(0, height):       ← Проходим каждую клетку модели
        FOR переменная j in range(0, width):
            cell = model.grid_model[i][j]
            IF cell == 1:                       ← Если клетка живая, закрашиваем соответствующий пиксель представления
                DRAW черная клетка в точке (i, j) решетки grid_view
```

Единственное отличие псевдокода от реального кода состоит в том, что мы будем рисовать на холсте не пиксели, а небольшие квадратики размером 5×5 пикселей. Почему? Одиночные пиксели слишком сложно разглядеть на экране, нужно что-то покрупнее. Если помните, мы создали для этого глобальную переменную `cell_size`. Далее вы увидите, как она применяется. И еще: мы будем рисовать только живые клетки. Таким образом, чтобы избавиться от клеток, которые умерли в предыдущем поколении, нужно будет сначала полностью очистить игровую область, а затем перерисовать на ней клетки текущего поколения. Теперь перейдем к программному коду.

Не бойтесь задавать вопросы

В: Не будут ли конфликтовать имена задаваемых мною переменных, функций и классов с содержимым модуля `Tkinter` при его импорте с помощью инструкции `from tkinter import *`?

О: Инструкция `from...import` подключает переменные, функции и классы модуля так, что их имена не нужно предварять именем модуля. Это позволяет сократить объем вводимого кода. С другой стороны, возникает риск конфликтов имен с пользовательскими переменными, функциями и классами. Зачем же рисковать? Это имеет смысл делать, если вы точно знаете, что не собираетесь использовать в коде имена, которые будут конфликтовать с содержимым модуля (в случае модуля `Tkinter` речь идет о таких именах, как `Button`, `Tk` и другие названия виджетов). Тогда текст программы можно сделать более коротким и понятным.

Сила мысли

Как вы думаете, почему мы отображаем в игровой области только живые клетки, а не живые и мертвые?

```
def update():
    global grid_view

    grid_view.delete(ALL)
    model.next_gen()

    for i in range(0, model.height):
        for j in range(0, model.width):
            if model.grid_model[i][j] == 1:
                draw_cell(i, j, 'black')
```

Удаляем все, что есть на холсте, с помощью метода `delete()` объекта `Canvas`

Вычисляем следующее поколение клеток

Следуем псевдокоду...

Если клетка в позиции (i, j) живая, рисуем небольшой черный прямоугольник

Это наша функция, а не модуля `tkinter`, поэтому ее еще нужно написать

На этом функция `update()` завершена, но нужно еще написать функцию `draw_cell()`, отвечающую за прорисовку клеток прямоугольной формы.

Функция `draw_cell()` получает номер строки и столбца, а также цвет

Это чисто эстетические настройки

```
def draw_cell(row, col, color):
    global grid_view, cell_size

    if color == 'black':
        outline = 'grey'
    else:
        outline = 'white'

    grid_view.create_rectangle(row*cell_size,
                               col*cell_size,
                               row*cell_size+cell_size,
                               col*cell_size+cell_size,
                               fill=color, outline=outline)
```

Здесь выбирается цвет контура прямоугольников. В случае черных прямоугольников он серый

Координаты левого верхнего угла прямоугольника

Координаты правого нижнего угла прямоугольника

Функция рисует небольшой прямоугольник, представляющий клетку

Указываем цвет заливки прямоугольника и цвет его контура

Функция `draw_cell()` является вспомогательной. В ней используется метод `create_rectangle()` объекта холста для рисования прямоугольника клетки. Аргументы этого метода задают координаты левого верхнего и правого нижнего углов прямоугольника, вычисляемые с помощью переменной `cell_size`.

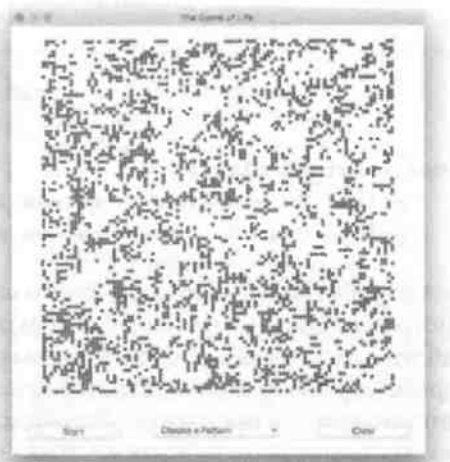
Код симулятора становится все более интересным. Откройте файл *view.py* и добавьте в него функции `update()` и `draw_cell()`. Также не забудьте включить в него вызов функции `update()`, чтобы протестировать работу программы.

```
def update():
    global grid_view

    grid_view.delete(ALL)

    model.next_gen()
    for i in range(0, model.height):
        for j in range(0, model.width):
            if model.grid_model[i][j] == 1:
                draw_cell(i, j, 'black')


def draw_cell(row, col, color):
    global grid_view, cell_size

    if color == 'black':
        outline = 'grey'
    else:
        outline = 'white'

    grid_view.create_rectangle(row*cell_size,
                               col*cell_size,
                               row*cell_size+cell_size,
                               col*cell_size+cell_size,
                               fill=color, outline=outline)

if __name__ == '__main__':
    setup()
    update()
    mainloop()
```

Поместите функцию `update()` после функции `setup()`, непосредственно перед проверкой переменной `__name__`

Вставьте вызов функции `update()`

Вот что мы получили. Уже лучше!



Новый способ вычислений

Написанный нами симулятор напоминает автомобиль в том смысле, что мы, как водители, задаем направление движения. В любой момент времени программа знает, что следует делать дальше. Однако во многих графических приложениях это совершенно не так. В них применяется модель *реактивного программирования*.

Чтобы понять его суть, представьте, что вам нужно настроить работу кнопки Start, которая просто отображается на экране. Внезапно пользователь щелкает на ней. Что дальше? Очевидно, что в ответ должна быть запущена какая-то функция, отвечающая за выполнение определенных операций. Иными словами, в приложении нужно предусмотреть *реакцию* на возникающие *события*. Чаще всего такими событиями будут щелчки на кнопках, выбор опций меню, ввод текста в поле и т.п. Но это могут быть и менее очевидные вещи, например срабатывание таймера, получение данных по сети и еще много чего. Подобная модель работы приложения называется *событийно-ориентированным программированием*.

```
def start_handler():
    print("Вы щелкнули на кнопке Start.")
    print("Спасибо, что обратили на меня внимание!")
```

Код обработки события, подобного щелчку на кнопке, в разных языках программирования называется по-разному: обработчик события, наблюдатель, функция обратного вызова и т.п. Мы будем использовать термин *обработчик события*, поскольку пишем код реакции на событие, происходящее в программе. Впрочем, независимо от терминологии механизм всегда один и тот же. Вы сообщаете объекту, который генерирует события, например кнопке, имя функции, вызываемой при наступлении события. Кроме того, обработчику события обычно передается специальный объект события. Но об этом мы поговорим позже, а пока рассмотрим, как организовать вызов обработчика.

Обработка щелчков мыши

Код контроллера будет не слишком большим, поэтому мы добавим его в файл *view.py* (вместо того чтобы создавать отдельный файл *controller.py*), снова откройте файл *view.py* и добавьте следующую строку сразу же после инструкции создания кнопки Start.

```
start_button = Button(root, text='Start', width=12)
start_button.bind('<Button-1>', start_handler)
```

Мы хотим, чтобы при щелчке на кнопке вызывалась функция `start_handler()`

Интересующее нас событие — это щелчок левой кнопкой мыши

Метод `bind()` можно вызвать для любого виджета, чтобы связать событие с функцией-обработчиком

Добавьте новую функцию `start_handler()` между функциями `setup()` и `update()`.

Обработчикам передается объект события, содержащий информацию о событии, в частности, о том, на какой кнопке был выполнен щелчок. В данном случае нам не нужна эта информация, но позже она понадобится

Эта функция вызывается при щелчке на кнопке Start

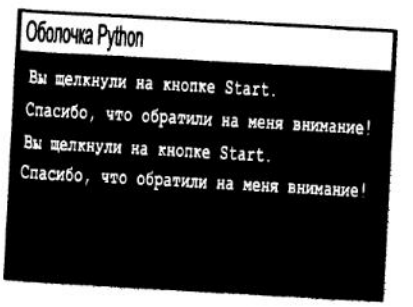
```
def start_handler(event):
    print("Вы щелкнули на кнопке Start.")
    print("Спасибо, что обратили на меня внимание!")
```

Тест-драйв

Внесите указанные изменения в файл *view.py* и протестируйте полученный результат. Щелкните на кнопке **Start** и посмотрите, что будет выведено в окне оболочки Python.

Мы щелкнули на кнопке Start несколько раз

Каждый раз при щелчке вызывается функция `start_handler()`



Все верно. В событийно-ориентированном программировании для каждого ожидаемого события, такого как щелчок на кнопке, выбор опции меню, щелчок в игровой области для добавления живых клеток (нас это тоже ожидает), пишется специальная функция-обработчик, которая будет вызываться при наступлении именно этого события. Со временем вы найдете такой стиль программирования вполне естественным.

Может показаться удивительным, что программа все время находится в ожидании событий, не выполняя никакого кода. Почему она не завершается? Или "за кулисами" все же выполняется какой-то код в процессе ожидания? Вот тут нам пора вспомнить о функции `mainloop()`. В нашей программе, как и в примере с черепашками, мы всякий раз вызываем данную функцию в последней строке кода. Именно она занимается мониторингом событий, связанных с пользовательским интерфейсом. И как только она обнаруживает, что пользователь осуществил какое-то действие, она вызывает соответствующий обработчик. Таким образом, в программе все время выполняется какой-то код, в данном случае код функции `mainloop()`.

СИЛА МЫСЛИ

Ниже приведен вызов, с которым вам еще не доводилось встречаться. Как думаете, что он делает? Подсказка: здесь также вызывается обработчик события.

Это тот самый объект Tk

Это время в миллисекундах (1 секунда = 1000 мс)

```
root.after(100, update)
```

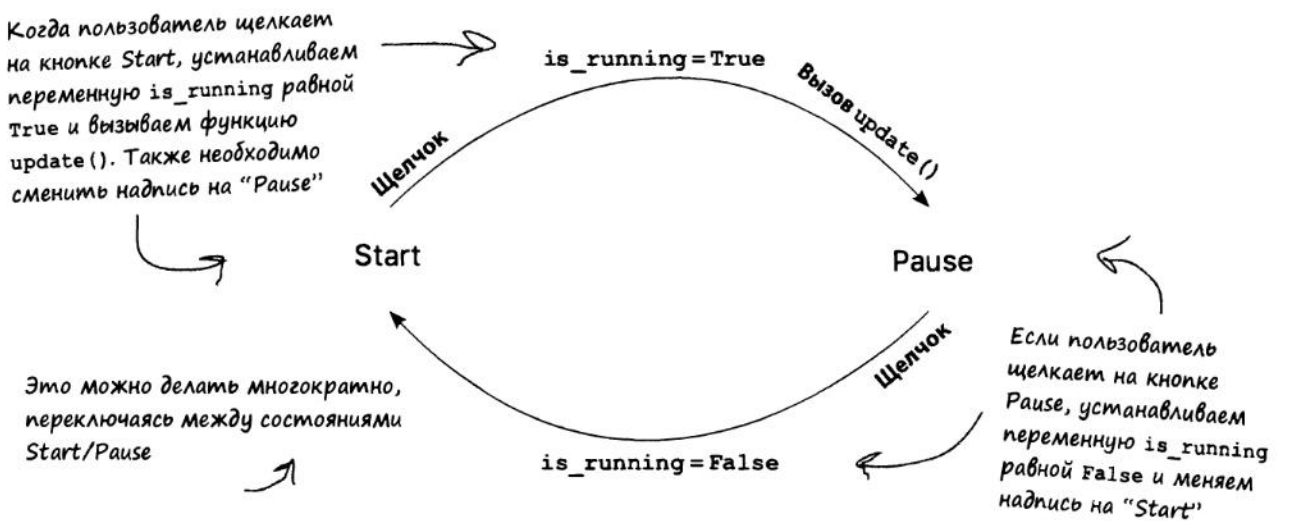
Это метод объекта Tk

Это функция update(), которую мы уже написали

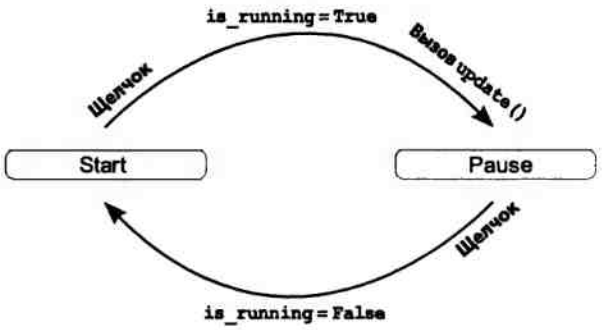
Кажется, что-то похожее было в программе, которая отслеживала положение МКС в главе 10

Обработка событий, связанных с кнопкой Start/Pause

При щелчке на кнопке **Start** симулятор должен начать вычислять новые поколения клеток. Если вы не забыли бумажный прототип пользовательского интерфейса, то знаете, что одновременно кнопка **Start** должна превращаться в кнопку **Pause**. Чтобы не запутаться, давайте нарисуем *диаграмму состояний*, описывающую поведение кнопки **Start**. Согласно диаграмме нам понадобится новая глобальная переменная `is_running`, которая становится равна `True` при щелчке на кнопке **Start**, что означает вычисление следующих поколений клеток. Если же симулятор не запущен или поставлен на паузу, то переменная `is_running` будет равна `False`.



Реализация кнопки Start/Pause



Сначала необходимо создать и инициализировать глобальную переменную `is_running`. Добавьте выделенную ниже инструкцию в начало файла *view.py*.

```
from tkinter import *
import model

cell_size = 5
is_running = False
```

Инициализируем глобальную переменную значением `False`

Теперь нужно реализовать логику, описываемую диаграммой состояний. Внесите следующие изменения в функцию `start_handler()`.

```
def start_handler(event):
    print("Вы щелкнули на кнопке Start.")
    print("Спасибо, что обратили на меня внимание!")
    global is_running, start_button

    if is_running:
        is_running = False
        start_button.configure(text='Start')
    else:
        is_running = True
        start_button.configure(text='Pause')
        update()
```

Избавимся от старого кода

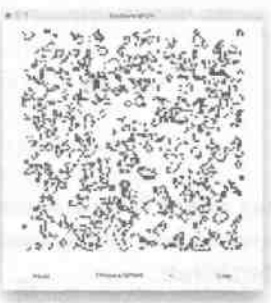
Если симулятор уже запущен (на кнопке отображается надпись "Pause"), задаем переменную `is_running` равной `False` и меняем надпись на "Start"

В противном случае задаем переменную `is_running` равной `True`, меняем текст на "Pause" и вызываем функцию `update()`, чтобы вычислить очередное поколение клеток

Тест-драйв

Обновите файл *view.py*, внеся в него указанные изменения. Запустите программу и пощелкайте на кнопках **Start** и **Pause**. Вычисляются ли при этом новые поколения клеток?

Убедитесь в том, что кнопка Start переключается между состояниями Start и Pause



При многократных щелчках на кнопке Start/Pause вы должны увидеть, как вычисляются новые поколения клеток

Пример того, как небольшой фрагмент кода вызывает огромные изменения. Но это еще не все!

Другой тип событий

Если щелкать на кнопках достаточно быстро, то можно увидеть, как функция `next_gen()` генерирует новые поколения клеток. Это неплохо, но пальцы быстро устанут, а кроме того, компьютер способен выполнять расчеты быстрее, чем мы способны щелкать на кнопках. Для автоматизации вычислений нам нужно обратиться к иному типу событий, связанных не с действиями пользователя (такими, как щелчок на кнопке), а с хронометражом.

У объекта `Tk` имеется интересный метод `after()`, с которым стоит познакомиться поближе.

Подобные методы есть в большинстве языков программирования

В секунде 1000 миллисекунд

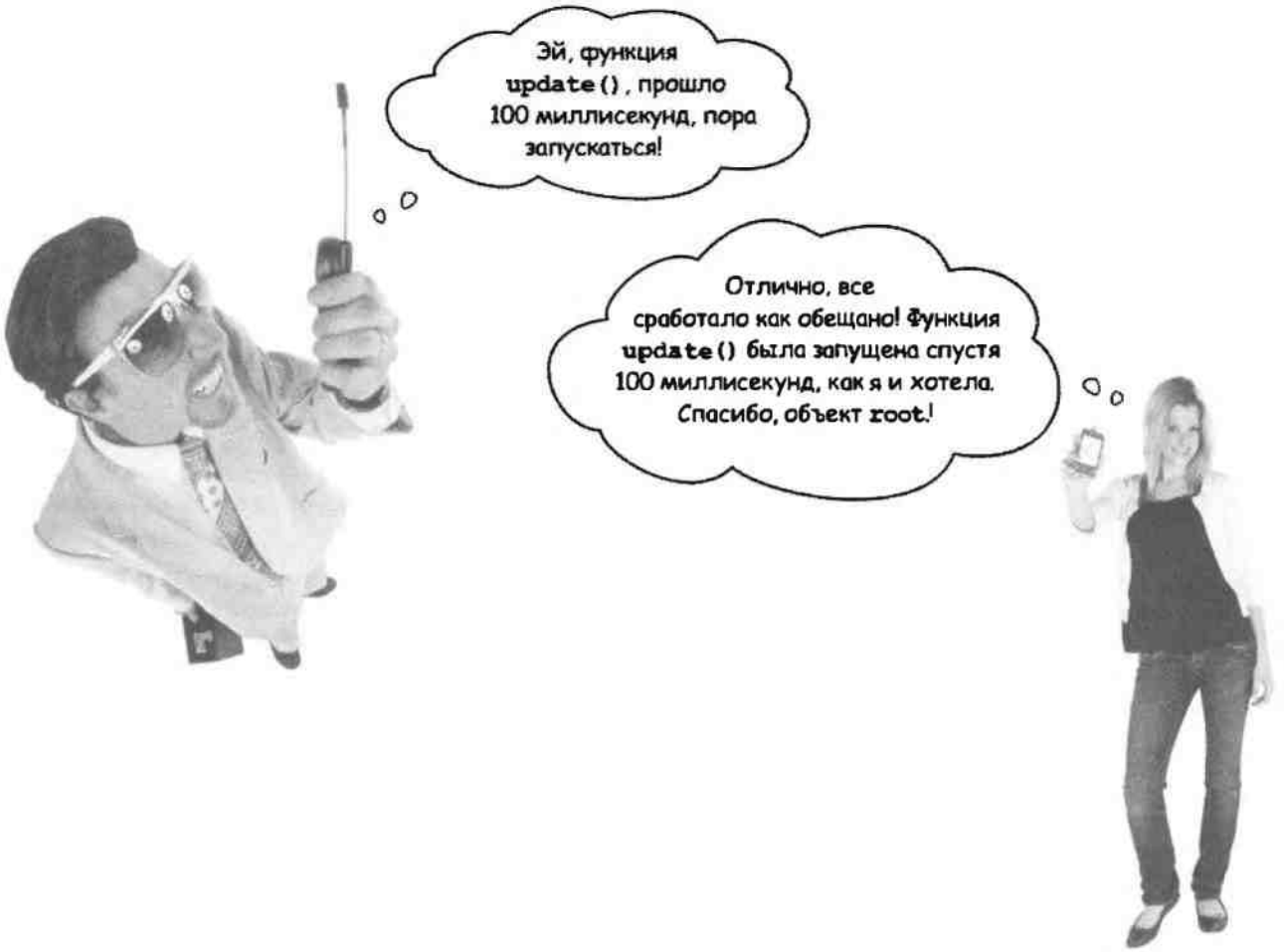
У объекта `Tk` есть удобный метод `after()`

Первый аргумент — это время в миллисекундах

Второй аргумент — это функция, вызываемая по прошествии заданного времени

```
root.after(100, update)
```

Что же именно делает метод `after()`? Чтобы понять это, необходимо рассмотреть код, в котором он вызывается.



Возьмите карандаш

Изучите приведенный ниже код. Что, по-вашему, он выводит? Как работает эта программа? Сверьтесь с ответом, приведенным в конце главы.

Тренируем мозг
Как думаете, это рекурсивная функция?

```
from tkinter import *

root = Tk()
count = 10

def countdown():
    global root, count


    if count > 0:
        print(count)
        count = count - 1
        root.after(1000, countdown)
    else:
        print('Взлет!')

countdown()
mainloop()
```

А как же метод after()?

С помощью метода `after()` вычисления в симуляторе можно выполнять через регулярные промежутки времени. Надеемся, вы догадались об этом, выполняя предыдущее упражнение. Воспользуемся аналогичной методикой для периодического запуска метода `update()`.

```
def update():
    global grid_view, root, is_running

    grid_view.delete(ALL)
    model.next_gen()

    for i in range(0, model.height):
        for j in range(0, model.width):
            if model.grid_model[i][j] == 1:
                draw_cell(i, j, 'black')
    if (is_running):
        root.after(100, update)
```

Необходимо добавить глобальные переменные `root` и `is_running`, так как они нам понадобятся

Теперь, когда вызывается функция `update()`, если переменная `is_running` равна `True`, планируется следующий вызов функции `update()` спустя 100 мс (т.е. 1/10 с)

Тот случай, когда все решает *единственная строка* кода. Внесите изменения в файл *view.py* и приготовьтесь увидеть, как программа превращается в полноценный симулятор игры "Жизнь".

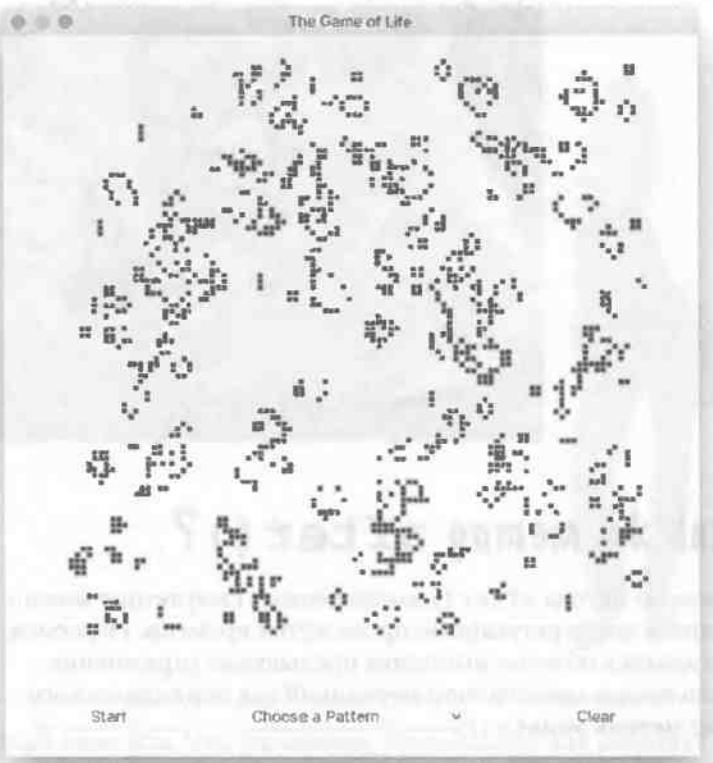
(1) Щелкните на кнопке Start и наблюдайте за моделированием

(2) Щелкните на кнопке Pause, чтобы приостановить моделирование

(3) Повторите процесс несколько раз и перезапустите симулятор, чтобы получить новый случайный набор начальных клеток

Это уже не выглядит как случайный набор клеток

Заметили повторяющиеся шаблоны? Планеры, парящие по экрану? Области хаоса, которые то появляются, то исчезают?



Это уже конец?

Мы проделали большую работу, реализовав основной код симулятора. Осталось немного доработать графический интерфейс. Прежде всего, нужно реализовать код кнопки Clear. Щелчок на ней должен приводить к очистке игрового поля от клеток – все они становятся мертвыми. Когда экран пуст, мы хотим иметь возможность щелкать на игровом поле для добавления живых клеток. И наконец, у нас есть меню, в котором можно загрузить готовый шаблон размещения клеток. Будем действовать последовательно, начав с кнопки Clear.

Что должна делать кнопка **Clear**? При щелчке на ней переменная `is_running` должна стать равной `False`, а все клетки должны обнулиться. Кроме того, на кнопке слева должна снова появиться надпись "Start". Наконец, должна быть вызвана функция `update()`, которая обновит игровое поле (когда все клетки обнулены, поле будет пустым).

Используя код обработчика событий для кнопки **Start** в качестве образца, напишите код управления кнопкой **Clear**.

```
start_button.bind('<Button-1>', start_handler)

def start_handler(event):
   global is_running, start_button

   if is_running:
       is_running = False
       start_button.configure(text='Start')
   else:
       is_running = True
       start_button.configure(text='Pause')
       update()
```

Необходимо сообщить кнопке Clear о наличии обработчика, как и в случае кнопки Start

Это обработчик `start_handler()`, который послужит примером

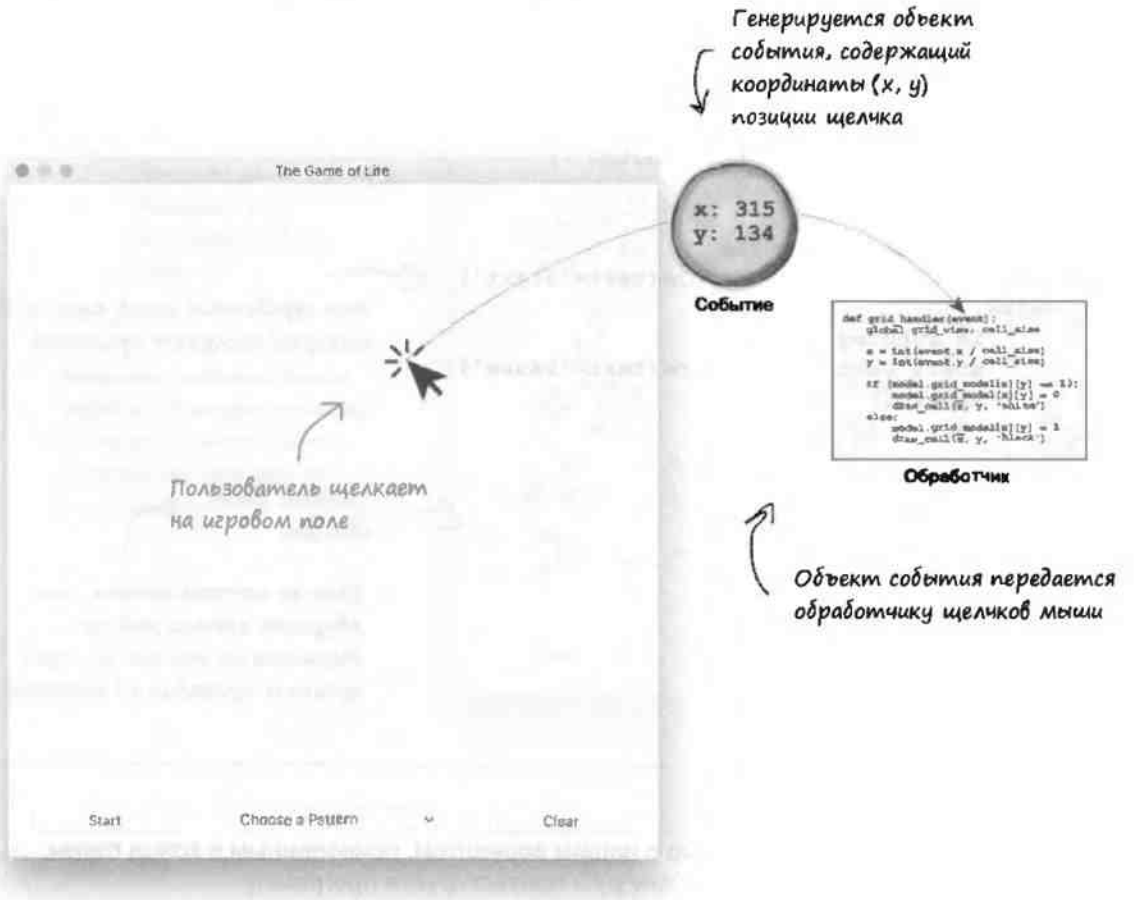
Введите здесь свой код

Если не можете понять, как обнулить клетки модели, взгляните на то, как функция `update()` проходит по клеткам

Когда напишете код, сравните его с нашим вариантом, приведенным в конце главы, после чего добавьте его в файл *view.py* и протестируйте программу.

Непосредственное редактирование клеток

На данный момент игра начинается с произвольного выбора живых клеток на игровом поле. Затем по щелчку на кнопке Start запускается расчет следующих поколений. Но мы бы хотели иметь возможность до запуска моделирования вводить живые клетки, щелкая прямо на решетке. Для этого мы воспользуемся методикой, аналогичной той, которая применяется при обработке щелчков на кнопке. Другими словами, всякий раз, когда пользователь щелкает на игровом поле, будет запускаться обработчик события, добавляющий в выбранной ячейке живую клетку как на экране, так и в модели данных.



Мы начнем с привязки обработчика щелчков левой кнопки мыши к игровой области (холсту), как это делалось для кнопок Start и Clear:

```
grid_view.bind('<Button-1>', grid_handler)
```

Когда пользователь щелкает левой кнопкой мыши на холсте `grid_view`, *вызывается функция* `grid_handler()`

Обработчик grid_handler()

Нужно понять, что должна делать функция grid_handler(). Как насчет такого поведения: если пользователь щелкает на белой (мертвой) клетке, она должна окрашиваться в черный цвет и становиться живой? Но если клетка уже черная, она становится мертвой и меняет свой цвет на белый. При этом клетка меняется не только в представлении, но и в модели данных.

Рассмотрим программный код.

Функция grid_handler() получает объект события в качестве аргумента

Координаты (x, y) позиции щелчка можно извлечь из объекта события. Для этого есть соответствующие атрибуты

```
def grid_handler(event):
    global grid_view, cell_size

    x = int(event.x / cell_size)
    y = int(event.y / cell_size)

    if (model.grid_model[x][y] == 1):
        model.grid_model[x][y] = 0
        draw_cell(x, y, 'white')
    else:
        model.grid_model[x][y] = 1
        draw_cell(x, y, 'black')
```

Не забывайте о том, что решетка масштабируется о размеру клетки. Поэтому для определения истинных координат (x, y) (т.е. номера строки и столбца игрового оля) необходимо разделить значения атрибутов на cell_size. Функция int() позволяет получить результат в виде целого числа, а не числа с плавающей точкой

Если текущая клетка в модели равна 1, устанавливаем ее равной 0 и вызываем функцию draw_cell(), чтобы окрасить клетку в белый цвет

В противном случае устанавливаем клетку равной 1 и вызываем функцию draw_cell(), чтобы окрасить клетку в черный цвет

Тест-драйв

Добавьте функцию grid_handler() в файл *view.py* сразу же после функции setup(). И не забудьте вставить вызов метода bind(), как показано ниже. *Наконец, пора удалить вызов функции randomize() из файла model.py, чтобы начинать с пустой решетки.*

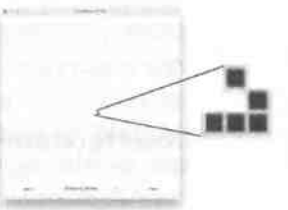
Не пропустите этот этап

```
grid_view.grid(row=0, columnspan=3, padx=20, pady=20)
grid_view.bind('<Button-1>', grid_handler)
```

Добавьте вызов функции bind() сразу после создания решетки grid_view

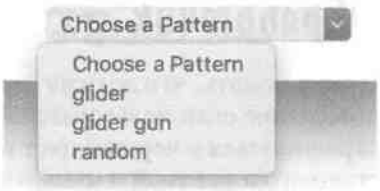
Теперь при запуске симулятора вы должны увидеть пустое поле. Пощелкайте на поле для добавления живых клеток, а затем щелкните на кнопке Start

Попробуйте нарисовать планер и щелкнуть на кнопке Start. Если промахнетесь, просто щелкните повторно, чтобы стереть ячейку



Если щелчки не работают, убедитесь в том, что вы щелкаете посредине окна

Добавление шаблонов



Мы хотим снабдить симулятор еще одной дополнительной функцией, создав меню, из которого пользователь сможет выбрать готовый шаблон. По нашей задумке шаблонов будет три, но никто не мешает вам добавить и свои собственные шаблоны.

Мы уже создали в программе экземпляр меню, однако намеренно не обсуждали соответствующий код, поскольку виджет меню работает немного не так, как виджеты кнопки и холста. Давайте вспомним, как выглядит этот код.

В модуле `Tkinter` имеется объект для хранения значений. В данном случае мы создаем объект, хранящий строку, и записываем его в переменную `choice`

Устанавливаем значение объекта `choice` равным `'Choose a Pattern'`. Это начальный вариант выбора в виджете

```
choice = StringVar(root)
choice.set('Choose a Pattern')
option = OptionMenu(root, choice, "Choose a Pattern",
                                  "glider",
                                  "glider gun",
                                  "random")

option.config(width=20)
```

Далее создаем список выбора и передаем ему объект нашего основного окна (как это всегда бывает в случае с виджетами), переменную для хранения строки и набор вариантов, которые появятся в списке

Для улучшения внешнего вида растягиваем виджет до ширины 20

С точки зрения интерфейса меню готово, но пока что оно ничего не делает. Как и в случае кнопок, необходимо связать с ним обработчик событий, вот только способ привязки обработчика отличается. Соответствующий код выглядит следующим образом.

В конструктор `OptionMenu` добавляется аргумент `command`, который задает обработчик, вызываемый при выборе элемента списка

```
choice = StringVar(root)
choice.set('Choose a Pattern')
option = OptionMenu(root, choice, "Choose a Pattern",
                                  "glider",
                                  "glider gun",
                                  "random",
                        command=option_handler)
option.config(width=20)
```

УПРАЖНЕНИЕ

Не забудьте добавить этот фрагмент кода в файл *view.py*!

Обработчик событий для объекта OptionMenu

Сам обработчик тоже пишется по-другому, ведь теперь мы реагируем не на банальный щелчок на кнопке, а на выбор пользователем элемента списка. Необходимо понять, какой пункт меню выбран, и предпринять соответствующие действия.

Как вы уже поняли, первый пункт меню – Choose a Pattern (Выбрать шаблон) – служит подсказкой для пользователя и не является командой. Для остальных вариантов выбора нужно написать код обработки.

Этот обработчик тоже получает объект события, хотя он нам не понадобится

Мы будем использовать рассмотренный ранее объект StringVar модуля Tkinter

```
def option_handler(event):
    global is_running, start_button, choice

    is_running = False
    start_button.configure(text='Start')

    selection = choice.get()

    if selection == 'glider':
        model.load_pattern(model.glider_pattern, 10, 10)

    elif selection == 'glider gun':
        model.load_pattern(model.glider_gun_pattern, 10, 10)

    elif selection == 'random':
        model.randomize(model.grid_model, model.width, model.height)

    update()
```

Останавливаем процесс моделирования и сбрасываем состояние кнопки Start

Переменная choice хранит объект StringVar со значением, которое пользователь выбрал в списке. У объекта StringVar есть метод get()

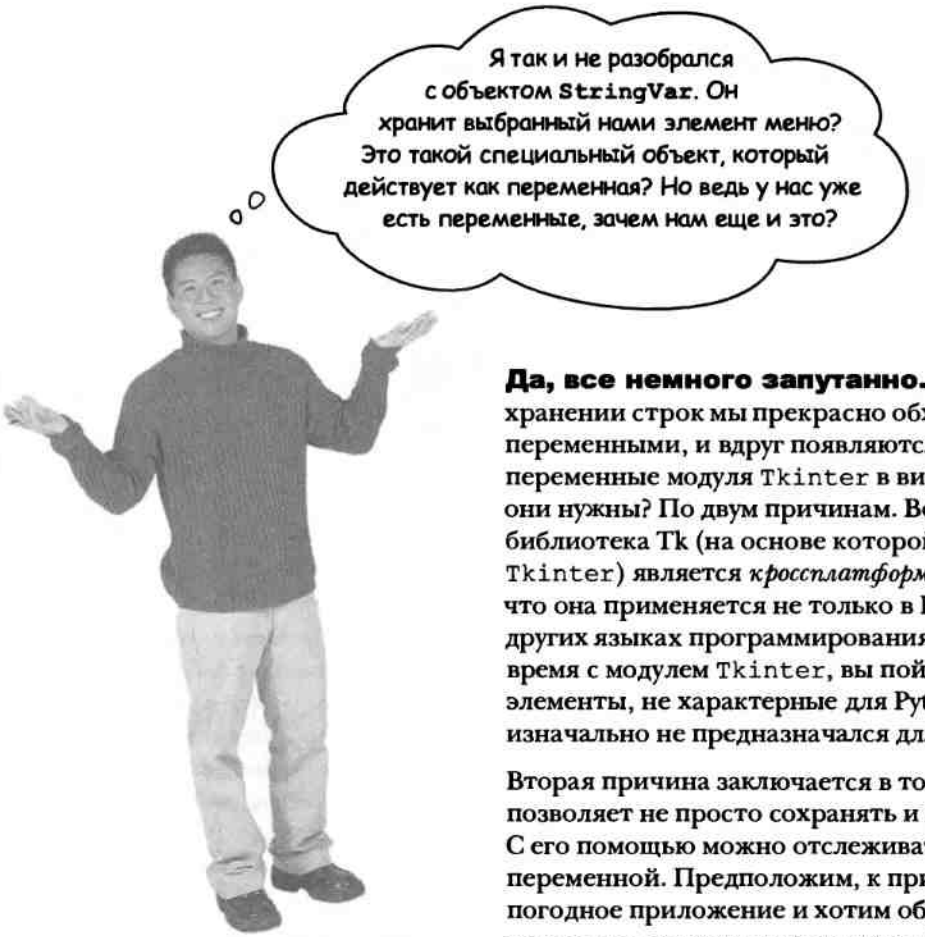
Проверяем, какое значение выбрал пользователь, и либо загружаем соответствующую фигуру, либо вызываем функцию randomize()

Учтите, что нам еще необходимо написать функцию load_pattern(), а также определить фигуры

Если пользователь выбирает вариант 'random', то у нас на этот случай уже есть готовая функция randomize(), нужно просто ее вызвать

После изменения модели необходимо обновить игровое поле

Добавьте приведенный выше код в файл *view.py* перед функцией `start_handler()`. Мы не можем его пока что запустить, поскольку еще не написали метод `load_pattern()` и не определили готовые фигуры. Когда все будет готово, мы проведем финальный тест-драйв.



Да, все немного запутанно. Раньше при хранении строк мы прекрасно обходились обычными переменными, и вдруг появляются собственные переменные модуля `Tkinter` в виде объектов. Зачем они нужны? По двум причинам. Во-первых, графическая библиотека Tk (на основе которой написан модуль `Tkinter`) является *кроссплатформенной*. Это означает, что она применяется не только в Python, но и во многих других языках программирования. Поработав какое-то время с модулем `Tkinter`, вы поймете, что в нем есть элементы, не характерные для Python, поскольку он изначально не предназначался для Python.

Вторая причина заключается в том, что класс `StringVar` позволяет не просто сохранять и считывать значения. С его помощью можно отслеживать изменение переменной. Предположим, к примеру, что мы создаем погодное приложение и хотим обновлять экран при каждом изменении температуры. Используя метод `trace()` объекта `StringVar`, мы будем получать уведомление о каждом таком событии.

```
temperature = StringVar()
temperature.trace("w", my_handler)
```

При каждом изменении температуры (переменная `temperature`) вызывается функция `my_handler()`

В нашем симуляторе мы не будем задействовать расширенные возможности объекта `StringVar`, но вы должны знать о них, поскольку они являются важным элементом событийно-ориентированного программирования.

Создание собственных фигур

Давайте определим несколько готовых фигур в виде двухмерных списков. В частности, шаблон планера (`glider`) представляется следующим образом.

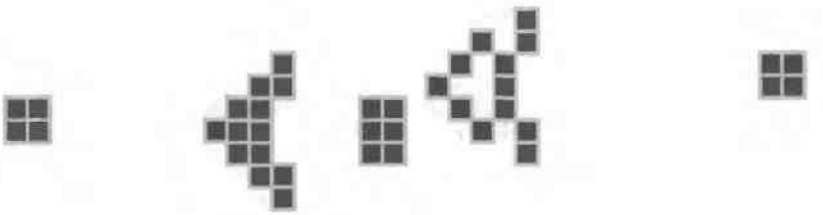
```
glider_pattern = [[0, 0, 0, 0, 0],
                  [0, 0, 1, 0, 0],
                  [0, 0, 0, 1, 0],
                  [0, 1, 1, 1, 0],
                  [0, 0, 0, 0, 0]]
```

Замечаете сходство?

Шаблон *планерного ружья* (`glider gun`) описывается намного сложнее.

```
glider_gun_pattern =  [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0],
                       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0],
                       [0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                       [0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

А здесь?



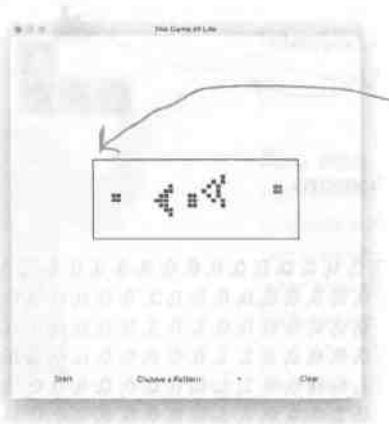
УПРАЖНЕНИЕ

Вам не нужно вводить этот код вручную. Откройте папку с материалами к главе 11, найдите в ней файлы *glider.py* и *glider_gun.py* и скопируйте их содержимое в конец файла *model.py* (сразу после функции `count_neighbors()`). Тестировать пока нечего, просто убедитесь в отсутствии синтаксических ошибок.

Загрузчик шаблонов

Теперь напишем код, который будет загружать выбранную фигуру на игровое поле. Все, что должен делать загрузчик, — получать двухмерный список и копировать его содержимое (нули и единицы) в модель данных. Только предварительно следует очистить решетку, заполнив ее одними нулями.

Мы будем загружать фигуры, копируя их в список `grid_model`



Будут также поддерживаться дополнительные смещения по осям (x, y) для фигуры

Также необходимо позволить, чтобы фигура имела смещение, т.е., к примеру, располагалась по центру игрового поля. Для этого мы задаем величину смещения по осям x и y в виде аргументов загрузчика. Вот код загрузчика.

Помните, в главе 5 мы говорили о том, что у параметров могут быть значения по умолчанию?

Функция получает фигуру (такую, как планер или планерное ружье) в виде двухмерного списка

В функцию также передаются два других аргумента, задающих смещение по осям (x, y), что позволяет определять левую верхнюю клетку фигуры

Внимание: здесь имеет место дублирование кода (взгляните на обработчик `clear_handler()`). Стоит переписать код в будущем, чтобы устранить ненужную избыточность (считайте это домашним заданием)

```
def load_pattern(pattern, x_offset=0, y_offset=0):
    global grid_model

    for i in range(0, height):
        for j in range(0, width):
            grid_model[i][j] = 0

    j = y_offset

    for row in pattern:
        i = x_offset
        for value in row:
            grid_model[i][j] = value
            i = i + 1
        j = j + 1
```

Обнуляем все клетки

Задаем переменные `i` и `j` равными смещениям

Проходим по каждой клетке фигуры и назначаем ее соответствующей ячейке решетки `grid_model`

Добавьте в файл *model.py* приведенную на предыдущей странице функцию `load_pattern()`, расположив ее сразу после кода шаблонов. Теперь наконец-то можно приступать к полноценному тестированию симулятора игры "Жизнь"! На протяжении главы мы рассматривали отдельные его фрагменты, поэтому ниже приведен весь исходный код программы.

model.py

```
import random

height = 100
width = 100

def randomize(grid, width, height):
    for i in range(0, height):
        for j in range(0, width):
            grid[i][j] = random.randint(0, 1)


grid_model = [0] * height
next_grid_model = [0] * height
for i in range(height):
    grid_model[i] = [0] * width
    next_grid_model[i] = [1] * width

def next_gen():
    global grid_model, next_grid_model

    for i in range(0, height):
        for j in range(0, width):
            cell = 0
            count = count_neighbors(grid_model, i, j)


            if grid_model[i][j] == 0:
                if count == 3:
                    cell = 1
            elif grid_model[i][j] == 1:
                if count == 2 or count == 3:
                    cell = 1
            next_grid_model[i][j] = cell

    temp = grid_model
    grid_model = next_grid_model
    next_grid_model = temp
```

```
def count_neighbors(grid, row, col):

   count = 0
   if row-1 >= 0:
        count = count + grid[row-1][col]
   if (row-1 >= 0) and (col-1 >= 0):
       count = count + grid[row-1][col-1]
   if (row-1 >= 0) and (col+1 < width):
       count = count + grid[row-1][col+1]
   if col-1 >= 0:
       count = count + grid[row][col-1]
   if col + 1 < width:
       count = count + grid[row][col+1]
   if row + 1 < height:
       count = count + grid[row+1][col]
   if (row + 1 < height) and (col-1 >= 0):
       count = count + grid[row+1][col-1]
   if (row + 1 < height) and (col+1 < width):
       count = count + grid[row+1][col+1]
   return count

glider_pattern = [[0, 0, 0, 0, 0],
                  [0, 0, 1, 0, 0],
                  [0, 0, 0, 1, 0],
                  [0, 1, 1, 1, 0],
                  [0, 0, 0, 0, 0]]

glider_gun_pattern = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0],
              [0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

```
def load_pattern(pattern, x_offset=0, y_offset=0):
    global grid_model

    for i in range(0, height):
        for j in range(0, width):
            grid_model[i][j] = 0

    j = y_offset

    for row in pattern:
        i = x_offset
        for value in row:
            grid_model[i][j] = value
            i = i + 1
        j = j + 1

if __name__ == '__main__':
    next_gen()
```

view.py

```
from tkinter import *
import model

cell_size = 5
is_running = False


def setup():
    global root, grid_view, cell_size, start_button, clear_button, choice

    root = Tk()
    root.title('The Game of Life')

    grid_view = Canvas(root, width=model.width*cell_size,
                             height=model.height*cell_size,
                             borderwidth=0,
                             highlightthickness=0,
                             bg='white')

    start_button = Button(root, text='Start', width=12)
    clear_button = Button(root, text='Clear', width=12)

    choice = StringVar(root)
    choice.set('Choose a Pattern')
    option = OptionMenu(root, choice, 'Choose a Pattern', 'glider', 'glider gun', 'random',
                        command=option_handler)
    option.config(width=20)
```

```
 grid_view.grid(row=0, columnspan=3, padx=20, pady=20)
    grid_view.bind('<Button-1>', grid_handler)
    start_button.grid(row=1, column=0, sticky=W,padx=20, pady=20)
    start_button.bind('<Button-1>', start_handler)
    option.grid(row=1, column=1, padx=20)
    clear_button.grid(row=1, column=2, sticky=E, padx=20, pady=20)
    clear_button.bind('<Button-1>', clear_handler)

def option_handler(event):
    global is_running, start_button, choice

    is_running = False
    start_button.configure(text='Start')

    selection = choice.get()

    if selection == 'glider':
        model.load_pattern(model.glider_pattern, 10, 10)

    elif selection == 'glider gun':
        model.load_pattern(model.glider_gun_pattern, 10, 10)

    elif selection == 'random':
        model.randomize(model.grid_model, model.width, model.height)

    update()

def start_handler(event):
   global is_running, start_button

   if is_running:
       is_running = False
       start_button.configure(text='Start')
   else:
       is_running = True
       start_button.configure(text='Pause')
       update()

def clear_handler(event):
   global is_running, start_button

   is_running = False
   for i in range(0, model.height):
       for j in range(0, model.width):
           model.grid_model[i][j] = 0

   start_button.configure(text='Start')
   update()

def grid_handler(event):
    global grid_view, cell_size

    x = int(event.x / cell_size)
    y = int(event.y / cell_size)

    if (model.grid_model[x][y] == 1):
        model.grid_model[x][y] = 0
        draw_cell(x, y, 'white')
    else:
        model.grid_model[x][y] = 1
        draw_cell(x, y, 'black')
```

```
def update():
    global grid_view, root, is_running

    grid_view.delete(ALL)

    model.next_gen()
    for i in range(0, model.height):
        for j in range(0, model.width):
            if model.grid_model[i][j] == 1:
                draw_cell(i, j, 'black')
    if (is_running):
        root.after(100,update)

def draw_cell(row, col, color):
    global grid_view, cell_size

    if color == 'black':
        outline = 'grey'
    else:
        outline = 'white'

    grid_view.create_rectangle(row*cell_size,
                               col*cell_size,
                               row*cell_size+cell_size,
                               col*cell_size+cell_size,
                               fill=color, outline=outline)

if __name__ == '__main__':
    setup()
    update()
    mainloop()
```

Супер! Теперь можно вдоволь поиграть планерным ружьем!

Чтобы получить этот рисунок, выберите шаблон **glider gun** *и щелкните на кнопке* **Start**

Отличная работа! Симулятор игры "Жизнь" просто великолепен! Дайте знать, если захотите обсудить стратегию продвижения. Могу предложить рекламу в книге.

САМОЕ ГЛАВНОЕ

- Порождающий код генерирует результат, который невозможно предугадать путем анализа кода.
- Игра "Жизнь" была придумана математиком Джоном Конвеем.
- Процесс создания графического интерфейса отличается от привычного процедурного стиля программирования.
- Бумажное прототипирование — это методика тестирования графического интерфейса до его программной реализации.
- Мы спроектировали графическое приложение, разделив его на модель, представление и контроллер.
- `Tkinter` — это модуль Python, предназначенный для создания пользовательских интерфейсов.
- Модуль `Tkinter` позволяет добавлять в приложение виджеты — программные объекты, реализующие элементы пользовательского интерфейса.
- Основное окно программы представляется объектом `Tk`.
- В модуле `Tkinter` имеется несколько менеджеров компоновки, позволяющих упорядочивать виджеты в пределах окна.
- В нашем симуляторе мы управляли размещением виджетов с помощью менеджера компоновки по сетке.
- При создании пользовательских интерфейсов обычно придерживаются событийно-ориентированного, или реактивного, стиля программирования.
- В модели реактивного программирования мы создаем функции-обработчики, которые вызываются при наступлении определенных событий.
- В модуле `Tkinter` обработчик события привязывается к виджету с помощью метода `bind()`.
- Большинству обработчиков передается специальный объект, содержащий дополнительную информацию о событии.
- В большинстве языков программирования существует возможность запуска обработчика события по истечении определенного периода времени.
- У объекта `Tk` имеется метод `after()`, позволяющий запланировать выполнение кода через заданный промежуток времени.
- Выбранный пользователем пункт меню OptionMenu сохраняется в объекте `StringVar`.
- Функция `mainloop()` модуля `Tkinter` осуществляет мониторинг событий, связанных с пользовательским интерфейсом.

Улучшаем симулятор игры "Жизнь"

ЕЩЕ БОЛЬШЕ!

Мы успешно написали прекрасно работающий симулятор, но это только начало, ведь в нем еще многое можно улучшить. Вместо кроссворда обдумайте ряд идей.

ЧТО ПОЧИТАТЬ

- Начать стоит со страницы в Википедии: `https://ru.wikipedia.org/wiki/Игра_"Жизнь"`.
- О математической модели клеточных автоматов, лежащей в основе игры "Жизнь", также можно прочитать в Википедии: `https://ru.wikipedia.org/wiki/Клеточный_автомат`.
- Чтобы увидеть примеры похожих игр, поищите информацию о таких популярных клеточных автоматах, как Maze, HighLife и Day & Night.

УЛУЧШЕНИЯ

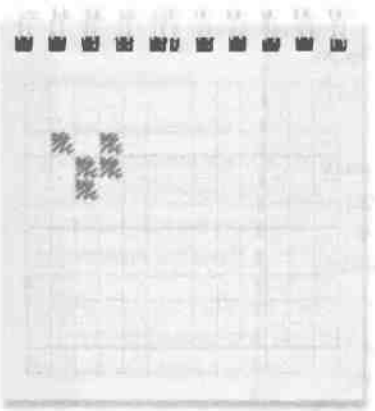
- Попробуйте поменять правила.
  1. Если клетка живая, то она остается живой в следующем поколении.
  2. Если у мертвой клетки два живых соседа, то она становится живой.
- Добавьте возможность сохранять и загружать шаблоны фигур из файла.
- Реализуйте тороидальное игровое поле. В имеющейся реализации прямоугольное поле ограничено краями. Поменяйте код так, чтобы по достижении левого края клетки оказывались справа, а по достижении верхнего края — снизу. В результате решетка превратится в замкнутую поверхность (это не так сложно сделать, как кажется).
- Добавьте цвет: пусть цвет клетки определяет ее возраст (количество поколений, в течение которых она существует).
- Добавьте затенение: пусть у родившейся клетки будет светло-серая заливка, которая со временем будет исчезать.
- Оптимизируйте код, добившись максимальной производительности.

Возьмите карандаш
Решение

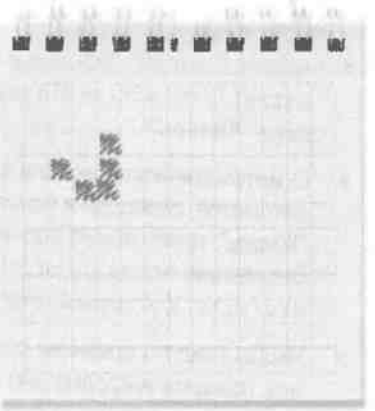
Прежде чем приступать к написанию кода, попробуйте вручную проследить за судьбой нескольких поколений клеток. В качестве начального используйте поколение 1 и вычислите для него поколения 2–6, применяя правила игры "Жизнь". Повторим их еще раз:

- мертвая клетка, у которой три живые соседние клетки, оживает в следующем поколении;
- живая клетка, у которой две или три живые соседние клетки, продолжает жить в следующем поколении;
- живая клетка, у которой менее двух живых соседних клеток, умирает в следующем поколении;
- живая клетка, у которой более трех живых соседних клеток, умирает в следующем поколении.

Поколение 1



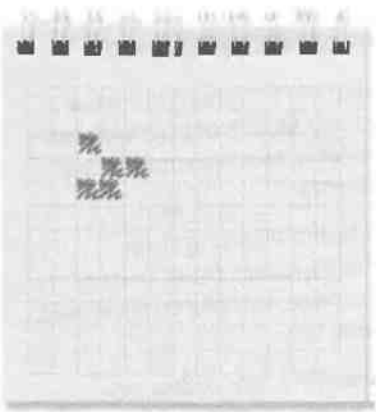
Поколение 2



Поколение 3

Обратите внимание на эти два поколения

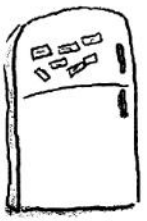
Через четыре поколения фигура повторилась, сдвинувшись по диагонали. Если продолжить игру, фигура будет перемещаться по полю!



Поколение 4

Поколение 5

Поколение 6



Код на магнитиках: решение

Вам повезло! Пока вы изучали код, приведенный во врезке "Готовый рецепт", мы не стали терять время и создали функцию `next_gen()`, код которой записали на магнитиках, прикрепленных к дверце холодильника. Но, как всегда, кто-то решил усложнить нам задачу и все перемешал. Не поможете восстановить исходный код функции? Только будьте внимательны: некоторые магнитики могут оказаться лишними.

ПРАВИЛА ИГРЫ

РОЖДЕНИЕ: новая клетка рождается, только если она окружена ровно тремя живыми клетками.

ЖИЗНЬ: клетка живет до тех пор, пока с ней граничат две или три живые клетки.

СМЕРТЬ: клетка умирает от одиночества, если с ней граничит менее двух живых клеток.

Клетка умирает от перенаселенности, если с ней граничат четыре и более живых клеток.

```
def next_gen():
    global grid_model
    for i in range(0, height):
        for j in range(0, width):
            cell = 0
            count = count_neighbors(grid_model, i, j)
            if grid_model[i][j] == 0:
                if count == 3:
                    cell = 1
            elif grid_model[i][j] == 1:
                if count == 2 or count == 3:
                    cell = 1
```

Чаще всего клетка умирает в следующем поколении, поэтому готовимся заранее

Здесь реализуется правило рождения

Здесь реализуется правило жизни

А это не понадобилось

```
if cou
cell =
count < 2
```

and

or

:

Возьмите карандаш

Решение Мы не можем считать код функции `next_gen()` завершенным, если он тестируется на решетке из одних нулей. Напишите функцию `randomize()`, которая, получив решетку заданной ширины и высоты, случайным образом заполняет ее нулями и единицами.

```
import random

def randomize(grid, width, height):
    for i in range(0, height):
        for j in range(0, width):
            grid[i][j] = random.randint(0,1)
```

Проходим по всей решетке и записываем в каждую ячейку случайное значение 0 или 1

Возьмите карандаш

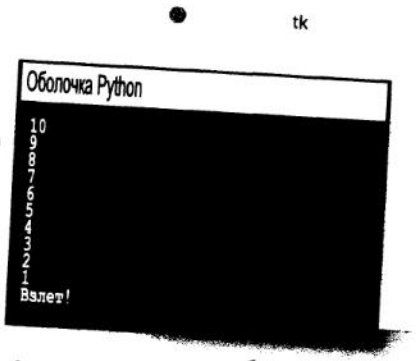
Решение Изучите приведенный ниже код. Что, по-вашему, он выводит? Как работает эта программа? Сверьтесь с ответом, приведенным в конце главы.

```
from tkinter import *

root = Tk()
count = 10

def countdown():
    global root, count


    if count > 0:
        print(count)
        count = count - 1
        root.after(1000, countdown)
    else:
        print('Взлет!')

countdown()
mainloop()
```

Первый раз мы в явном виде вызываем функцию `countdown()`, после чего она с помощью вызова `after()` планирует очередной запуск самой себя. При каждом новом вызове она планирует следующий вызов спустя одну секунду, пока отсчет не дойдет до нуля, после чего функция завершается

Тренируем мозг
Как думаете, это рекурсивная функция?

С технической точки зрения функция никогда не вызывает саму себя. Вместо этого она просит объект `root` вызвать ее в определенный момент в будущем (в данном случае через секунду). Но что-то рекурсивное в этом есть!



Поскольку вы создаете экземпляр виджета `Tk`, при запуске программы появится новое окно

УПРАЖНЕНИЕ (РЕШЕНИЕ)

Что должна делать кнопка **Clear**? При щелчке на ней переменная `is_running` должна стать равной `False`, а все клетки должны обнулиться. Кроме того, на кнопке слева должна снова появиться надпись "Start". Наконец, должна быть вызвана функция `update()`, которая обновит игровое поле (когда все клетки обнулены, поле будет пустым).

Используя код обработчика событий для кнопки **Start** в качестве образца, напишите код управления кнопкой **Clear**.

```
start_button.bind('<Button-1>', start_handler)

def start_handler(event):
    global is_running, start_button

    if is_running:
        is_running = False
        start_button.configure(text='Start')
    else:
        is_running = True
        start_button.configure(text='Pause')
        update()
```

Необходимо сообщить кнопке Clear о наличии обработчика, как и в случае кнопки Start

Это обработчик `start_handler()`, который послужит примером

Это код для регистрации обработчика `clear_handler()`

```
clear_button.bind('<Button-1>', clear_handler)

def clear_handler(event):
    global is_running, start_button

    is_running = False
    for i in range(0, model.height):
        for j in range(0, model.width):
            model.grid_model[i][j] = 0

    start_button.configure(text='Start')
    update()
```

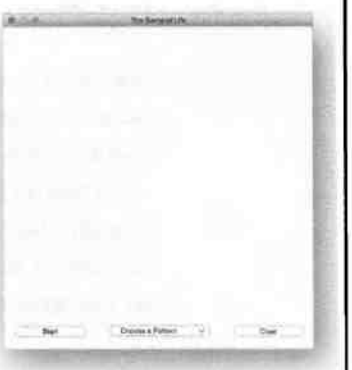
Сначала задаем переменную `is_running` равной `False`

После этого обнуляем клетки модели

Меняем надпись на кнопке слева

Наконец, обновляем окно

Когда напишете код, сравните его с нашим вариантом, приведенным в конце главы, после чего добавьте его в файл *view.py* и протестируйте программу.



Игровое поле очищено! Теперь нужно добавить код, позволяющий добавлять клетки щелчком мыши

12 объектно-ориентированное программирование

Путешествие в страну объектов

В предыдущих главах для создания абстракций в коде применялись функции. Мы всячески придерживались **процедурного** подхода, используя в функциях простые выражения, условные конструкции и циклы `for/while` — ничего такого, что относилось бы к **объектно-ориентированному** программированию. Конечно, вы познакомились с объектами и научились применять их, но вам еще ни разу не приходилось создавать классы и писать полностью объектно-ориентированный код. Пришло время попрощаться со скучным процедурным прошлым. Нас ожидает новый мир, полный объектов и надежд на лучшую жизнь (в профессиональном смысле; хотя, если вы станете профессиональным программистом, ваша жизнь наверняка изменится в лучшую сторону). Но помните: открыв для себя страну объектов, вы захотите остаться там навсегда. Пришлете нам потом открытку.

Другой подход к структурированию программ

Помните, в главе 1 мы обсуждали навыки, которые понадобятся вам для изучения программирования? Первый из них заключается в умении разбивать сложную задачу на последовательность более простых действий, а второй предполагает освоение языка программирования, на котором вы сможете описать эти действия компьютеру. К настоящему моменту вы преуспели и в том, и в том.

Все это замечательно, поскольку полученные знания навсегда останутся с вами и послужат основой для разработки программ. Но существует и другой подход к структурированию программ, поддерживаемый почти всеми современными языками программирования и предпочитаемый профессиональными программистами. Он известен как *объектно-ориентированное программирование* (ООП), с которым вы успели вкратце познакомиться в главе 7.

В ООП, вместо того чтобы описывать привычный алгоритм решения задачи с помощью функций, условий и т.п., мы моделируем набор взаимодействующих объектов. Это непростая тема со своим жаргоном, своими методиками и приемами программирования. В то же время данный стиль программирования зачастую оказывается более интуитивно понятным при решении самых разных задач, в чем вы вскоре убедитесь.

Объектно-ориентированному программированию посвящено множество книг. В этой главе мы постараемся понять саму суть ООП, чтобы вы научились анализировать объектно-ориентированный код. Вы достигнете того уровня, когда сможете создавать собственные классы и применять их в программах. Получив базовые знания, вы будете готовы продолжить самостоятельное обучение.

Помните классы, о которых говорилось в главе 7?

Преимущества объектно-ориентированного программирования

ООП позволяет писать программы на более высоком уровне, концентрируясь на общей схеме их работы.

Мы уже говорили о высокоуровневом подходе, когда изучали способы создания программных абстракций в виде функций. Это позволяло решать задачи путем вызова готовых функций, вместо того чтобы заниматься анализом "спагетти-кода", т.е. мешанины из низкоуровневых инструкций `if`, `elif`, `for` и операций присваивания. В ООП данная концепция поднимается на еще более высокий уровень: вы моделируете реальные (или виртуальные) объекты, описывая их состояние и поведение, а затем позволяете им взаимодействовать друг с другом для решения поставленных задач.

Например, рассмотрим следующую последовательность действий:

1) изготовить нагреватель из подходящего куска провода;
2) подключить его к электросети;
3) включить электричество;
4) отрезать кусок хлеба;
5) удерживать его на расстоянии 2 см от нагревателя;
6) держать хлеб до полной прожарки;
7) убрать хлеб;
8) выключить электричество.

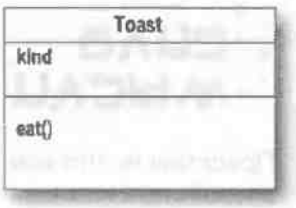
Процедурный способ анализа задачи

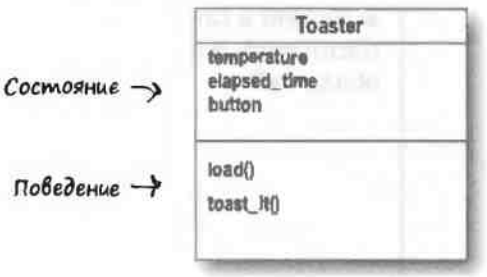
А вот она же, но записанная с помощью объектов:

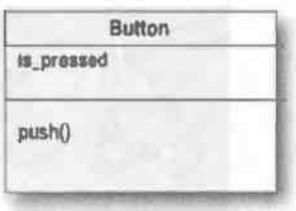
1) поместить `Toast` (тост) в `Toaster` (тостер);
2) нажать `Button` (кнопка) на `Toaster`;
3) вынуть `Toast` из `Toaster`.

Объектно-ориентированный способ анализа задачи

Первая последовательность процедурная, а вторая — объектно-ориентированная: у нас есть набор объектов (`Toast`, `Toaster` и `Button`), и мы анализируем, как ими пользоваться (помещаем ломтик хлеба в тостер и нажимаем кнопку), не опускаясь до рассмотрения технических деталей (до какой температуры нагревается спираль, как долго мы сможем удерживать хлеб возле спирали и т.п.).





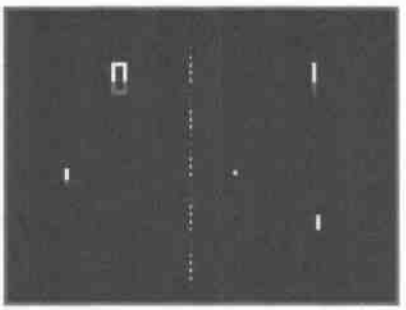


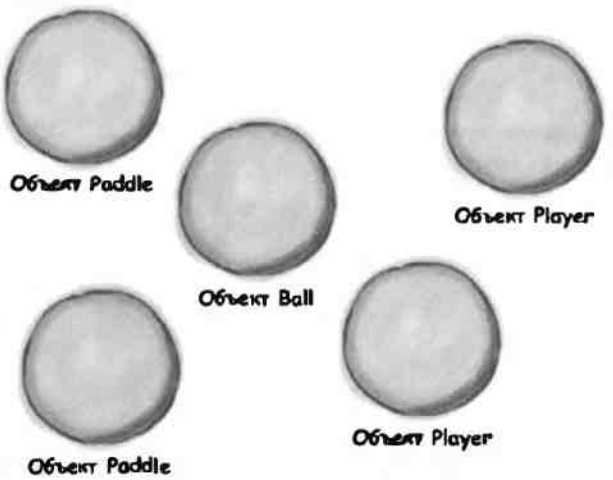
Помните, в главе 7 мы говорили о том, что все объекты создаются на основе классов, а также обладают атрибутами (состоянием) и методами (поведением)?

Сила мысли

Представьте, что вам поручили разработать классическую аркадную видеоигру *Pong* (пинг-понг). Что вы выберете в качестве объектов? Каким состоянием и поведением они должны обладать?

Сыграем в пинг-понг?





Что вам нравится в объектно-ориентированных языках программирования?

"Объекты проще понять, чем мешанину из функций и переменных в модуле".

Джой, 27 лет, разработчик программного обеспечения

"Мне нравится, что данные и функции, которые работают с этими данными, хранятся в одном объекте".

Брэд, 19 лет, программист

"Для меня это естественный стиль программирования. Он лучше всего соответствует реальным задачам".

Крис, 39 лет, руководитель проекта

"Вот кто бы говорил?! Сам-то за пять лет не написал ни одной строчки кода!"

Дэрил, 44 года, сотрудник Криса

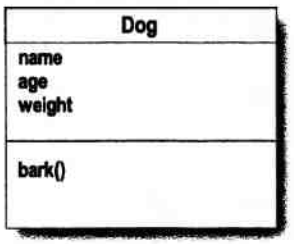
"Люблю черепашек!"

Эйвери, 7 лет, юный хакер

Разработка своего первого класса

В главе 7 вы узнали о том, как создавать объекты на основе классов. С тех пор мы познакомились с целым рядом объектов: встроенные типы (в частности, строки и числа с плавающей точкой), черепашки, виджеты, веб-запросы и многое другое. Но мы так ни разу и не написали свой собственный класс. Этим мы сейчас и займемся.

В процедурном программировании мы создавали предварительный план в виде псевдокода. В случае объектов (точнее, классов, на основе которых они создаются) тоже не помешает все заранее спланировать. Начнем с простого и составим схему класса `Dog` (собака).



У класса есть имя, `Dog`...

...состояние, описываемое атрибутами...

...и поведение, реализуемое методом `bark()`

Прежде чем писать код класса, нужно понять, как он будет применяться. Вот пример кода, в котором задействован будущий класс `Dog`.

Пока что не нужно вводить или запускать этот код

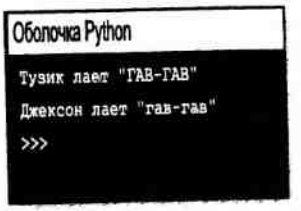
```
tuzik = Dog('Тузик', 12, 38)
jackson = Dog('Джексон', 9, 12)
tuzik.bark()
jackson.bark()
```

Это вызов конструктора

Сначала мы создаем двух собак с разными атрибутами

Затем вызываем метод `bark()`, чтобы услышать каждую из собак*

Это наши результаты



* Заметили, что каждая из собак лает по-своему? Интересно, почему?

В соответствии с данным примером нам нужно, чтобы конструктор класса `Dog` создавал объект собаки, получая ее имя, возраст и вес в качестве атрибутов. Кроме того, наш класс должен включать метод `bark()`, описывающий лай собаки в зависимости от ее характеристик: для крупных собак он будет возвращать строку "ГАВ-ГАВ", а для мелких — "гав-гав". Кажется, что-то похожее уже было?

Помните Тузика?

Написание кода класса

Мы начнем с того, что в ООП называется *конструктором*. Это специальный метод, содержащий код инициализации объектов `Dog`. Методом `bark()` мы займемся позже.

Вот код конструктора.

Создаем класс `Dog`

В Python конструктор — это функция `__init__()`*, точнее, метод, а не функция, потому что он определен внутри класса*

```
class Dog:
    def __init__(self, name, age, weight):
        self.name = name
        self.age = age
        self.weight = weight
```

В теле конструкутора настраиваются атрибуты объекта: имя, возраст и вес

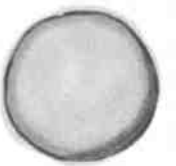
Пытливым читателям наверняка не терпится узнать, почему в списке параметров содержится запись `self` *и почему она предшествует именам атрибутов в коде?*

Как работает конструктор

Ключ к пониманию того, как работает конструктор (и другие методы класса), дает изучение параметра `self`. Давайте пошагово выясним, что происходит при вызове конструктора. Будьте внимательны, чтобы ничего не упустить.

```
tuzik = Dog('Тузик', 12, 38)
```

1. Конструктор вызывается, когда после имени класса указывается список аргументов в скобках. В первую очередь создается новый, пока еще пустой, объект `Dog`.



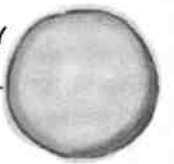
Объект Dog

Это наш новый объект `Dog`*, только он пока что не содержит никаких атрибутов*

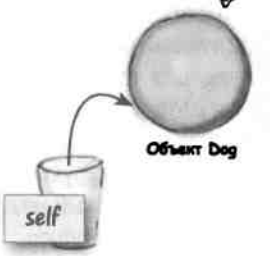
2 Аргументы конструктора передаются в функцию `__init__()`, но это еще не все: интерпретатор Python подставляет только что созданный объект в качестве первого параметра функции, именуя его `self`.

...а именно: только что созданный объект передается в первом аргументе

Далее вызывается конструктор класса с переданными ему аргументами и одним дополнением...



После завершения второго этапа созданный объект Dog присваивается параметру self

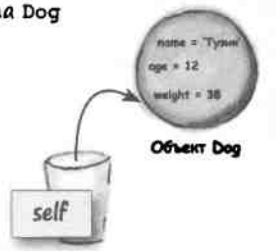



```
__init__(    , 'Тузик', 12, 38)

def __init__(self, name, age, weight):
```

---

3 Далее выполняется код конструктора. Параметры метода (`name`, `age` и `weight`) присваиваются одноименным атрибутам экземпляра объекта `Dog` с использованием точечной нотации.

После завершения третьего этапа все аргументы, переданные конструктору, назначаются атрибутам объекта Dog



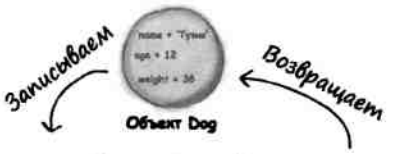
Новый объект Dog передается в качестве первого аргумента

```
def __init__(self, name, age, weight):
    self.name = name
    self.age = age
    self.weight = weight
```

Поскольку self — это новый объект Dog, мы записываем значения name, age и weight в атрибуты созданного объекта

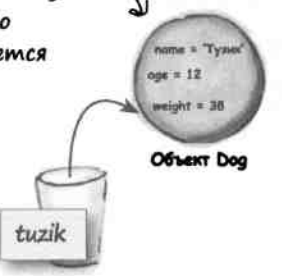
---

4 По завершении конструктора интерпретатор возвращает объект `Dog` в качестве результата. В нашем случае объект `Dog` присваивается переменной `tuzik`.




```
tuzik = Dog('Тузик', 12, 38)
```

После завершения четвертого этапа созданный объект Dog, все атрибуты которого установлены, записывается в переменную tuzik



Не бойтесь задавать вопросы

В: Почему метод `__init__()` не содержит инструкцию `return`, и тем не менее конструктор возвращает значение?

О: При запуске конструктора интерпретатор Python выполняет целый ряд скрытых операций. Во-первых, создается новый объект, затем он передается в качестве первого аргумента методу `__init__()`, и, наконец, этот объект возвращается при завершении конструктора. Именно так работает встроенный механизм создания объектов в Python.

В: Имя `self` является зарезервированным словом?

О: Нет, это просто общепринятое соглашение для названия копии объекта, которая передается в качестве первого аргумента метода `__init__()` при вызове конструктора. Называть параметр именно так совсем не обязательно. Тем не менее программисты стараются придерживаться подобных соглашений, чтобы не сбивать никого с толку.

Во многих объектно-ориентированных языках программирования имеются аналогичные соглашения, только с другим ключевым словом, например `this`

Во избежание недоразумений старайтесь не использовать имя `self` для глобальных и локальных переменных.

В: Получается, что атрибуты объекта — это всего-навсего переменные, хранящие самые обычные значения?

О: Так и есть. В атрибут, как и в переменную, можно записать любое допустимое значение Python. И предваряя следующий вопрос: методы — это функции Python, которые объявляются не в общем коде, а внутри объекта. Правда, у них есть одно отличие: параметр `self`, с которым вы уже познакомились.

Тест-драйв

Метод `bark()` еще не написан, но имеющийся код можно протестировать и без него. Скопируйте приведенный ниже код в файл *dog.py* и запустите его.

```
class Dog:
    def __init__(self, name, age, weight):
        self.name = name
        self.age = age
        self.weight = weight

def print_dog(dog):
    print(dog.name + ': возраст -', dog.age,
                   ', вес -', dog.weight)

tuzik = Dog('Тузик', 12, 38)
jackson = Dog('Джексон', 9, 12)
print_dog(tuzik)
print_dog(jackson)
```

Это наш новый класс...

...и функция для вывода информации о собаках

Это не самый рациональный способ вывода информации о собаках, но вскоре вы увидите, как все можно улучшить

Создаем два экземпляра объектов `Dog` и передаем их в функцию `print_dog()`

```
Оболочка Python
Тузик: возраст - 12, вес - 38
Джексон: возраст - 9, вес - 12
>>>
```

Все работает так, как и ожидалось!

Метод bark()

Прежде чем писать код метода `bark()`, нужно понять разницу между функциями и методами. Понятно, что методы определяются в классе, но у них есть и другие особенности. В частности, способ вызова: метод всегда вызывается для *объекта*. Например:

```
tuzik.bark()
```

Объекты

```
jackson.bark()
```

Как вы увидите далее, методы обычно обрабатывают атрибуты объекта, для которого они вызываются. Вот почему в качестве первого аргумента методам передается сам объект.

Подобно тому, как это происходит в конструкторе `__init()__`

Теперь давайте напишем метод `bark()` и посмотрим, как он работает.

```
class Dog:
    def __init__(self, name, age, weight):
        self.name = name
        self.age = age
        self.weight = weight

    def bark(self):
        if self.weight > 29:
            print(self.name, 'лает "ГАВ-ГАВ"')
        else:
            print(self.name, 'лает "гав-гав"')
```

Метод имеет тот же синтаксис, что и функция

Методу `bark()` в качестве первого аргумента передается объект, для которого он вызван (других параметров у метода нет)

Здесь мы проверяем вес собаки и, если он превышает 29, выводим "ГАВ-ГАВ", в противном случае выводим "гав-гав"

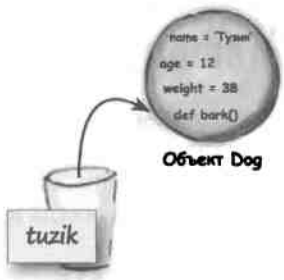
Всегда используйте параметр `self` для доступа к атрибутам объекта

Поскольку в качестве первого аргумента всегда передается объект, для которого вызывается метод, к аргументам этого объекта можно получить доступ через параметр `self`

В каждом случае мы также выводим имя собаки

Как работает метод

Для полного понимания попробуем пошагово проанализировать вызов метода.



Рассмотрим, что происходит при вызове метода `bark()` *для объекта* `tuzik`

```
tuzik.bark()
```

1. При вызове метода объекта (в нашем случае объекта `tuzik`) интерпретатор Python передает ему в качестве первого аргумента сам объект, а остальные аргументы подставляются в порядке указания (у метода `bark()` они отсутствуют).

Когда вызывается метод `bark()` *объекта* `tuzik`*, интерпретатор Python передает этот объект в качестве первого аргумента, записывая его в параметр* `self`

```
def bark(self):
    if self.weight > 29:
        print(self.name, 'лает "ГАВ-ГАВ"')
    else:
        print(self.name, 'лает "гав-гав"')
```

Параметр `self` *содержит объект* `tuzik`*, поэтому значение* `self.weight` *равно 38, что превышает 29. В результате выполняется первая функция* `print()`

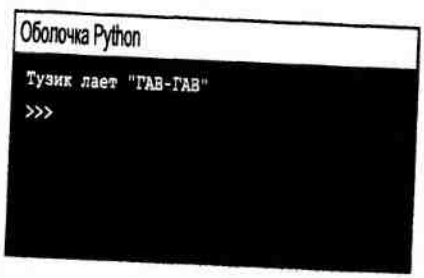
2. Далее выполняются инструкции в теле метода. В первой строке проверяется, превышает ли атрибут `self.weight` значение `29`. В нашем случае параметр `self` содержит объект `tuzik` класса `Dog`, атрибут `self.weight` которого равен `38`. Таким образом, условие справедливо, и выполняется первая функция `print()`.

```
print(self.name, 'лает "ГАВ-ГАВ"')
```

Функция `print()` *сначала выводит атрибут* `name` *объекта, присвоенного параметру* `self`

3. Функция `print()` сначала выводит значение атрибута `self.name`, где `self`, как и прежде, представляет объект, для которого вызывается метод `bark()`. В нашем случае это объект `tuzik` с именем `'Тузик'`, поэтому на экране появляется сообщение `'Тузик лает "ГАВ-ГАВ"'`.

4. На этом работа метода `bark()` завершается. В нашем случае он ничего не возвращает, но есть и методы, которые возвращают значения.



Возьмите карандаш

В главе 2 мы написали программу, которая определяла возраст собаки по человеческим меркам. Добавьте в наш класс `Dog` метод, реализующий такие вычисления, и назовите его `human_years()`. Метод не требует аргументов и возвращает целочисленный результат.

```
dog_name = input("Как зовут вашу собаку? ")
dog_age = input("Сколько лет вашей собаке? ")
human_age = int(dog_age) * 7
print('Вашей собаке',
      dog_name,
      'сейчас',
      human_age,
      'по меркам людей')
```

Это код из главы 2. Приятно вспомнить...

Это наш код на данный момент. Добавьте метод `human_years()`*, возвращающий возраст собаки по человеческим меркам*

```
class Dog:
    def __init__(self, name, age, weight):
        self.name = name
        self.age = age
        self.weight = weight

    def bark(self):
        if self.weight > 29:
            print(self.name, 'лает "ГАВ-ГАВ"')
        else:
            print(self.name, 'лает "гав-гав"')
```

Добавьте новый метод здесь

```
def print_dog(dog):
    print(dog.name + ': возраст -', dog.age,
                     ', вес -', dog.weight)


tuzik = Dog('Тузик', 12, 38)
jackson = Dog('Джексон', 9, 12)
print(tuzik.name + ": возраст по меркам людей -", tuzik.human_years())
print(jackson.name + ": возраст по меркам людей -", jackson.human_years())
```

Немного о наследовании

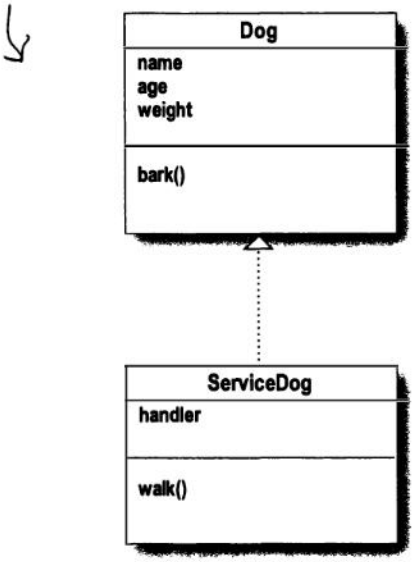
Предположим, что в приложении требуется описать новый класс — служебных собак-поводырей, обученных помогать людям. Это те же собаки, но с дополнительными характеристиками. Нужно ли все начинать заново и писать класс `ServiceDog` с нуля? Это было бы нелепо, ведь мы потратили немало времени на разработку существующего класса `Dog`, и нам хотелось бы воспользоваться имеющимися наработками, если такое возможно. Конечно возможно!

Все современные языки программирования позволяют классам заимствовать атрибуты и методы у других классов. Эта технология называется *наследованием* и является краеугольным камнем объектно-ориентированного программирования.

Применительно к задаче описания собак-поводырей мы могли бы определить класс `ServiceDog` так, чтобы он наследовал атрибуты (имя, возраст и вес) и способность лаять от исходного класса `Dog`. В то же время он может обладать собственными атрибутами, такими как имя хозяина (человека, для которого собака служит поводырем), и методами, например `walk()`.

Давайте попробуем определить такой класс.

Так выглядит диаграмма класса `ServiceDog`, *наследуемого от класса* `Dog`



Сила мысли

Какие еще категории собак могут наследовать класс `Dog`? Какими собственными атрибутами и методами они при этом будут обладать?

Реализация класса ServiceDog

Напишем код класса ServiceDog и проанализируем его синтаксис и семантику.

Как вы помните, синтаксис определяет правила записи инструкций, а семантика — их назначение

В соответствии с этим синтаксисом мы объявляем новый класс ServiceDog, наследуемый от класса Dog

Имя нового класса

Наследуемый класс

Это конструктор класса ServiceDog

У него те же параметры, что и в классе Dog, плюс добавлен еще один параметр

Конструктор класса ServiceDog требует дополнительного параметра handler

```
class ServiceDog(Dog):

    def __init__(self, name, age, weight, handler):

        Dog.__init__(self, name, age, weight)

        self.handler = handler

    def walk(self):

        print(self.name, 'помогает своему хозяину (' + self.handler + ') ходить')
```

В этой строке вызывается конструктор класса Dog, которому передаются все необходимые аргументы, включая self

Заносим новый атрибут в объект self

У нас также появился новый метод walk()

В методе walk() используются атрибуты классов Dog и ServiceDog

С синтаксисом мы разобрались, теперь рассмотрим, как применять такой класс.

Создаем объект ServiceDog для хозяина 'Джозеф'

```
rudy = ServiceDog('Руди', 8, 38, 'Джозеф')
print("Собаку зовут", rudy.name)
print("Хозяина собаки зовут", rudy.handler)
print_dog(rudy)
rudy.bark()
rudy.walk()
```

Можно получать доступ к унаследованным атрибутам, таким как name, или к атрибутам класса ServiceDog, таким как handler

Класс ServiceDog — потомок класса Dog, поэтому можно вызывать метод print_dog()

Можно вызывать унаследованные методы, такие как bark()...

...или же методы, специфичные только для класса ServiceDog, такие как walk()

Вообще-то, собаки-поводыри не должны громко лаять

Оболочка Python

```
Собаку зовут Руди
Хозяина собаки зовут Джозеф
Руди: возраст - 8, вес - 38
Руди лает "ГАВ-ГАВ"
Руди помогает своему хозяину (Джозеф) ходить
>>>
```

Руди, собака-поводырь

Подклассы

При создании класса, наследующего другой класс, мы получаем *подкласс*, или *производный класс*. В нашем случае подкласс `ServiceDog` является производным от класса `Dog`.

Давайте еще раз рассмотрим синтаксис и семантику подкласса `ServiceDog`, начав с инструкции `class`.

При создании нового класса можно либо определить его с нуля, либо указать в скобках наследуемый класс

И еще о терминологии: класс `Dog` часто называют базовым, поскольку он служит основной для всех производных классов (пока что такой класс всего один, но мы создадим еще)

```
class ServiceDog(Dog):
```

Такой класс называют суперклассом. В данном случае `Dog` — это суперкласс класса `ServiceDog`

А мы предупреждали, что в ООП много терминологии!

Теперь изучим конструктор `__init__()`.

В объявлении параметров нет ничего необычного, мы просто добавили один новый, `handler` (новых параметров может быть сколько угодно)

```
def __init__(self, name, age, weight, handler):
    Dog.__init__(self, name, age, weight)
    self.handler = handler
```

Следующая строка заслуживает внимания. Здесь вызывается конструктор базового класса `Dog`. Тем самым задаются все атрибуты, характерные для собак. Если этого не сделать, то для создаваемого объекта не будут заданы имя, возраст и вес

В последней строке параметр `handler` записывается в одноименный атрибут

Этот атрибут будет только у объектов `ServiceDog`, так как лишь они выполняют данный метод `__init__()`

Обычно при создании подкласса это первое, что делается в конструкторе

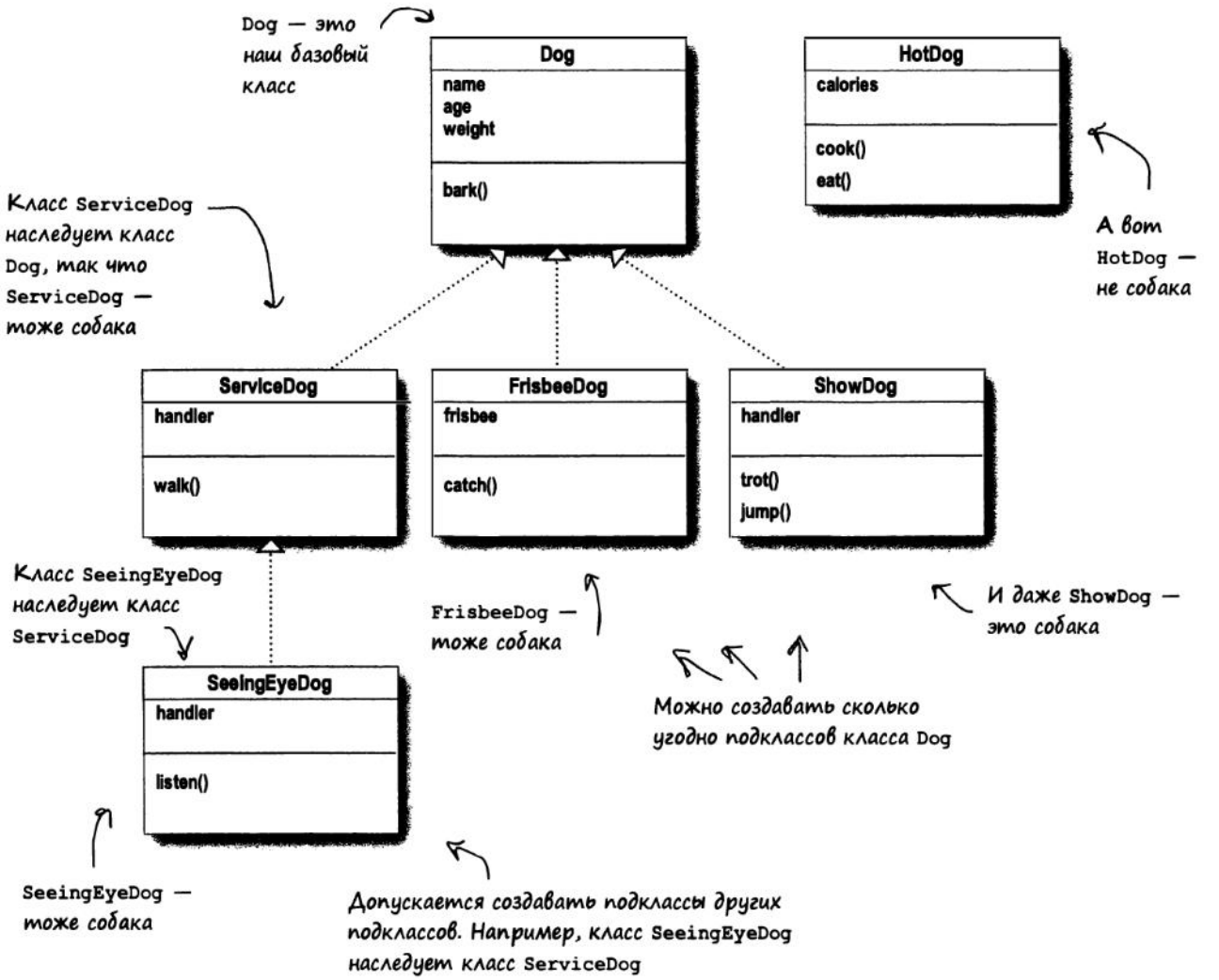
Наконец, у нас есть новый метод `walk()`.

В подклассе можно определять собственные методы. Учтите, что этот метод доступен только для объектов, порождаемых от класса `ServiceDog`, но не от старого класса `Dog`

```
def walk(self):
    print(self.name, 'помогает своему хозяину (' + self.handler + ') ходить')
```

ServiceDog — это потомок класса Dog

Когда между двумя классами существует наследственная связь, один из них называется *потомком* другого. В частности, класс ServiceDog является потомком класса Dog. Данная концепция относится не только к случаям прямого наследования. Например, согласно приведенной ниже схеме класс SeeingEyeDog является потомком класса ServiceDog, который, в свою очередь, наследует класс Dog. Таким образом, класс SeeingEyeDog – потомок и класса ServiceDog, и класса Dog. С другой стороны, класс ServiceDog наследует только класс Dog, но не SeeingEyeDog, поскольку не располагает возможностями последнего (наоборот, класс SeeingEyeDog наследует атрибуты и поведение класса ServiceDog, расширяя его).



Проверка принадлежности к классу

Можно ли для произвольного объекта определить, экземпляром какого класса он является? Предположим, кто-то создал следующий объект и передал его вам:

```
mystery_dog = ServiceDog('Загадка', 5, 13, 'Хелен')
```

Это объект класса `Dog`? Или `ServiceDog`? Или какой-то другой объект? Как это определить?

Здесь нам пригодится встроенная функция `isinstance()`. Вот как она работает.

Функция `isinstance()` получает объект и класс в качестве аргументов

```
if isinstance(mystery_dog, ServiceDog):
    print("Да, это ServiceDog")
else:
    print('Нет, это не ServiceDog')
```

Функция `isinstance()` возвращает `True`, если объект относится к указанному классу или производному от него

В данном случае функция `isinstance()` вернула `True`, поскольку `MysteryDog` — это объект класса `ServiceDog`

Оболочка Python

```
Да, это ServiceDog
>>>
```

Еще один пример.

```
f isinstance(mystery_dog, Dog):
   print("Да, это Dog")
lse:
   print('Нет, это не Dog')
```

Теперь функция возвращает `True`, если объект относится к классу `Dog`

вернула `True`, так как `MysteryDog` наследует класс `Dog`

Оболочка Python

```
Да, это Dog
>>>
```

И еще один.

Теперь функция `isinstance()` возвращает `True`, если объект относится к классу `SeeingEyeDog`

```
if isinstance(mystery_dog, SeeingEyeDog):
    print("Да, это SeeingEyeDog")
else:
    print('Нет, это не SeeingEyeDog')
```

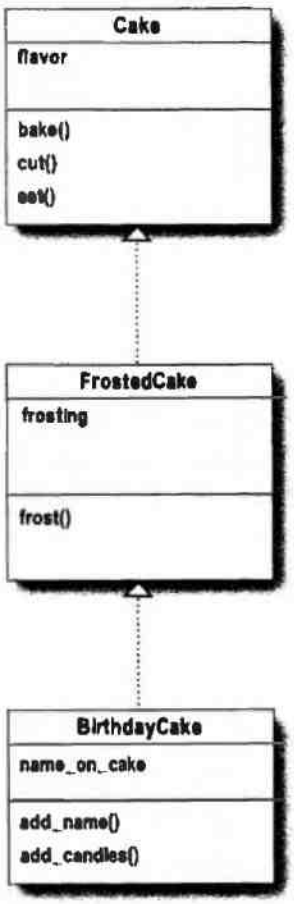
На этот раз функция `isinstance()` возвращает `False`, так как `ServiceDog` не является подклассом `SeeingEyeDog`

Оболочка Python

```
Нет, это не SeeingEyeDog
>>>
```

Возьмите карандаш

Используя показанную слева диаграмму классов, укажите справа значения, возвращаемые функцией `isinstance()`. Помните, что эта функция может возвращать только `True` или `False`. Первое задание мы выполнили за вас.



Ваш ответ: True или False

```
simple_cake = Cake()
chocolate_cake = FrostedCake()
bills_birthday_cake = BirthdayCake()
```

False — `isinstance(simple_cake, BirthdayCake)`

_____ `isinstance(simple_cake, FrostedCake)`

_____ `isinstance(simple_cake, Cake)`

_____ `isinstance(chocolate_cake, Cake)`

_____ `isinstance(chocolate_cake, FrostedCake)`

_____ `isinstance(chocolate_cake, BirthdayCake)`

_____ `isinstance(bills_birthday_cake, FrostedCake)`

_____ `isinstance(bills_birthday_cake, Cake)`

_____ `isinstance(bills_birthday_cake, BirthdayCake)`

УПРАЖНЕНИЕ

Подклассы можно порождать не только от собственных, но и от встроенных классов Python. Как насчет того, чтобы создать подкласс для строкового класса Python (т.е. `str`)? Давайте напишем подкласс `PalindromeString`, включающий метод `is_palindrome()`. Изучите приведенный ниже код и протестируйте его. Подумайте над тем, какие еще методы можно было бы добавить в этот подкласс. Не ищите решения в конце главы: все находится здесь.

Мы создадим класс `PalindromeString`, являющийся подклассом встроенного класса `str`

```
class PalindromeString(str):

    def is_palindrome(self):
        i = 0
        j = len(self) - 1
        while i < j:
            if self[i] != self[j]:
                return False
            i = i + 1
            j = j - 1
        return True
```

Мы лишь добавим в подкласс новый метод, поэтому нет необходимости реализовывать конструктор

Если не предоставить конструктор, то при создании объекта такого класса вызывается конструктор его суперкласса (в данном случае `str`)

Это наш новый метод. Помните итеративную версию функции `is_palindrome()`?

Обратите внимание на использование параметра `self`. В данном случае `self` — это сам объект `str`, с которым можно делать все то же самое, что и со строкой

Давайте проверим наш подкласс. Помните о том, что объект `PalindromeString` является строкой, поэтому с ним можно делать все то же самое, что и со строкой. Он наследует всю функциональность от класса `str`

```
word = PalindromeString('radar')
word2 = PalindromeString('rader')
print(word + ': длина -', len(word) + ', верхний регистр -', word.upper())
print(word, word.is_palindrome())
print(word2 + ': длина -', len(word2) + ', верхний регистр -', word2.upper())
print(word2, word2.is_palindrome())
```

Вот наши результаты. Сможете придумать другие способы использования подклассов класса `str`?

Помните, в главе 9 мы искали способ сравнивать слова без учета регистра? Можете использовать метод `upper()` или `lower()` для преобразования строки в верхний или нижний регистр перед сравнением

Оболочка Python

```
radar: длина - 5, верхний регистр - RADAR
radar True
rader: длина - 5, верхний регистр - RADER
rader False
>>>
```

Опишите себя

Как описать самого себя в коде? С помощью метода `__str__()`! Пора заменить функцию `print_dog()` чем-то объектно-ориентированным. В Python существует следующее соглашение: если в класс добавляется метод `__str__()`, возвращающий строку, то именно он будет вызываться при выводе любого объекта данного класса с помощью функции `print()`.

```
class Dog:
    def __init__(self, name, age, weight):
        self.name = name
        self.age = age
        self.weight = weight

    def bark(self):
        if self.weight > 29:
            print(self.name, 'лает "ГАВ-ГАВ"')
        else:
            print(self.name, 'лает "гав-гав"')

    def human_years(self):
        human_age = self.age * 7
        return human_age

    def __str__(self):
        return "Я собака по имени " + self.name
```

Добавим в класс Dog метод __str__(), который будет формировать строку приветствия для функции print()

Проведем тест, вызвав функцию print() для нескольких объектов

```
tuzik = Dog('Тузик', 12, 38)
jackson = Dog('Джексон', 9, 12)
rudy = ServiceDog('Руди', 8, 38, 'Джозеф')
print(tuzik)
print(jackson)
print(rudy)
```

Мы не меняем способ работы функции print(), а лишь задаем способ вывода информации о конкретном экземпляре класса Dog

Не бойтесь задавать вопросы

В: Я правильно понимаю, что при создании объекта ServiceDog фактически создаются два объекта, Dog и ServiceDog, каждый со своими атрибутами и методами?

О: Нет, создается лишь один объект со всеми атрибутами. Что касается методов, то они заимствуются из определений классов и не хранятся в объектах. Но с концептуальной точки зрения считается, что создается только один экземпляр ServiceDog.

В: Я слышал, что в Python поддерживается множественное наследование. Так ли это?

О: Да, множественное наследование означает, что атрибуты и методы наследуются не от одного класса, а сразу от нескольких. Например, если вы захотите описать летающий автомобиль, то он будет наследовать поведение как от класса автомобилей, так и от класса самолетов. Прекрасно, что вы знаете о концепции множественного наследования, но далеко не все специалисты считают это хорошей практикой. В некоторых языках программирования множественное наследование даже запрещено. В любом случае к его рассмотрению стоит переходить только после углубленного изучения ООП. Также не забывайте, что задачу всегда можно решить, не используя множественное наследование.

Оболочка Python

```
Я собака по имени Тузик
Я собака по имени Джексон
Я собака по имени Руди
>>>
```

Ого, функция сработала даже для Руди, который относится к классу ServiceDog!

Переопределение и расширение поведения

Несколькими страницами ранее наш пес Руди громко лаял "ГАВ-ГАВ". Однако такое поведение недопустимо для собаки, которая на службе. Если взглянуть на определение класса Dog, то можно заметить, что данное поведение реализовано для собак весом 29 кг и больше. Должны ли собаки-поводыри всегда вести себя так? Конечно нет, ведь мы всегда можем переопределить и расширить поведение наследуемого класса. Вот как это сделать.

```
class Dog:
    def __init__(self, name, age, weight):
        self.name = name
        self.age = age
        self.weight = weight

    def bark(self):
        if self.weight > 29:
            print(self.name, 'лает "ГАВ-ГАВ"')
        else:
            print(self.name, 'лает "гав-гав"')

    def human_years(self):
        human_age = self.age * 7
        return human_age

    def __str__(self):
        return "Я собака по имени " + self.name

class ServiceDog(Dog):
    def __init__(self, name, age, weight, handler):
        Dog.__init__(self, name, age, weight)
        self.handler = handler
        self.is_working = False

    def walk(self):
        print(self.name, 'помогает своему хозяину (' + self.handler + ') ходить')

    def bark(self):
        if self.is_working:
            print(self.name, 'говорит: "Не могу лаять, я работаю"')
        else:
            Dog.bark(self)
```

В методе __init__() можно добавить в класс любые атрибуты, которые нужны для формирования внутреннего состояния объекта. Атрибуты не обязаны соответствовать параметрам метода

В класс ServiceDog добавляется новый булев атрибут is_working, первоначально равный False

Мы переопределяем метод bark() в классе ServiceDog. Всякий раз, когда метод вызывается для собаки типа ServiceDog, выполняется именно эта версия, а не версия из класса Dog. Это называется перекрытием метода

Если атрибут is_working равен True, собака сообщает о том, что она работает и не может лаять. В противном случае вызывается метод bark() класса Dog, которому передается объект self

Тест-драйв

Пора протестировать имеющийся код. Замените содержимое файла *dog.py* кодом, приведенным на предыдущей странице, после чего добавьте в конец файла показанный ниже код.

Создаем объект для Руди

```
rody = ServiceDog('Руди', 8, 38, 'Джозеф')

rody.bark()

rody.is_working = True
rody.bark()
```

Просим его залаять (помните: атрибут `is_working` изначально равен `False`)

Делаем атрибут `is_working` равным `True` и пробуем снова

Подобное еще не встречалось, но принцип прост: атрибуту можно присвоить значение, как обычной переменной

Класс `ServiceDog` расширяет функциональность класса `Dog`, но если атрибут `is_working` равен `False`, то методы `bark()` ведут себя одинаково

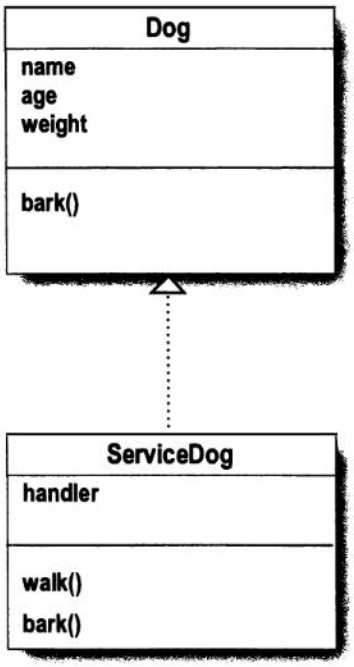
Оболочка Python

```
Руди лает "ГАВ-ГАВ"
Руди говорит: "Не могу лаять, я работаю"
>>>
```

Учим термины

Вы уже знаете, что в объектно-ориентированном программировании используется собственный жаргон. Давайте еще раз вспомним основные термины, прежде чем двигаться дальше. Знать профессиональный сленг не помешает!

У классов есть **атрибуты** и **методы**



Мы говорим, что `Dog` – это **суперкласс** класса `ServiceDog`. Класс `Dog` еще называют **базовым** классом

`Dog` и `ServiceDog` – **это классы**

Мы говорим, что `ServiceDog` является **подклассом** класса `Dog`. Его еще называют **производным** классом

Мы говорим, что класс `ServiceDog` **наследует** класс `Dog`

Мы говорим, что класс `ServiceDog` **переопределяет** метод `bark()` класса `Dog`

Можно создавать **экземпляры (объекты)** классов `Dog` и `ServiceDog` путем вызова их **конструкторов**

Сила мысли

Правда ли, что все объекты `ServiceDog` являются экземплярами класса `Dog`, но не все объекты `Dog` являются экземплярами класса `ServiceDog`?

Ответ: True.

Возьмите карандаш

Изучите показанный слева код классов, включающий несколько переопределяемых методов. Мысленно выполните код, приведенный справа, и запишите предполагаемые результаты. Решение будет дано в конце главы.

```
class Car():
   def __init__(self):
       self.speed = 0
       self.running = False

   def start(self):
       self.running = True

   def drive(self):
       if self.running:
           print('Машина едет')
       else:
           print('Сначала заведите машину')

class Taxi(Car):
   def __init__(self):
       Car.__init__(self)
       self.passenger = None
       self.balance = 0.0

   def drive(self):
       print('Бип-бип, с дороги!')
       Car.drive(self)

   def hire(self, passenger):
       print('Заказано', passenger)
       self.passenger = passenger

   def pay(self, amount):
       print('Заплачено', amount)
       self.balance = self.balance + amount
       self.passenger = None

class Limo(Taxi):
   def __init__(self):
       Taxi.__init__(self)
       self.sunroof = 'закрыт'

   def drive(self):
       print('Лимузин - не роскошь')
       Car.drive(self)

   def pay(self, amount, big_tip):
       print('Заплачено', amount +
             ', чаевые', big_tip)
       Taxi.pay(self, amount + big_tip)

   def pour_drink(self):
       print('Налить напитки')

   def open_sunroof(self):
       print('Открыть люк')
       self.sunroof = 'открыт'

   def close_sunroof(self):
       print('Закрыть люк')
       self.sunroof = 'закрыт'
```

Проанализируйте работу программы и запишите предполагаемые результаты

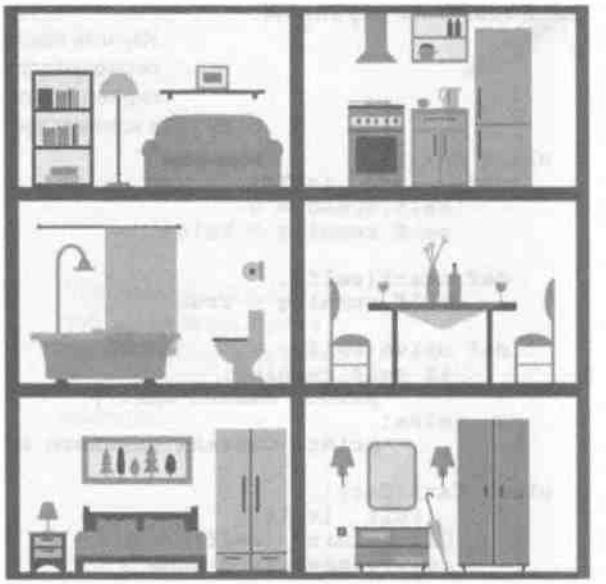
```
car = Car()
taxi = Taxi()
limo = Limo()

car.start()
car.drive()

taxi.start()
taxi.hire('Ким')
taxi.drive()
taxi.pay(5.0)

limo.start()
limo.hire('Джен')
taxi.drive()
limo.pour_drink()
limo.pay(10.0, 5.0)
```

Здесь внимательнее!

Запишите свои результаты здесь

Оболочка Python

Объекты внутри объектов



Атрибутами объектов могут быть не только простые типы данных, такие как числа или строки, но и более сложные структуры, в частности, списки и словари. Более того, в них даже могут храниться другие объекты. Такой тип отношений называется *включением*. Например, мы можем создать класс `House` (дом), атрибутом которого будет объект `Kitchen` (кухня). В таком случае мы говорим, что объект `House` включает объект `Kitchen`.

Специалисты по объектно-ориентированному программированию дополнительно уточняют подобные отношения, определяя, могут ли объекты существовать по отдельности. В конце концов, кухня не имеет смысла отдельно от дома, тогда как владелец дома – самостоятельная сущность. К данному вопросу мы еще вернемся позже.

Но зачем уделять этому внимание? Что такого необычного в использовании объекта в качестве атрибута другого объекта? Ведь это вполне естественно для Python.

Дело в том, что объекты наследуют поведение других объектов. Например, объект `ServiceDog` наследует от объекта `Dog` метод `human_years()` и часть поведения метода `bark()`. Но при включении объекта его поведение тоже наследуется! Возьмем объект `House`. Поскольку он включает объект `Kitchen`, у вас появляется возможность готовить еду!

Сейчас мы рассмотрим, как хранить объекты в других объектах, а затем узнаем, как добавлять поведение к другому объекту и даже делегировать ему часть полномочий.

Сила мысли

Какой набор классов потребуется для построения объектно-ориентированного дома?

За такую сложную терминологию стоит доплачивать

УПРАЖНЕНИЕ

Теперь ваша очередь создавать новый класс. Как насчет собаки, умеющей ловить диск-фрисби? Класс `Frisbee` мы уже создали за вас.

```
class Frisbee:
    def __init__(self, color):
        self.color = color

    def __str__(self):
        return 'Я ' + self.color + ' диск-фрисби'
```

У класса `Frisbee` не так много возможностей: у него есть атрибут цвета и метод `__str__()` для вывода готового приветствия

Это непростое задание. Наберитесь терпения и потратьте время, чтобы все тщательно обдумать. Решение приведено в конце главы.

Ваша задача — помочь нам завершить класс `FrisbeeDog`. Собака должна уметь ловить диск (метод `catch()`) и возвращать его хозяину (метод `give()`). Также нужно добавить метод `__str__()`. Ну и, конечно, придется переопределить метод `bark()`, поскольку собака не может лаять, пока держит фрисби.

```
class FrisbeeDog(Dog):
    def __init__(self, name, age, weight):
        Dog.__init__(self, name, age, weight)
        self.frisbee = None

    def bark(self):

    def catch(self, frisbee):

    def give(self):

    def __str__(self):
```

Собака класса `FrisbeeDog` должна лаять, как остальные собаки, если только она не держит в зубах фрисби. В таком случае она должна сообщить: "Не могу лаять, я держу фрисби"

Если вызывается метод `catch()`, необходимо сохранить переданный объект `frisbee` в атрибуте `frisbee`

Если вызван метод `give()`, необходимо задать атрибут `frisbee` равным `None` и вернуть прежний объект `frisbee`

Если собака держит фрисби, необходимо вернуть строку "Я собака по имени <имя>, и у меня есть фрисби". В противном случае нужно вернуть то же, что и все остальные собаки

Тест-драйв

Сверьте свое решение предыдущего упражнения с нашим, после чего добавьте новый код в файл *dog.py*. Удалите предыдущий код тестирования и вставьте вместо него следующий фрагмент (добавляется в конец файла).

```
def test_code():
    jim = FrisbeeDog('Джим', 5, 20)
    blue_frisbee = Frisbee('синий')

    print(jim)
    jim.bark()
    jim.catch(blue_frisbee)
    jim.bark()
    print(jim)
    frisbee = jim.give()
    print(frisbee)
    print(jim)

test_code()
```

Создаем объекты FrisbeeDog и Frisbee

Выводим информацию о собаке, заставляем ее лаять и ловить фрисби

Затем просим ее пролаять с фрисби в зубах

Выводим информацию о собаке (когда она держит диск) и заставляем ее отдать фрисби

Выводим информацию о фрисби и снова выводим информацию о собаке (теперь у нее нет фрисби)

Теперь код выглядит больше объектно-ориентированным, чем процедурным

Вот что мы получили

Оболочка Python

```
Я собака по имени Джим
Джим лает "гав-гав"
Джим поймал синий диск-фрисби
Джим говорит: "Не могу лаять, я держу фрисби"
Я собака по имени Джим, и у меня есть фрисби
Джим отдает синий диск-фрисби
Я синий диск-фрисби
Я собака по имени Джим
>>>
```

Джим — собака, умеющая ловить фрисби

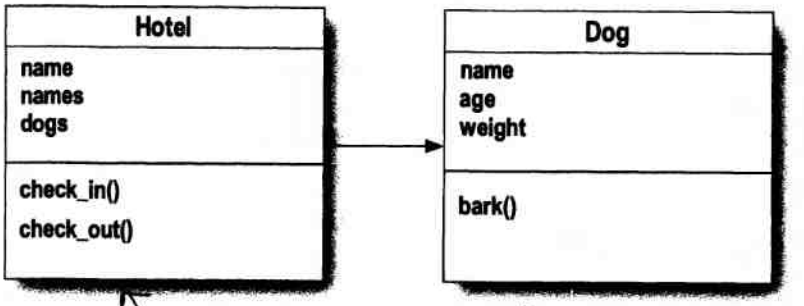
Создание отеля для собак

Интересные бизнес-проекты всегда были нашей слабостью. Недавно мы узнали, что появился серьезный спрос на отели для собак. Все, что нам нужно, – принять нескольких собак на передержку, позволить владельцам регистрировать и выписывать своих питомцев, ну и, конечно же, разрешить собакам лаять в свое удовольствие. Что ж, давайте приступим. Но сначала спланируем работу отеля, составив для него диаграмму классов.

У каждого отеля есть атрибут `name`*, в котором хранится название отеля*

У отеля также есть два атрибута, являющихся параллельными списками. В них хранятся имена собак и соответствующие им объекты `Dog`

У нас также есть два метода...



Метод `check_in()` *получает имя собаки вместе с объектом* `Dog` *и регистрирует собаку в отеле*

Метод `check_out()` *получает имя и возвращает объект* `Dog`*, если собака находилась в отеле*

Можно сказать, что объект `Hotel` *представляет собой набор объектов* `Dog`*, хранящихся в списке атрибутов* `dogs`

Прежде чем писать код класса `Hotel`, нужно определить для себя несколько важных моментов. Во-первых, регистрацию в отеле могут пройти только собаки. Чтобы считаться собакой, объект должен быть экземпляром класса `Dog` или его подкласса. Во-вторых, собаку можно выписать из отеля, только если она была в него заселена (этот факт следует проверять).

Сила мысли

Знаете ли вы лучший способ хранения данных о собаках, чем параллельные списки?

Реализация отеля для собак

Теперь, когда у нас есть диаграмма класса `Hotel`, займемся его реализацией. Начнем с создания конструктора, а затем напишем два метода: `check_in()` и `check_out()`.

```
class Hotel:
    def __init__(self, name):
        self.name = name
        self.kennel_names = []
        self.kennel_dogs = []

    def check_in(self, dog):
        if isinstance(dog, Dog):
            self.kennel_names.append(dog.name)
            self.kennel_dogs.append(dog)
            print(dog.name, 'заселился в', self.name)
        else:
            print('Извините,', self.name, 'только для собак')

    def check_out(self, name):
        for i in range(0, len(self.kennel_names)):
            if name == self.kennel_names[i]:
                dog = self.kennel_dogs[i]
                del self.kennel_names[i]
                del self.kennel_dogs[i]
                print(dog.name, 'выселился из', self.name)
                return dog
        print('Извините,', name, 'не обнаружен в', self.name)
        return None
```

Когда мы создаем экземпляр класса `Hotel`, мы присваиваем ему имя, например `'Отель "У пса"'`

Мы используем два списка: один для хранения имен собак и второй для хранения соответствующих объектов `Dog`

Метод для регистрации собак в отеле. В качестве аргумента передается объект `Dog`

Сначала убедимся, что это собака. Кошки и другие животные не допускаются

При заселении добавляем имя собаки и объект `Dog` в соответствующие списки...

...а также выводим сообщение для проверки

Если переданный объект не является собакой, не заселяем его

Чтобы выселить собаку из отеля, достаточно предоставить ее имя

Сначала убедимся, что собака заселена в отель...

...и, если это так, извлекаем объект `Dog` из списка, после чего удаляем имя и объект из списков атрибутов

Также возвращаем объект `Dog`. В конце концов, вы ведь хотите получить своего питомца обратно?

Если собаки нет в отеле, даем пользователю знать об этом и возвращаем `None`

Тест-драйв

Добавьте класс `Hotel` в файл *dog.py*. Удалите предыдущий код тестирования приложения, заменив его приведенным ниже кодом (добавляется в конец файла). Да, и не забудьте про класс `Cat`.

```
class Cat():
    def __init__(self, name):
        self.name = name

    def meow(self):
        print(self.name, 'говорит мяу')

def test_code():
    tuzik = Dog('Тузик', 12, 38)
    jackson = Dog('Джексон', 9, 12)
    smiley = Dog('Смайли', 2, 14)
    rudy = ServiceDog('Руди', 8, 38, 'Джозеф')
    jim = FrisbeeDog('Джим', 5, 20)
    kitty = Cat('Китти')

    hotel = Hotel('Отель "У пса"')
    hotel.check_in(tuzik)
    hotel.check_in(jackson)
    hotel.check_in(rudy)
    hotel.check_in(jim)
    hotel.check_in(kitty)

    dog = hotel.check_out(tuzik.name)
    print('Выселен', dog.name, ', возраст -', dog.age, ', вес -', dog.weight)
    dog = hotel.check_out(jackson.name)
    print('Выселен', dog.name, ', возраст -', dog.age, ', вес -', dog.weight)
    dog = hotel.check_out(rudy.name)
    print('Выселен', dog.name, ', возраст -', dog.age, ', вес -', dog.weight)
    dog = hotel.check_out(jim.name)
    print('Выселен', dog.name, ', возраст -', dog.age, ', вес -', dog.weight)
    dog = hotel.check_out(smiley.name)

test_code()
```

Это наш новый класс `Cat`

Создадим несколько собак общего типа `Dog` *и парочку более специфичных типов, таких как* `ServiceDog` *и* `FrisbeeDog`

Попробуем также создать кошку. Кто знает, вдруг повезет?

Создаем отель и заселяем в него питомцев

Теперь выселим животных и убедимся в том, что отель возвращает нам правильных собак

Проверим бдительность отеля: Смайли никогда не заселялся

Результат на следующей странице

Ну и ладно, не очень-то и хотелось!

Вот наши результаты

Заселились все...
...кроме кошки

Отель вернул нам правильных собак

И они сумели разобраться с проблемой Смайли

Оболочка Python

```
Тузик заселился в Отель "У пса"
Джексон заселился в Отель "У пса"
Руди заселился в Отель "У пса"
Джим заселился в Отель "У пса"
Извините, Отель "У пса" только для собак
Тузик выселился из Отель "У пса"
Выселен Тузик, возраст - 12, вес - 38
Джексон выселился из Отель "У пса"
Выселен Джексон, возраст - 9, вес - 12
Руди выселился из Отель "У пса"
Выселен Руди, возраст - 8, вес - 38
Джим выселился из Отель "У пса"
Выселен Джим, возраст - 5, вес - 20
Извините, Смайли не обнаружен в Отель "У пса"
>>>
```

Списки выглядят слишком громоздкими в данной программе. Мне кажется, словари стали бы намного более эффективным решением.

Вы правы. Мы подумали о том же самом. Нас остановило лишь то, что, как только начинаешь менять списки на словари, изменения касаются всего кода, и приходится переписывать программу целиком.

Но постойте-ка! У нас ведь объектно-ориентированный код. А одно из преимуществ ООП – **инкапсуляция** данных. Ее суть заключается в том, что объекты изолируют свое внутреннее состояние и поведение. Внешний код не видит того, что происходит внутри объекта, поэтому то, как мы реализуем объект, – наше личное дело. Способ реализации объекта не окажет влияния на работу остальной части программы.

Давайте исправим код отеля, и вы поймете, что мы имеем в виду.

Усовершенствование отеля для собак

Пора взяться за отбойный молоток и заняться перестройкой отеля! Нам нужно переписать методы `check_in()` и `check_out()`, изменение которых не повлияет на работу остальной программы. Сделать это несложно.

Когда мы создаем экземпляр класса Hotel, мы присваиваем ему имя, например 'Отель "У пса"'

```
class Hotel:
    def __init__(self, name):
        self.name = name
        ~~self.kennel_names = []~~
        ~~self.kennel_dogs = []~~
        self.kennel = {}
```

Теперь для хранения объектов собак будем использовать словарь, который назовем kennel

Метод для регистрации собак в отеле. В качестве аргумента передается объект Dog

Сначала убедимся, что это собака. Кошки и другие животные не допускаются

При заселении добавляем собаку в словарь, используя ее имя в качестве ключа...

...а также выводим сообщение для проверки

```
    def check_in(self, dog):
        if isinstance(dog, Dog):
            self.kennel[dog.name] = dog
            print(dog.name, 'заселился в', self.name)
        else:
            print('Извините,', self.name, 'только для собак')
```

Чтобы выселить собаку из отеля, достаточно предоставить ее имя

...и, если это так, извлекаем объект Dog из словаря, после чего удаляем собаку из словаря

Сначала убедимся, что собака заселена в отель...

```
    def check_out(self, name):
        if name in self.kennel:
            dog = self.kennel[name]
            print(dog.name, 'выселился из', self.name)
            del self.kennel[dog.name]
            return dog
        else:
            print('Извините,', name, 'не обнаружен в', self.name)
            return None
```

Также возвращаем объект Dog. В конце концов, вы ведь хотите получить своего питомца обратно?

Гораздо более понятный код!

Если собаки нет в отеле, даем пользователю знать об этом

Вы должны помнить из главы 8, что словари намного эффективнее списков в плане поиска элементов

Тест-драйв

Замените методы `check_in()` и `check_out()` в файле *dog.py* и протестируйте обновленное приложение.

Обратите внимание: мы не меняли тестовый код! Мы лишь изменили внутренние методы для регистрации и выселения собак. Так работает инкапсуляция. Другим программам не обязательно знать, как именно реализован класс Dog. Главное, чтобы интерфейс класса (доступные методы и атрибуты) оставался прежним

Оболочка Python

```
Тузик заселился в Отель "У пса"
Джексон заселился в Отель "У пса"
Руди заселился в Отель "У пса"
Джим заселился в Отель "У пса"
Извините, Отель "У пса" только для собак
Тузик выселился из Отель "У пса"
Выселен Тузик, возраст - 12, вес - 38
Джексон выселился из Отель "У пса"
Выселен Джексон, возраст - 9, вес - 12
Руди выселился из Отель "У пса"
Выселен Руди, возраст - 8, вес - 38
Джим выселился из Отель "У пса"
Выселен Джим, возраст - 5, вес - 20
Извините, Смайли не обнаружен в Отель "У пса"
>>>
```

Расширение функций отеля

Угадайте, чем больше всего любят заниматься собаки во время пребывания в отеле? Конечно же, лаять! Давайте снабдим класс `Hotel` методом `barktime()`, который разрешает всем собакам лаять.

Берем имя каждой собаки в словаре и используем его в качестве ключа для получения объекта Dog

```
def barktime(self):
    for dog_name in self.kennel:
        dog = self.kennel[dog_name]
        dog.bark()
```

Просим собаку залаять

Тест-драйв

Добавьте метод `barktime()` в файл *dog.py*. Удалите предыдущий код тестирования, заменив его приведенным ниже кодом (добавляется в конец файла), после чего запустите приложение.

```
def test_code():
    tuzik = Dog('Тузик', 12, 38)
    jackson = Dog('Джексон', 9, 12)
    rudy = ServiceDog('Руди', 8, 38, 'Джозеф')
    frisbee = Frisbee('красный')
    jim = FrisbeeDog('Джим', 5, 20)
    jim.catch(frisbee)

    hotel = Hotel('Отель "У пса"')
    hotel.check_in(tuzik)
    hotel.check_in(jackson)
    hotel.check_in(rudy)
    hotel.check_in(jim)

    hotel.barktime()

test_code()
```

Все наши собаки отозвались на команду "Голос"

Оболочка Python

```
Джим поймал красный диск-фрисби
Тузик заселился в Отель "У пса"
Джексон заселился в Отель "У пса"
Руди заселился в Отель "У пса"
Джим заселился в Отель "У пса"
Тузик лает "ГАВ-ГАВ"
Джексон лает "гав-гав"
Руди лает "ГАВ-ГАВ"
Джим говорит: "Не могу лаять, я держу фрисби"
>>>
```

В этой главе рассматривается множество новых концепций. Сделайте перерыв, поспите, дайте мозгу возможность усвоить материал. Сейчас самое время отдохнуть перед финальным рывком.

Мы с тобой одной крови: полиморфизм

Еще раз взглянем на результат вызова метода `barktime()` при последнем тестировании программы.

```
Тузик лает "ГАВ-ГАВ"
Джексон лает "гав-гав"
Руди лает "ГАВ-ГАВ"
Джим говорит: "Не могу лаять, я держу фрисби"
```

Тузик — это объект `Dog`

Джексон — тоже объект `Dog`

Руди — объект `ServiceDog`

А Джим — объект `FrisbeeDog`

С технической точки зрения все собаки относятся к разным классам (кроме Тузика и Джексона, которые являются объектами `Dog`). И тем не менее мы смогли написать простой код для просмотра всех объектов и вызова метода `bark()`.

```
for dog_name in self.kennel:
    dog = self.kennel[dog_name]
    dog.bark()
```

Метод `bark()` *вызывается независимо от того, какого типа собака*

А что было бы, если бы в классах `ServiceDog` и `FrisbeeDog` отсутствовал метод `bark()`? Как мы знаем, такого не может произойти, поскольку они являются подклассами класса `Dog`, а значит, наследуют от него метод `bark()`. В нашем случае метод `bark()` переопределяется в классах `ServiceDogs` и `FrisbeeDogs`, но это не имеет значения. Главное, что метод так или иначе имеется.

Почему это важно? Потому что это дает нам возможность всегда полагаться на наличие данного метода независимо от того, как он реализован в разных классах, даже если мы решим поменять алгоритмы их работы или создадим совершенно новые типы собак, которые изначально не были предусмотрены (например, `ShowDog` и `PoliceDog`). Фактически все собаки, относящиеся к типу `Dog` или его подтипам, будут поддерживаться методом `barktime()`, не требуя внесения каких-либо изменений в код работы отеля.

Со временем вы поймете, какое это огромное преимущество

У описанной технологии есть специальное название: **полиморфизм**. Это еще один устрашающий термин из лексикона объектно-ориентированного программирования. Полиморфизм означает, что один и тот же программный интерфейс (в нашем случае метод `bark()`) может по-разному реализовываться в разных классах (например, `FrisbeeDog` и `ServiceDog`). Это важная тема, которая требует глубокого изучения в контексте наследования. А пока достаточно знать, что можно писать универсальный код, применяемый к самым разным объектам при условии, что они поддерживают один и тот же набор методов.

Учимся выгуливать собак

Пока что выгулу у нас обучены только собаки-поводыри, но это неправильно, ведь выгуливать нужно всех собак. У нас есть основной класс `Dog`, а также подкласс `FrisbeeDog`, в которые следует добавить такую возможность. Нужно ли реализовывать метод `walk()` в обоих классах? Мы уже сталкивались с подобным, когда изучали метод `__str__()`. Поскольку класс `FrisbeeDog` наследует класс `Dog`, он унаследует от него и метод `walk()`. С учетом этого давайте немного переработаем программу, а заодно усовершенствуем класс `ServiceDog`, раз уж представилась такая возможность.

```
class Dog:
    def __init__(self, name, age, weight):
        self.name = name
        self.age = age
        self.weight = weight

    def bark(self):
        if self.weight > 29:
            print(self.name, 'лает "ГАВ-ГАВ"')
        else:
            print(self.name, 'лает "гав-гав"')

    def human_years(self):
        human_age = self.age * 7
        return human_age

    def walk(self):
        print(self.name, 'на прогулке')

    def __str__(self):
        return "Я собака по имени " + self.name

class ServiceDog(Dog):
    def __init__(self, name, age, weight, handler):
        Dog.__init__(self, name, age, weight)
        self.handler = handler
        self.is_working = False

    def walk(self):
        if self.is_working:
            print(self.name, 'помогает своему хозяину',
                  self.handler, 'ходить')
        else:
            Dog.walk(self)

    def bark(self):
        if self.is_working:
            print(self.name, 'говорит: "Не могу лаять, я работаю"')
        else:
            Dog.bark(self)
```

Показаны только изменившиеся классы

Добавим простой метод `walk()` *к классу* `Dog`

Для класса `ServiceDog`*, если собака работает, выводим специальное сообщение. В противном случае делаем то же, что и с остальными собаками*

Тест-драйв

Внесите рассмотренные изменения в файл *dog.py*. Удалите прежний код тестирования приложения и добавьте показанный ниже код в конец файла, после чего запустите программу.

```
def test_code():
    tuzik = Dog('Тузик', 12, 38)
    jackson = Dog('Джексон', 9, 12)
    rudy = ServiceDog('Руди', 8, 38, 'Джозеф')
    frisbee = Frisbee('красный')
    jim = FrisbeeDog('Джим', 5, 20)
    jim.catch(frisbee)

    tuzik.walk()
    jackson.walk()
    rudy.walk()
    jim.walk()

test_code()
```

Все собаки вышли на прогулку

Оболочка Python

```
Джим поймал красный диск-фрисби
Тузик на прогулке
Джексон на прогулке
Руди на прогулке
Джим на прогулке
>>>
```

УПРАЖНЕНИЕ

Переопределите метод `walk()` в классе `FrisbeeDog` так, чтобы собака, поймавшая фрисби, говорила: "Не могу гулять, я играю с фрисби". В остальном поведение этого класса должно повторять поведение класса `Dog`. Измените соответствующим образом файл *dog.py* и протестируйте полученный результат, воспользовавшись тем же кодом тестирования, что и в предыдущей врезке "Тест-драйв".

```
class FrisbeeDog(Dog):
    def __init__(self, name, age, weight):
        Dog.__init__(self, name, age, weight)
        self.frisbee = None

    def bark(self):
        if self.frisbee != None:
            print(self.name,
                  'говорит: "Не могу лаять, я держу фрисби"')
        else:
            Dog.bark(self)

     def walk():

    def catch(self, frisbee):
        self.frisbee = frisbee
        print(self.name, 'поймал', frisbee.color, 'диск-фрисби')

 :
```

Переопределите здесь метод `walk()`

Остальная часть класса `FrisbeeDog`

(и уязвимость)

Сила наследования

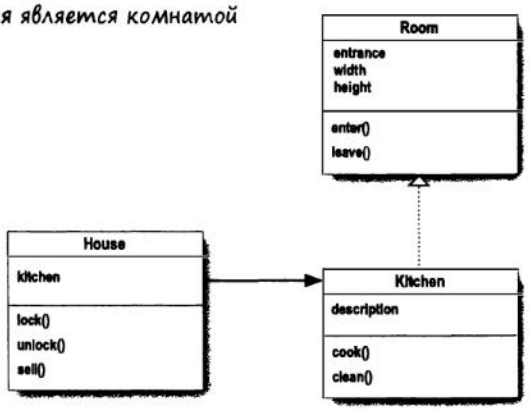
Мы всего-навсего добавили новый метод `walk()` в базовый класс `Dog`, и магическим образом остальные собаки (относящиеся к классам `ServiceDog` и `FrisbeeDog`) научились гулять, хотя раньше не умели. А все благодаря наследованию. Мы можем менять и расширять поведение целого набора классов, внося изменения в родительский класс, который они наследуют. Наследование — невероятно мощный инструмент, но пользоваться им нужно с осторожностью, так как можно столкнуться с непредсказуемыми последствиями при добавлении нового поведения. Что, если мы, например, добавим метод `chase_squirrel()` (преследовать белку) в класс `Dog`, не подумав о том, что это означает для собак класса `ServiceDog`?

Не стоит и чрезмерно усердствовать с наследованием при расширении функциональных возможностей классов, ведь для этого есть и другие способы. С одним из них мы уже знакомы: включение класса. Когда класс включается в атрибут другого класса, мы получаем более гибкую объектно-ориентированную модель, чем при использовании одного лишь наследования.

Для правильного применения наследования (в той степени, в какой оно оказывается эффективным и полезным) требуется хорошо понимать принципы объектно-ориентированного проектирования и уметь анализировать решаемые задачи. Все это приходит с опытом. Конечно, сейчас мы говорим о профессиональных тонкостях, но не помешает знать о них заранее. Многие программисты только спустя годы открывают для себя преимущества включения классов.

Повторимся: все эти нюансы станут вам понятны в будущем, по мере приобретения опыта. А пока вернемся к обустройству нашего отеля для собак и попробуем создать новые классы.

Вы уже знаете, как обозначается наследование с помощью диаграммы классов. В данном случае класс `Kitchen` наследуется от класса `Room`, то есть кухня является комнатой



Вот как обозначается включение класса. В данном случае класс `House` содержит объект класса `Kitchen`

Сила мысли

После того как данные о собаках в классе `Hotel` стали храниться в словаре, возникла одна неочевидная проблема. Как быть, если в отеле регистрируются две собаки с одинаковыми именами? Как решить эту проблему?

Служба выгула собак

Отель для собак не прочь подзаработать на дополнительных услугах, так почему бы не организовать при нем службу выгула заселившихся собак? Надеемся, идея сработает! Приступим к реализации.

```
class Hotel:
    def __init__(self, name):
        self.name = name
        self.kennel = {}

    def check_in(self, dog):
        if isinstance(dog, Dog):
            self.kennel[dog.name] = dog
            print(dog.name, 'заселился в', self.name)
        else:
            print('Извините,', self.name, 'только для собак')

    def check_out(self, name):
        if name in self.kennel:
            dog = self.kennel[name]
            print(dog.name, 'выселился из', self.name)
            del self.kennel[dog.name]
            return dog
        else:
            print('Извините,', name, 'не обнаружен в', self.name)
            return None

    def barktime(self):
        for dog_name in self.kennel:
            dog = self.kennel[dog_name]
            dog.bark()

    def walking_service(self):
        for dog_name in self.kennel:
            dog = self.kennel[dog_name]
            dog.walk()
```

Добавить службу выгула несложно. Все выглядит так же, как и в методе `barktime()`*, только теперь собаки гуляют, а не лают*

Проходим по каждому элементу словаря и вызываем метод `walk()` *каждой собаки*

Как видите, это было несложно, но представьте, что на выгул всех собак в отеле элементарно не хватает обслуживающего персонала. В конце концов, сотрудникам отеля есть чем заниматься и помимо этого. Чтобы не попасть впросак, нужно *делегировать* ответственность за выгул собак кому-то другому.

Как нанять человека для выгула собак

Это непростая задача, ведь у нас даже нет объекта, который описывал бы человека. Что ж, давайте исправим ситуацию. Мы создадим простой класс `Person` (человек), а затем напишем подкласс `DogWalker` (человек для выгула собак), в котором реализуем соответствующее поведение.

По-серьезному

В ООП, когда один объект поручает выполнение задачи другому объекту, это называется **делегированием**. Оно позволяет снабдить объект дополнительным поведением, не задействуя наследование и не реализуя поведение напрямую.

```
class Person:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return "Я человек, и меня зовут " + self.name
```

Добавим простой класс `Person` *с атрибутом* `name`

```
class DogWalker(Person):
    def __init__(self, name):
        Person.__init__(self, name)

    def walk_the_dogs(self, dogs):
        for dog_name in dogs:
            dogs[dog_name].walk()
```

`DogWalker` *— это класс типа* `Person`*, но с дополнительным методом* `walk_the_dogs()`

Этот метод вызывает метод `walk()` *каждой собаки*

Отлично, теперь перепишем класс `Hotel` так, чтобы в нем можно было нанять объект `DogWalker` и делегировать ему обязанности по выгулу собак с помощью метода `walking_service()`.

Эти методы добавляются в класс `Hotel`

В методе `hire_walker()` *необходимо убедиться, что переданный объект относится к классу* `DogWalker`*, и, если это так, нанять человека, добавив его в качестве атрибута*

```
def hire_walker(self, walker):
    if isinstance(walker, DogWalker):
        self.walker = walker
    else:
        print('Извините,', walker.name, 'не может выгуливать собак')

def walking_service(self):
    if self.walker != None:
        self.walker.walk_the_dogs(self.kennel)
```

Теперь в методе `walking_service()` *проверяем, установлен ли атрибут* `walker`*, и, если это так, просим человека выгулять собак*

Здравая мысль. Действительно, классу `DogWalker` известно о том, что данные о собаках хранятся в словаре или, точнее, в наборе вложенных словарей. Просто словарь — это типичная структура данных для решения подобных задач. Гипотетически, даже если бы для хранения данных о собаках применялась какая-то более сложная внутренняя структура, при передаче данных объекту `DogWalker` имело бы смысл упаковать их в словарь.

Но вы правы: если менять существующую внутреннюю реализацию отеля, то нужно учитывать, что класс `DogWalker` ожидает получения данных о собаках в формате словаря. В этом смысле нам не удалось обеспечить полную инкапсуляцию.

Чтобы скрыть реализацию отеля и разделить классы `Hotel` и `DogWalker`, нужно применить рассматривавшийся в предыдущих главах шаблон итератора, который позволяет просматривать последовательности значений, не зная ничего о том, как именно они реализованы.

Впрочем, решение этой задачи выходит за рамки книги. Но, так или иначе, проблема обозначена правильно, и ей следует уделять внимание при проектировании приложений.

Тест-драйв

Добавьте в файл *dog.py* классы `Person` и `DogWalker`, а в класс `Hotel` — методы `hire_walker()` и `walking_service()`. Замените код тестирования приложения следующим фрагментом.

```
def test_code():
    tuzik = Dog('Тузик', 12, 38)
    jackson = Dog('Джексон', 9, 12)
    smiley =  Dog('Смайли', 2, 14)
    rudy = ServiceDog('Руди', 8, 38, 'Джозеф')
    rudy.is_working = True
    jim =  FrisbeeDog('Джим', 5, 20)

    hotel = Hotel('Отель "У пса"')
    hotel.check_in(tuzik)
    hotel.check_in(jackson)
    hotel.check_in(rudy)
    hotel.check_in(jim)

    joe = DogWalker('Джо')
    hotel.hire_walker(joe)

    hotel.walking_service()

test_code()
```

Руди не работал бы в отеле, но все равно давайте попробуем

Создадим и наймем человека для выгула собак...

... и делегируем эти полномочия Джо

Похоже, все работает. Погнали на прогулку!

Объект Python

```
Тузик заселился в Отель "У пса"
Джексон заселился в Отель "У пса"
Руди заселился в Отель "У пса"
Джим заселился в Отель "У пса"
Тузик на прогулке
Джексон на прогулке
Руди помогает своему хозяину (Джозеф) ходить
Джим на прогулке
>>>
```

А тем временем на черепашьих бегах...

Помните, в главе 7 начали происходить странные вещи: зеленая черепашка стала все время побеждать, причем с огромным преимуществом? Полиции так и не удалось выяснить, почему это происходит. Теперь, когда вы приобрели достаточно знаний по ООП, давайте еще раз рассмотрим программу. Сможете определить, в чем там дело?

```
import turtle
import random

turtles = list()

class SuperTurtle(turtle.Turtle):
    def forward(self, distance)
        cheat_distance = distance + 5
        turtle.Turtle.forward(self, cheat_distance)

def setup():
    global turtles
    startline = -620
    screen = turtle.Screen()
    screen.setup(1290, 720)
    screen.bgpic('pavement.gif')

    turtle_ycor = [-40, -20, 0, 20, 40]
    turtle_color = ['blue', 'red', 'purple', 'brown', 'green']

    for i in = range(0, len(turtle_ycor)):
        if i == 4:
            new_turtle = SuperTurtle()
        else:
            new_turtle = turtle.Turtle()
        new_turtle.shape('turtle')
        new_turtle.penup()
        new_turtle.setpos(startline, turtle_ycor[i])
        new_turtle.color(turtle_color[i])
        new_turtle.pendown()
        turtles.append(new_turtle)

def race():
    global turtles
    winner = False
    finishline = 590

    while not winner:
        for current_turtle in turtles:
            move = random.randint(0, 2)
            current_turtle.forward(move)

            xcor = current_turtle.xcor()
            if (xcor >= finishline):
                winner = True
                winner_color = current_turtle.color()
                print('Победитель --', winner_color[0])
setup()
race()

turtle.mainloop()
```

Внимательно изучите код еще раз. Очевидно, что кто-то взломал программу, добавив в нее несколько новых строк. Что делает новый код? Какие принципы объектно-ориентированного программирования применялись при взломе?

Решение будет найдено через две страницы

CRIME SCENE DO NOT ENTER CRIME SCENE DO NOT ENTER CRIME SCENE DO NOT *ENTER*

Страна объектов

Как наладить жизнь благодаря объектам

Приветствуем тебя, путешественник! Дадим ряд советов о том, как приятно проводить время в стране объектов. Надеемся, тебе здесь понравится.

- Познакомься со всеми. Потрать время на изучение объектов (и классов), которые тебя окружают. Пощупи другой объектно-ориентированный код Python, чтобы понять, как он создан и как в нем используются объекты.
- Не стесняйся расширять встроенные классы Python. Относись к ним как к своим собственным классам.
- Приготовься к тому, что изучение объектно-ориентированного программирования — это на всю жизнь!
- Продолжай учиться. Основы ты уже знаешь, осталось набраться опыта. А единственный способ приобрести его — побольше практиковаться, непрерывно пополняя багаж знаний.
- Займись углубленным изучением принципов ООП, таких как наследование и полиморфизм.
- Старайся создавать простые и конкретные объекты, строя на их основе сложный код. Нужно практиковаться в строительстве маленьких домов, прежде чем переходить к небоскребам.
- Узнай побольше о включении объектов и принципах делегирования в коде. Это позволит сделать программы более гибкими.
- Не останавливайся на достигнутом. Ты выбрал правильный путь — не сворачивай с него!

Загадка черепашьих бегов РЕШЕНА

Отличная работа! Мы взяли его!

Разобрались, в чем дело? Хитрый хакер использовал свои знания ООП для создания **подкласса** `SuperTurtle` класса `Turtle`. После этого он **переопределил** в подклассе `SuperTurtle` метод `forward()` так, чтобы к параметру distance прибавлялись 5 единиц перед вызовом метода `forward()` **базового класса** (в данном случае `Turtle`). Хакер также продемонстрировал хорошие знания **полиморфизма**, разобравшись с тем, что в методе `race()` метод `forward()` вызывается для любого объекта, относящегося к классу `Turtle` и его потомкам. Оказывается, ООП может быть опасной штукой!

```
import turtle
import random

turtles = list()

class SuperTurtle(turtle.Turtle):
    def forward(self, distance):
        cheat_distance = distance + 5
        turtle.Turtle.forward(self, cheat_distance)

def setup():
    global turtles
    startline = -620
    screen = turtle.Screen()
    screen.setup(1290, 720)
    screen.bgpic('pavement.gif')

    turtle_ycor = [-40, -20, 0, 20, 40]
    turtle_color = ['blue', 'red', 'purple', 'brown', 'green']

    for i in range(0, len(turtle_ycor)):
        if i == 4:
            new_turtle = SuperTurtle()
        else:
            new_turtle = turtle.Turtle()
        new_turtle.shape('turtle')
        new_turtle.penup()
        new_turtle.setpos(startline, turtle_ycor[i])
        new_turtle.color(turtle_color[i])
        new_turtle.pendown()
        turtles.append(new_turtle)

def race():
    global turtles
    winner = False
    finishline = 590

    while not winner:
        for current_turtle in turtles:
            move = random.randint(0, 2)
            current_turtle.forward(move)

            xcor = current_turtle.xcor()
            if (xcor >= finishline):
                winner = True
                winner_color = current_turtle.color()
                print('Победитель --', winner_color[0])

setup()
race()

turtle.mainloop()
```

Определяем подкласс класса `Turtle`

Объект `SuperTurtle` *создается всякий раз для черепашки с индексом 4 (зеленая)*

Полиморфизм в действии: в этом коде метод `forward()` *вызывается для любого объекта типа* `Turtle`*, даже если это* `SuperTurtle`

Хакер создал подкласс класса `Turtle`, в котором переопределил метод `forward()`, прибавив 5 единиц к смещению черепашки.

CRIME SCENE DO NOT ENTER CRIME SCENE DO NOT ENTER CRIME SCENE DO NOT ENTER

Вы задумывались о карьере программиста?

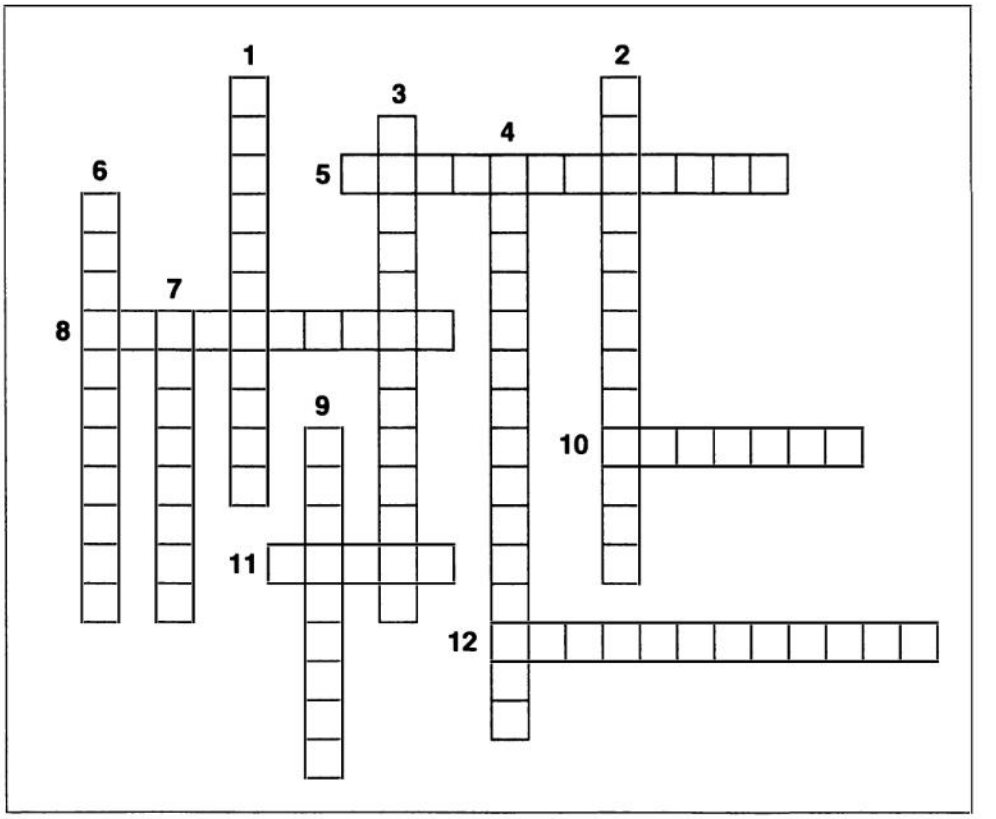
Если вы дочитали до этой страницы, ничего не пропустив, то примите наши поздравления! Спросите себя, насколько далеко вы готовы пойти дальше. Возможно, вы еще не осознали, как много прочитали. Но не забывайте: данная книга – лишь вершина айсберга в том, что касается разработки программного обеспечения. Как бы там ни было, советуем серьезно отнестись к предложению, вынесенному в заголовок, ведь очевидно, что у вас есть все необходимое для успешной карьеры!

САМОЕ ГЛАВНОЕ

- В объектно-ориентированном программировании (ООП) решение задачи сводится к моделированию поведения реальных (или виртуальных) объектов.
- У объектов есть состояние (атрибуты) и поведение (методы).
- Объекты создаются на основе классов, которые служат в качестве шаблонов.
- При создании нового объекта мы получаем экземпляр класса.
- Конструктор — это метод, создающий и инициализирующий экземпляр класса.
- В Python конструктор называется `__init__()`.
- В качестве первого аргумента конструктору передается создаваемый объект.
- По соглашению первый параметр конструктора называется `self`.
- Атрибуты объекта подобны локальным переменным и могут содержать значения любых допустимых типов данных Python.
- Методы объекта подобны функциям, за исключением того, что в качестве первого аргумента всегда передается объект `self`.
- Атрибуты и методы можно унаследовать от другого класса, создав его подкласс.
- Класс, для которого создается подкласс, называется родительским, или суперклассом.
- Методы суперкласса можно переопределять, создавая в подклассах методы с такими же именами.
- Объект подкласса также является объектом родительского класса.
- Проверить принадлежность к классу можно с помощью встроенной функции `isinstance()`.
- Функция `isinstance()` возвращает `True`, если объект является экземпляром указанного класса (или одного из его суперклассов).
- Переопределите метод `__str__()`, чтобы задать строку, которая должна выводиться на экран при передаче объекта функции `print()`.
- Полиморфизм подразумевает написание кода, в котором один и тот же программный интерфейс реализуется разными объектами.
- Интерфейсом объекта являются методы, которые в нем реализованы и которые можно вызывать.
- Объект может быть атрибутом другого объекта.
- Включение другого объекта в качестве атрибута — стандартный прием для расширения поведения класса.
- Переадресация вызова методу другого класса называется делегированием.
- Множественное наследование имеет место, когда объект наследует состояние и поведение сразу от нескольких классов.

Кроссворд

Это последний кроссворд в книге, и он объектно-ориентированный! По крайней мере, все слова в нем относятся к лексикону ООП.



По горизонтали

5. Изоляция состояния и поведения внутри класса
8. Родительский класс
10. Переменная, определяющая состояние объекта
11. Функция, определяющая поведение объекта
12. Получение состояния и поведения от родительского класса

По вертикали

1. Реализация одного и того же метода в разных классах
2. Передача вызова другому объекту
3. Тип наследования, когда методы заимствуются сразу от нескольких классов
4. Замена метода в другом классе
6. Метод, создающий объект
7. Дочерний класс
9. Образец класса

Возьмите карандаш
Решение

В главе 2 мы написали программу, которая определяла возраст собаки по человеческим меркам. Добавьте в наш класс `Dog` метод, реализующий такие вычисления, и назовите его `human_years()`. Метод не требует аргументов и возвращает целочисленный результат.

Это наш код на данный момент. Добавьте метод `human_years()`*, возвращающий возраст собаки по человеческим меркам*

```
class Dog:
    def __init__(self, name, age, weight):
        self.name = name
        self.age = age
        self.weight = weight

    def bark(self):
        if self.weight > 29:
            print(self.name, 'лает "ГАВ-ГАВ"')
        else:
            print(self.name, 'лает "гав-гав"')

    def human_years(self):
        years = self.age * 7
        return years

def print_dog(dog):
    print(dog.name + ': возраст -', dog.age,
                     ', вес -', dog.weight)

tuzik = Dog('Тузик', 12, 38)
jackson = Dog('Джексон', 9, 12)
print(tuzik.name + ": возраст по меркам людей -", tuzik.human_years())
print(jackson.name + ": возраст по меркам людей -", jackson.human_years())
```

Чтобы вычислить возраст собаки по человеческим меркам, необходимо умножить атрибут `age` *на 7*

Вот наши результаты

Оболочка Python

```
Тузик: возраст по меркам людей - 84
Джексон: возраст по меркам людей - 63
>>>
```

Возьмите карандаш
Решение

Используя показанную слева диаграмму классов, укажите справа значения, возвращаемые функцией `isinstance()`. Помните, что эта функция может возвращать только `True` или `False`. Первое задание мы выполнили за вас.

Ваш ответ: True или False

```
simple_cake = Cake()
chocolate_cake = FrostedCake()
bills_birthday_cake = BirthdayCake()
```

False	`isinstance(simple_cake, BirthdayCake)`
False	`isinstance(simple_cake, FrostedCake)`
True	`isinstance(simple_cake, Cake)`
True	`isinstance(chocolate_cake, Cake)`
True	`isinstance(chocolate_cake, FrostedCake)`
False	`isinstance(chocolate_cake, BirthdayCake)`
True	`isinstance(bills_birthday_cake, FrostedCake)`
True	`isinstance(bills_birthday_cake, Cake)`
True	`isinstance(bills_birthday_cake, BirthdayCake)`

Возьмите карандаш

Решение

Изучите показанный слева код классов, включающий несколько переопределяемых методов. Мысленно выполните код, приведенный справа, и запишите предполагаемые результаты.

```
class Car():
   def __init__(self):
       self.speed = 0
       self.running = False

   def start(self):
       self.running = True

   def drive(self):
       if self.running:
           print('Машина едет')
       else:
           print('Сначала заведите машину')

class Taxi(Car):
   def __init__(self):
       Car.__init__(self)
       self.passenger = None
       self.balance = 0.0

   def drive(self):
       print('Бип-бип, с дороги!')
       Car.drive(self)

   def hire(self, passenger):
       print('Заказано', passenger)
       self.passenger = passenger

   def pay(self, amount):
       print('Заплачено', amount)
       self.balance = self.balance + amount
       self.passenger = None

class Limo(Taxi):
   def __init__(self):
       Taxi.__init__(self)
       self.sunroof = 'закрыт'

   def drive(self):
       print('Лимузин - не роскошь')
       Car.drive(self)

   def pay(self, amount, big_tip):
       print('Заплачено', amount +
             ', чаевые', big_tip)
       Taxi.pay(self, amount + big_tip)

   def pour_drink(self):
       print('Налить напитки')

   def open_sunroof(self):
       print('Открыть люк')
       self.sunroof = 'открыт'

   def close_sunroof(self):
       print('Закрыть люк')
       self.sunroof = 'закрыт'
```

Проанализируйте работу программы и запишите предполагаемые результаты

```
car = Car()
taxi = Taxi()
limo = Limo()

car.start()
car.drive()

taxi.start()
taxi.hire('Ким')
taxi.drive()
taxi.pay(5.0)

limo.start()
limo.hire('Джен')
taxi.drive()
limo.pour_drink()
limo.pay(10.0, 5.0)
```

Здесь внимательнее!

Запишите свои результаты здесь

Оболочка Python

```
Машина едет
Заказано Ким
Бип-бип, с дороги!
Машина едет
Заплачено 5.0
Заказано Джен
Бип-бип, с дороги!
Машина едет
Налить напитки
Заплачено 10.0, чаевые 5.0
Заплачено 15.0
>>>
```

УПРАЖНЕНИЕ (РЕШЕНИЕ)

Теперь ваша очередь создавать новый класс. Как насчет собаки, умеющей ловить диск-фрисби? Класс `Frisbee` мы уже создали за вас.

```
class Frisbee:
    def __init__(self, color):
        self.color = color

    def __str__(self):
        return 'Я ' + self.color + ' диск-фрисби'
```

У класса `Frisbee` не так много возможностей: у него есть атрибут цвета и метод `__str__()` для вывода готового приветствия

Ваша задача — помочь нам завершить класс `FrisbeeDog`. Собака должна уметь ловить диск (метод `catch()`) и возвращать его хозяину (метод `give()`). Также нужно добавить метод `__str__()`. Ну и, конечно, придется переопределить метод `bark()`, поскольку собака не может лаять, пока держит фрисби.

```
class FrisbeeDog(Dog):
    def __init__(self, name, age, weight):
        Dog.__init__(self, name, age, weight)
        self.frisbee = None

    def bark(self):
        if self.frisbee != None:
            print(self.name,
                  'говорит: "Не могу лаять, я держу фрисби"')
        else:
            Dog.bark(self)

    def catch(self, frisbee):
        self.frisbee = frisbee
        print(self.name, 'поймал', frisbee.color, 'диск-фрисби')

    def give(self):
        if self.frisbee != None:
            frisbee = self.frisbee
            self.frisbee = None
            print(self.name, 'отдает', frisbee.color, 'диск-фрисби')
            return frisbee
        else:
            print(self.name, "не держит фрисби")
            return None

    def __str__(self):
        str = "Я собака по имени " + self.name
        if self.frisbee != None:
            str = str + ', и у меня есть фрисби'
        return str
```

У нас простой конструктор, в котором всего лишь устанавливается атрибут `frisbee`. Сам по себе `frisbee` — это другой объект, который включается в объект `FrisbeeDog`

Переопределяем метод `bark()`. Если собака в данный момент держит фрисби, то она не может лаять, в противном случае она лает, как и остальные собаки

Метод `catch()` получает объект `frisbee` и записывает его в атрибут `frisbee` текущего объекта

Метод `give()` задает атрибут `frisbee` равным `None` и возвращает прежний объект `frisbee`

Метод `__str__()` формирует разное приветствие в зависимости от того, держит собака фрисби или нет

УПРАЖНЕНИЕ (РЕШЕНИЕ)

Переопределите метод `walk()` в классе `FrisbeeDog` так, чтобы собака, поймавшая фрисби, говорила: "Не могу гулять, я играю с фрисби". В остальном поведение этого класса должно повторять поведение класса `Dog`. Измените соответствующим образом файл *dog.py* и протестируйте полученный результат, воспользовавшись тем же кодом тестирования, что и в предыдущей врезке "Тест-драйв".

```
class FrisbeeDog(Dog):
    def __init__(self, name, age, weight):
        Dog.__init__(self, name, age, weight)
        self.frisbee = None

    def bark(self):
        if self.frisbee != None:
            print(self.name,
                  'говорит: "Не могу лаять, я держу фрисби"')
        else:
            Dog.bark(self)

    def walk(self):
        if self.frisbee != None:
            print(self.name, 'говорит: "Не могу гулять, я играю с фрисби!"')
        else:
            Dog.walk(self)

    def catch(self, frisbee):
        self.frisbee = frisbee
        print(self.name, 'поймал', frisbee.color, 'диск-фрисби')

    def give(self):
        if self.frisbee != None:
            frisbee = self.frisbee
            self.frisbee = None
            print(self.name, 'отдает', frisbee.color, 'диск-фрисби')
            return frisbee
        else:
            print(self.name, "не держит фрисби")
            return None

    def __str__(self):
        str = "Я собака по имени " + self.name
        if self.frisbee != None:
            str = str + ', и у меня есть фрисби'
        return str
```

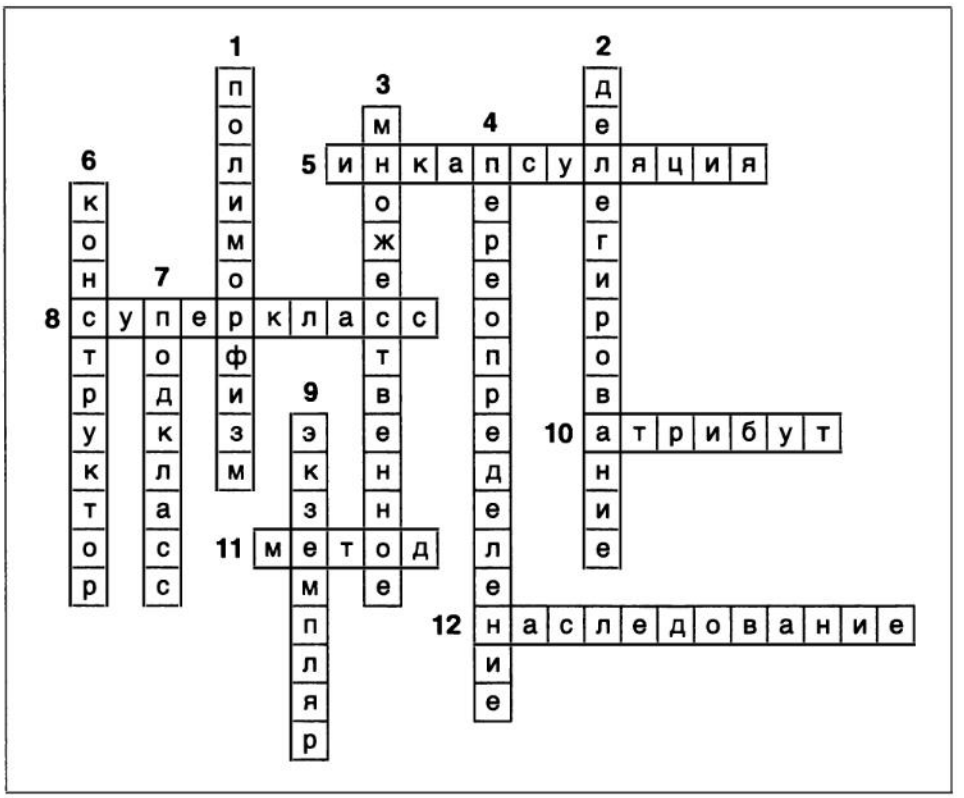
Собака держит фрисби, если атрибут `self.frisbee` *не равен* `None`

Если собака держит фрисби, то сообщаем, что она играет. В противном случае делаем то же, что и с остальными собаками, вызывая метод `walk()` *суперкласса*

Оболочка Python

```
Джим поймал красный диск-фрисби
Тузик на прогулке
Джексон на прогулке
Руди на прогулке
Джим говорит: "Не могу гулять, я играю с фрисби!"
>>>
```

Кроссворд: решение



приложение: не попавшее в избранное

Десять ключевых тем (которые не были рассмотрены)

Вы прошли длинный путь и находитесь у финишной черты. Нам искренне жаль расставаться с вами, и, прежде чем отправить вас в самостоятельное плавание, нам бы хотелось дать последние наставления. Конечно, в одном маленьком приложении невозможно охватить все необходимое, хотя мы попытались это сделать, уменьшив размер шрифта до 0,00004 пункта. К сожалению, никому в издательстве так и не удалось прочитать столь микроскопический текст, поэтому нам пришлось выкинуть все лишнее, оставив только самое важное.

1. Списковые включения

Вы уже знаете, как создать список чисел с помощью функции `range()`. Тем не менее в Python существует более эффективный способ создания списков, основанный на элементах теории множеств. Он называется *списковым включением* и позволяет создавать списки любого типа. Сначала рассмотрим список чисел.

```
[x + x for x in range(10)]
```

Удваиваем каждое число в диапазоне от 0 до 9

Оболочка Python

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
>>>
```

А теперь создадим список строк.

```
lyric = ['I', 'saw', 'heard', 'on', 'you', 'the', 'wireless', 'back', 'in', '52']
[s[0] for s in lyric]
```

Берем первую букву каждого слова в списке `lyric`

Оболочка Python

```
['I', 's', 'h', 'o', 'y', 't', 'w', 'b', 'i', '5']
>>>
```

По сути, список генерируется на основе уже существующего списка. Чтобы понять, как это происходит, рассмотрим формат спискового включения.

Первая часть — это выражение с использованием переменной, которая представляет каждый элемент существующего списка

Далее идет выражение `for` с указанием переменной и списка

Наконец, указывается условие (такой вариант нам еще не встречался)

[*выражение* **for** *элемент* **in** *список* **if** *условие*]

При необходимости списковое включение можно снабдить условием.

```
[s[0] for s in lyric if s[0] > 'm']
```

Добавляем элемент в список, только если это буква, которая идет в алфавите после 'm'

Оболочка Python

```
['s', 'o', 'y', 't', 'w']
>>>
```

Чтобы освоить данную конструкцию, как и многое другое в этом приложении, придется хорошенько попрактиковаться. Но вы быстро поймете, что это очень удобный способ создания новых списков.

2. Работа с датой и временем

Со значениями даты и времени часто приходится работать в программах. В Python для этого имеется модуль `datetime`, который импортируется так:

```
import datetime
```

После импорта модуля можно создать объект `date`, инициализировав его любой датой вплоть до 9999 года (отсчет ведется от года 1).

Сначала указывается год, затем месяц и день

```
my_date = datetime.date(2015, 10, 21)
```

Кроме того, можно создать объект `time`, указав время в формате "час, минута, секунда".

```
my_time = datetime.time(7, 28, 1)
```

Час, минута, секунда

Комбинируя их вместе, получаем объект `datetime`.

Комбинация объектов `date` и `time`

```
my_datetime = datetime.datetime(2015, 10, 21, 7, 28, 1)
```

Давайте выведем все имеющиеся значения.

```
print(my_date)
print(my_time)
print(my_datetime)
print(my_date.year, my_date.month, my_date.day)
print(my_time.hour, my_time.minute, my_time.second)
```

Выводим каждый объект даты и времени, чтобы узнать его содержимое

Оболочка Python

```
2015-10-21
07:28:01
2015-10-21 07:28:01
2015 10 21
7 28 1
>>>
```

Вот как можно узнать текущее время.

```
now = datetime.datetime.today()
print(now)
```

Текущее время можно определить с точностью до миллисекунд

Оболочка Python

```
2017-07-27 19:12:07.785931
```

Кроме того, модуль `datetime` поддерживает расширенное форматирование.

```
output = '{:%A, %B %d, %Y}'
print(output.format(my_date))
```

Объекты даты и времени поддерживают расширенное форматирование

Оболочка Python

```
Wednesday, October 21, 2015
```

Все это — только основы: в каждом языке программирования имеются средства работы с датой и временем. Для детального знакомства с ними в Python изучите документацию к модулю `datetime` и другим аналогичным модулям.

3. Регулярные выражения

'ac*\dc?'

Вспомните, с какими трудностями мы столкнулись при обработке текста, содержащего знаки препинания. Решением проблемы могли бы стать *регулярные выражения* — формальный язык описания поисковых шаблонов. Например, используя регулярное выражение, можно составить шаблон, который будет соответствовать любой подстроке, начинающейся с буквы `t`, заканчивающейся буквой `e`, содержащей не менее одной буквы `a` и не более двух строк `us`.

Регулярные выражения могут быть чрезвычайно сложными. Для непосвященных они выглядят словно язык инопланетян. Но если начать с простых поисковых шаблонов и хорошенько разобраться в них, то со временем можно стать настоящим экспертом.

Регулярные выражения поддерживаются в большинстве современных языков программирования, и Python — не исключение. Вот как применяются регулярные выражения в Python.

Оболочка Python

```
Найдено совпадение между: (19, 23)
Найдено совпадение между: (19, 23)
Совпадений не найдено
>>>
```

```
import re
for term in ['I heard you on the wireless back in 52',
             'I heard you on the Wireless back in 52',
             'I heard you on the WIRELESS back in 52']:
    result = re.search('[wW]ire', term)
    if result:
        loc = result.span()

        print('Найдено совпадение между:', loc)
    else:
        print('Совпадений не найдено')
```

Импортируем модуль регулярных выражений (`re`)

Протестируем несколько строк...

Поиск с использованием регулярного выражения

Это регулярное выражение, которое соответствует строке 'wire' или 'Wire'

Если переменная `result` не пуста, значит, есть совпадение, и метод `span()` сообщает позицию подстроки

Сообщаем позицию найденной подстроки...

...или сообщаем об отсутствии совпадений

В данном примере используется простое регулярное выражение, но поисковые шаблоны могут быть гораздо более сложными, ведь регулярные выражения применяются для нахождения имен пользователей, паролей, URL-адресов и т.п. Когда в следующий раз столкнетесь с необходимостью написать функцию для проверки имени пользователя или чего-то похожего, обратитесь к регулярным выражениям — это позволит обойтись всего несколькими строками кода.

Поиск совпадений — не единственный способ применения регулярных выражений. В модуле `re` собраны достаточно сложные инструменты обработки поисковых шаблонов, которые могут пригодиться при разработке приложений.

Для эффективной работы с регулярными выражениями нужно научиться их создавать и анализировать. Это потребует изучения общих принципов регулярных выражений, а также специального синтаксиса, применяемого в Python.

4. Другие типы данных: кортежи

У списков Python есть родственный тип данных, который не упоминался в книге: *кортеж*. Эти структуры данных схожи синтаксически, что хорошо видно по следующему примеру.

```
my_list = ['Back to the Future', 'TRON', 'Buckaroo Banzai']

my_tuple = ('Back to the Future', 'TRON', 'Buckaroo Banzai')
```

Список фильмов, а фактически список строк

Строковый кортеж

Так в чем же разница? С точки зрения синтаксиса в первом случае используются квадратные скобки, а во втором — круглые

В циклах они используются одинаковым образом.

```
for movie in my_list:
    print(movie)
for movie in my_tuple:
    print(movie)
```

Результат одинаковый!

Оболочка Python

```
Back to the Future
TRON
Buckaroo Banzai
Back to the Future
TRON
Buckaroo Banzai
```

И конечно же, на элемент кортежа тоже можно сослаться по индексу, например `my_tuple[2]`, что соответствует элементу `'Buckaroo Bonzai'`. Кортежи поддерживают почти те же методы, что и списки, так в чем же разница? В том, что кортеж — *неизменяемый тип данных*, в отличие от списка. После того как кортеж создан, он больше не подлежит изменению — можно лишь получать значения его элементов.

В таком случае зачем нужны кортежи? Для чего они существуют? Для экономии *времени* и *памяти*. Кортеж занимает меньше места в памяти и обрабатывается быстрее, чем список. Если в программе задействована ресурсоемкая коллекция элементов, с которыми планируется выполнять большое число операций, то имеет смысл применить именно кортеж, чтобы ускорить вычисления и сократить потребление ресурсов.

С вычислительной точки зрения безопаснее работать со структурой данных, которая не может быть изменена

Кроме того, кортежи поддерживают расширенный синтаксис, недоступный для списков.

```
x, y, z = (1, 2, 3)

apples = 'mac', 'red', 'green'

single = (3.14159265,)
```

Это называется распаковкой: Python назначает каждой переменной соответствующий элемент кортежа

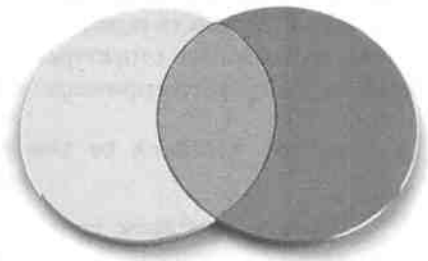
Даже без круглых скобок элементы, разделенные запятыми, становятся кортежем

Будьте внимательны: в операции распаковки требуется, чтобы при наличии кортежа, состоящего из одного элемента, этот элемент заканчивался запятой

Дополнительная запятая

5. Другие типы данных: множества

Мы не обсуждали еще один популярный тип данных Python: *множество*. Он должен быть хорошо вам знаком по школьному курсу алгебры. Множества не содержат повторяющихся значений и не упорядочены. Кроме того, над множествами можно выполнять различные операции, такие как объединение и пересечение. Рассмотрим, как объявляются множества в Python.



Множество представляет собой список разделенных запятыми значений, заключенных в фигурные скобки

```
set = {1, 3.14159264, False, 77}
```

У словаря в фигурных скобках находятся пары ключ/значение, а у множества — только значения

Множества могут хранить значения любых типов, главное, чтобы все значения были уникальными

Множество, как и список, — изменяемый тип данных. Например, в него можно добавить значение:

```
set.add(99)
```

или удалить:

```
set.remove(1)
```

Множество также поддерживает ряд интересных методов.

В Python пустое множество (не содержащее элементов) представляется как `set()`

```
even = {2, 4, 6, 8, 10}
odd = {1, 3, 5, 7, 9}
prime = {1, 3, 5, 7}

even_and_prime = even.intersection(prime)
print(even_and_prime)

odd_and_prime = odd.intersection(prime)
print(odd_and_prime)

even_or_prime = even.union(prime)
print(even_or_prime)
```

Оболочка Python

```
set()
{1, 3, 5, 7}
{1, 2, 3, 4, 5, 6, 7, 8, 10}
>>>
```

Проверка существования значения в множестве выполняется с помощью уже знакомого вам синтаксиса `if x in set`. С остальными операциями, такими как разность и симметричная разность, можно ознакомиться в документации Python.

6. Написание серверного кода

Многие приложения работают в виде веб-служб, предоставляя услуги поиска в Google, подключения к социальным сетями и платежным системам. Такие приложения пишутся на многих языках, включая Python.

Изучение принципов написания серверных приложений требует знания многих технологий и протоколов, включая HTTP (HyperText Transport Protocol – протокол передачи гипертекста), HTML (разметка веб-страниц), JSON (формат передачи данных) и др. Кроме того, в зависимости от выбранного языка программирования, вам понадобится использовать один из веб-фреймворков. Такие пакеты содержат функции, обеспечивающие низкоуровневую обработку данных и веб-страниц.

В случае Python два самых популярных фреймворка – Flask и Django.

```
@app.route("/")
def hello():
  return "Привет, Интернет!"
```

Короткий пример синтаксиса Flask. С помощью Flask можно перенаправлять трафик из корневого каталога (домашней страницы) сайта в функцию `hello()`. Сама функция `hello()` просто выводит строку

Обычно здесь выводится строка в формате HTML или JSON

Flask – это минималистичный фреймворк, ориентированный на небольшие проекты, которые нужно запустить побыстрее. В противоположность ему Django – полнофункциональный фреймворк для больших проектов, на изучение которого уйдет немало времени. Он позволяет работать с шаблонами веб-страниц и формами, выполнять аутентификацию и администрировать базы данных. Поскольку вы только начинаете свое знакомство с технологиями веб-разработки, мы рекомендуем Flask как фреймворк начального уровня. К Django можно будет перейти позже, по мере приобретения опыта.

7. Отложенные вычисления

Предположим, имеется такой код.

```
def nth_prime(n):
    count = 0
    for prime in list_of_primes():
        count = count + 1
        if count == n:
            return prime

def list_of_primes():
    primes = []
    next = 1
    while True:
        next_prime = get_prime(next)
        next = next + 1
        primes.append(next_prime)
    return primes
```

Эта функция возвращает n-е простое число...

...из списка простых чисел

Можно задать верхний предел, например 1000, но это тоже будет очень неэффективно

Очевидно, что так делать нельзя, ведь создание бесконечного списка простых чисел длится вечно

Эту функцию мы оставляем в качестве домашнего задания (или просто погуглите)

Более эффективное решение основано на методике, которая называется *отложенные вычисления*, или *вычисления по запросу*. В Python они поддерживаются благодаря шаблону генератора, который создается следующим образом.

```
def list_of_primes():
    next = 1
    while True:
        next_prime = get_prime(next)
        next = next + 1
        yield next_prime
```

Нам больше не нужен массив primes, поэтому мы его убрали

Теперь всякий раз, когда генерируется следующее простое число, мы выполняем инструкцию yield

Что же делает инструкция `yield`? Давайте разберемся. В функции `nth_prime()` мы проходим по списку всех простых чисел. Когда в цикле `for` впервые вызывается функция `list_of_primes()` и выполняется инструкция `yield`, создается генератор (во многих языках он называется *итератором*). Это объект, у которого всего один метод – `next()`, вызываемый (неявно в случае цикла `for`) для получения следующего значения. При каждом последующем вызове метода `next()` функция `list_of_primes()` продолжает вычисления, вместо того чтобы начинать все сначала, и возвращает очередное число. Так продолжается до тех пор, пока программа не перестанет запрашивать новые значения.

Отложенные вычисления – невероятно эффективный инструмент программирования, который заслуживает углубленного изучения.

8. Декораторы

Декоратор — один из шаблонов проектирования объектно-ориентированных приложений. В Python декораторы применяются в основном для расширения возможностей одних функций за счет других. Например, если имеется функция, возвращающая простой текст, то можно создать набор декораторов, добавляющих к тексту HTML-разметку. Предположим, что в возвращаемый текст нужно добавить теги абзацев. С помощью декораторов это делается так.

*В HTML, чтобы создать абзац, необходимо заключить текст между тегами и *

```
def paragraph(func):
    def add_markup():
        return '<p>' + func() + '</p>'
    return add_markup

@paragraph
def get_text():
    return 'Привет, читатель!'

print(get_text())
```

*Здесь функции используются как объекты первого класса. Сначала мы передаем функцию func() в качестве аргумента, после чего создаем и возвращаем другую функцию, которая обрамляет результаты, возвращаемые функцией func(), тегами и *

Используем синтаксис @ для декорирования другой функции, в данном случае get_text()

Теперь при обращении к функции get_text() она вызывается внутри декоратора, что приводит к добавлению тегов и к возвращаемым ею результатам

Как видите, появление декоратора меняет ход выполнения программы. Чтобы понять логику работы такого кода, необходимо дополнительно изучить концепцию объектов первого класса.

Декораторы — мощный инструмент, применяемый не только для форматирования текста. Их можно добавлять к любым функциям, которым нужно придать дополнительное поведение без изменения их кода.

Оболочка Python

```
<p>Привет, читатель!</p>
```

9. Объекты первого класса и функции высшего порядка

Вы хорошо знаете, что такое функции. Мы рассматривали их как инструмент создания абстракций в коде, но этим их применение не ограничивается. Фактически с функциями можно выполнять те же самые операции, что и с другими типами данных: присваивать переменным, передавать другим функциям в качестве аргументов и даже возвращать из других функций. Когда функция интерпретируется как объект или тип данных, она называется *объектом первого класса*, а когда она передается как аргумент или возвращается из другой функции, она называется *функцией высшего порядка*. Но как можно передавать или возвращать функцию?

Описание функций высшего порядка — тема для отдельной книги, поэтому мы лишь рассмотрим несколько примеров их применения. Вот как вернуть функцию из другой функции.

```
def pluralize(str):
    def helper(word):
        return word + "s"
    return helper(str)

val = pluralize('girl')
print(val)
```

А вы знали, что можно объявить функцию внутри другой функции?

Функция helper() доступна только внутри функции pluralize(); из другого кода ее вызвать невозможно

Функция pluralize() использует функцию helper(), чтобы добавить окончание "s" к слову

Оболочка Python

```
girls
>>>
```

Интересная возможность, но можно поступить еще интереснее.

```
def addition_maker(n):
    def maker(x):
        return n + x
    return maker

add_two = addition_maker(2)

val = add_two(1)
print(val)
```

Функция addition_maker() получает число n...

...и определяет функцию, которая прибавляет к n другое число, x

Созданная функция возвращается как результат вызова функции addition_maker()

Если мы передадим функции addition_maker() число 2, она вернет функцию, которая записывается в переменную add_two...

...и при вызове эта функция прибавляет 2 к любому переданному ей аргументу, в данном случае 1

Оболочка Python

```
3
>>>
```

Если вы не привыкли к такому стилю программирования, то вы не одиноки! Но, как и в случае с рекурсией, немного практики — и привычка создавать функции высшего порядка станет вашей второй натурой.

10. Важные библиотеки

В процессе обучения вам приходилось использовать некоторые библиотеки Python. Но в вашем распоряжении находится намного больше библиотек, как встроенных, так и сторонних. Ниже перечислено несколько популярных библиотек.

requests — Программный пакет сторонних разработчиков, с которым вы уже успели познакомиться. Применяется для отправки HTTP-запросов к серверу

Flask, Django — Мы уже упоминали эти фреймворки. Они понадобятся при написании серверного кода на Python. Flask лучше всего подходит для создания простых приложений, а Django — для разработки сложных проектов

sched — Требуется выполнять программы в конкретное время или по расписанию? Тогда вам нужен `sched` — стандартный модуль Python, позволяющий реализовать запуск программ по расписанию

logging — Встроенный модуль, позволяющий отображать разного рода предупреждения и сообщения об ошибках, вместо того чтобы выводить диагностику с помощью функции `print()`. Более того, можно настраивать вид сообщений в зависимости от режима выполнения программы (например, тестирование или запуск в производственной среде)

Pygame — Библиотека, предназначенная для написания видеоигр. Содержит модули работы с графикой и звуком. Подходящее решение для начинающих разработчиков игр

Beautiful Soup — Данные в Интернете далеко не всегда доступны в формате JSON. Часто приходится загружать веб-страницу целиком и анализировать HTML-разметку для поиска нужных сведений. К сожалению, многие веб-страницы содержат достаточно посредственную разметку. Библиотека Beautiful Soup содержит удобные инструменты для работы с веб-страницами на Python

Pillow — Графическая библиотека, содержащая все необходимое для чтения, записи и обработки изображений. Поддерживает множество популярных форматов, а также позволяет выводить изображения на экран и конвертировать их из одного формата в другой

Список основных библиотек Python доступен по следующему адресу:

`https://wiki.python.org/moin/UsefulModules`

Не могу поверить, что книга подошла к концу. Остался только предметный указатель. Все было так интересно! Надеюсь, мы еще увидимся...

Не печальтесь, мы не прощаемся

Это не конец истории. Теперь, когда вы приобрели базовые знания в области программирования, пора переходить на следующий уровень. Посетите наш сайт `https://www.wickedlysmart.com`, чтобы узнать, какие обучающие ресурсы имеются в вашем распоряжении.

Предметный указатель

A

API 472

B

Beautiful Soup 621

D

Django 617

F

Flask 617

H

HTTP 474

I

IDLE 32, 49, 54, 76
  перезапуск 339

J

JSON 481, 486

M

MVC 512

O

Open Notify 479
  запрос 485

P

Pillow 621
pip 477
Pygame 621
Python 52

S

Spotify 475

T

Timsort 275

U

URL 474

W

Web API 472

А

Абсолютный путь 437
Алгоритм 40
  эвристический 300
Анализ текста 286
Аргумент 97, 228
  командной строки 459
  позиционный 248
Ассоциативный массив 398
Атрибут 353, 566

Б

Баг 90
Библиотека 113, 330, 477
  Tk 544
Блок-схема 112
Булево значение 116
Бумажное прототипирование 509

В

Веб-запрос 473
  код состояния 484
Веб-служба 472
  адрес 474
Виджет 521
  меню 543
Вложенный цикл 267
Время 613
Вызов функции 73
Выражение 81
Вычислительное мышление 30, 50

Г

Генератор 618
Глобальная переменная 230, 238, 243
  __name__ 332
Графический интерфейс 503

Д

Дата 613
Декоратор 619
Делегирование 596
Диаграмма
  класса 352, 570
  состояний 533
Диапазон 181
Документация 134
Документирующая строка 336

З

Закрытие файла 438
Запрос 473
Значение по умолчанию 246

И

Импорт модуля 61, 113, 287, 288, 330, 335, 522
Индекс 163
  отрицательный 169
  удобочитаемости 283, 314, 318
Инкапсуляция 355, 588
Инструкция
  break 442
  from 522, 527
  if 118
  import 61, 113, 288, 330
  return 228
  try 453
  yield 618
Интерпретатор 49
Исключение 453
  KeyError 399
Исходный код 34, 42
Итератор 445

К

Кавычки 78
Кадр 387
Каркасный код 294
Класс 352, 563
  базовый 572
  наследование 570
  производный 572
Ключ 398
Ключевое слово 87, 246
  except 454
  global 243
Кнопка 522
Код 42
  выполнение 76
Командная строка 459
Комментарий 60, 135
Конкатенация 61, 82
Консоль 52
Константное время 403
Конструктор 354, 564
Контроллер 512, 527
Координаты 490
Корневой каталог 437
Кортеж 615

Л

Логический оператор 129
Локальная переменная 230, 238

М

Мемоизация 413
Менеджер компоновки 524
Меню 542
Метод 353, 566
  read() 439
  readline() 443
  strip() 449
  write() 457
  переопределение 578
Множественное наследование 577
Множество 616
Модель данных 513
Модуль 58
  datetime 613
  json 486
  logging 621
  random 61, 113
  re 614
  requests 477, 483, 621
  sched 621
  sys 459
  time 394
  Tkinter 521, 544
  turtle 342, 489
  импорт 61, 113, 287, 288, 330, 335, 522

Н

Наследование 570, 594
  множественное 577
Неоднородный список 170

О

Область видимости 236, 238
Обработчик события 530
Объект 39, 351, 567
  date 613
  datetime 613
  response 484
  screen 489
  self 564
  str 576
  StringVar 544
  time 613
  Tk 522, 544
    метод after() 535
  атрибут 566
  исключения 453
  первого класса 620

Объектно-ориентированное программирование (ООП) 560
Объявление функции 219
Окно 522, 526
Операнд 129
Оператор 81
  in 296
  булев 128
  логический 128
  приоритет 83
  точка 330, 354
Открытие файла 435
Отладка 90
Отложенные вычисления 618
Относительный путь 436
Отрицание 129
Ошибка 58
  выполнения 90

П

Пакет 330, 477
Палиндром 383
Параметр 222, 228, 238
  обязательный 247
  по умолчанию 246
Переменная 74, 80
  __name__ 332
  глобальная 230, 238, 243
  имя 87
  локальная 230, 238
  область видимости 236, 238
  экземпляра 353
Поведение 351
Подкласс 572
Подстрока 291, 308, 310
Полиморфизм 591
Порождающая система 506
Порядок выполнения 123
Последовательность 182
Потомок 573
Представление 512, 521
Приглашение командной строки 76
Примитивный тип 161
Приоритет операторов 83
Программа 42
Прототип 509
Псевдокод 40, 72, 267, 285
Пузырьковая сортировка 264
Пустая строка 124
Пустой список 170
Путь к файлу 436

Р

Разделитель 291
  пути 436
Реактивное программирование 530
Реализация 72
Регулярное выражение 614
Рекурсия 379
Рефакторинг 235
Рецепт 39

С

Семантика 571
Семантическая ошибка 90
Символ новой строки 443, 444
Синтаксис 73, 571
Синтаксическая ошибка 90
Скобки 84
Словарь 398, 482
  строковый 400
Случайное число 114
Снежинка Коха 418
Событие 453, 530
  запуск по таймеру 535
Сортировка
  алгоритм Timsort 275
  пузырьковая 264
Списковое включение 612

Список 161, 163, 615
вложенный 193
двухмерный 513
неоднородный 170
параллельный 198
пустой 170
размер 167, 170
создание 164, 192
суммирование 378
Среда разработки 32
Срез 308, 310
Стек вызовов 387, 391
Строка 74, 94
Структура данных 161
Суперкласс 572
Сценарий использования 509
Счетчик цикла 182

Т

Тестирование 509
Тип данных 94
примитивный 161

У

Удобочитаемость 283, 318
Управляющая последовательность 444
Уровень текста 283, 318
Условие 119
Условное выражение 118

Ф

Файл
закрытие 438
открытие 435
путь 436
режим доступа 435
сохранение 457
текстовый 432
чтение 435, 439, 443
Файловый объект 439
Фрактал 415
Фреймворк 617
Функция 219
append() 192
choice() 61, 122
del() 193
extend() 193
float() 95
get() 483
__init__() 565
input() 73
insert() 194
int() 95
isinstance() 574
len() 167, 304
load() 486
mainloop() 532
open() 435
print() 58, 97
randint() 113
range() 181
sort() 275
split() 289
str() 176
__str__() 577
sum() 378
аргумент 228, 248
вспомогательная 446
вызов 73
высшего порядка 620
значение None 250
имя 228
параметр 222, 228

Х

Хеш-таблица 403
Хеш-функция 403

Ц

Цикл 141
for 178
while 142

бесконечный 149
вложенный 267
досрочный выход 442

Ч

Число 94
случайное 114
Фибоначчи 393, 412
Чтение файла 435

Ш

Шаблон проектирования 445

Щ

Щелчок мыши 531

Э

Эвристический алгоритм 300
Экземпляр 353
Элемент списка 163

Я

Язык программирования 37, 43