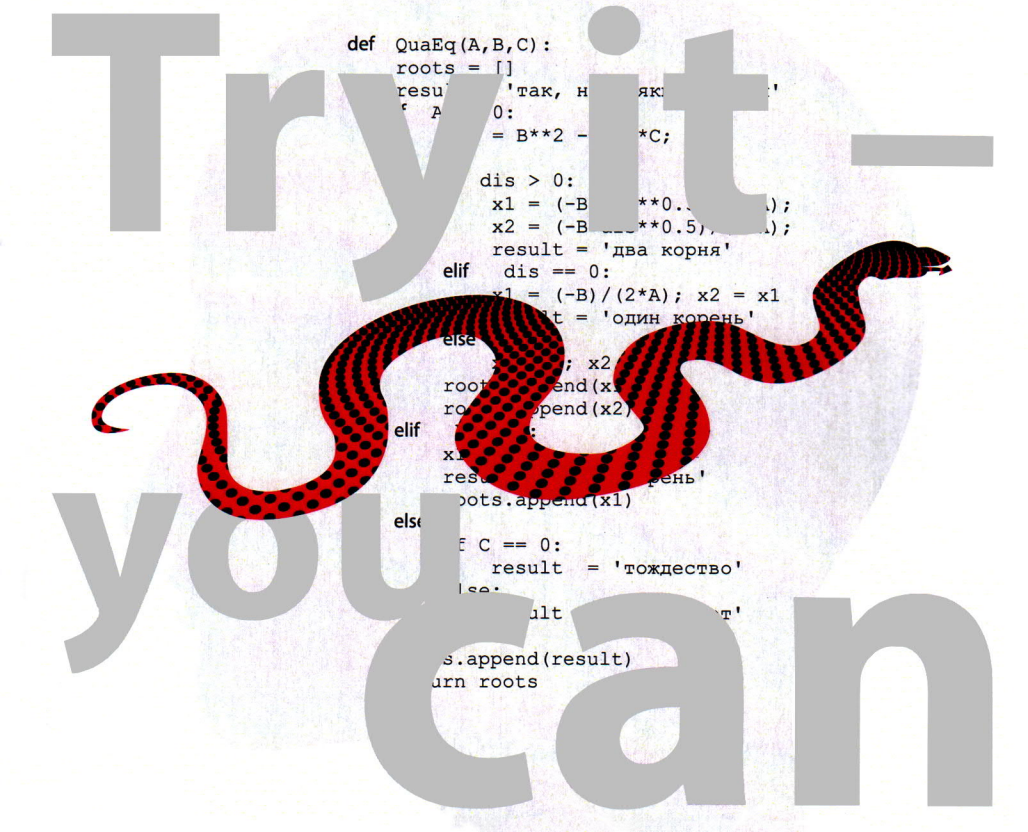
Серия «Программирование»

«СОЛОН»

Рик Гаско

Простой Python просто с нуля



Серия «Программирование»

Рик Гаско

Простой Python

просто с нуля

Москва
СОЛОН-Пресс
2019

УДК 681.3
ББК 32.973-18
К 63

Под редакцией Н. Комлева

Рик Гаско

Простой Python просто с нуля. — М.: СОЛОН-Пресс, 2019. — 256 с.: ил. (*Серия «Программирование»*)

ISBN 978-5-91359-334-4

Язык программирования Python. Он моден, он актуален, он в тренде. Python работает везде. Python используют все – от суперкорпораций до сдачи ЕГЭ. Python универсален. Для Python создано колоссальное количество расширений для решения буквально любой задачи. Python прост, очевиден и прозрачен.

Эта книга – лучший выбор для освоения языка – просто, доступно, живо.

Это не только учебник Python – это и начало долгого пути в мире программирования.

Присоединяйтесь!

По вопросам приобретения обращаться:
ООО «СОЛОН-Пресс»
Тел: (495) 617-39-64, (495) 617-39-65
E-mail: kniga@solon-press.ru, www.solon-press.ru

Ответственный за выпуск: **В. Митин**
Под редакцией: **Н. Комлева**
Обложка: **СОЛОН-Пресс**

ООО «СОЛОН-Пресс»
115487, г. Москва,
пр-т Андропова, дом 38, помещение № 8, комната № 2.
Формат 60×88/16. Объем 16 п. л. Тираж 100 экз.

ISBN 978-5-91359-334-4

Посвящается тем,
кто просто любит программировать,
и неважно, на каком языке.
И любит змей и лягушек, само собой

Содержание

Вступление

О чём эта книга?
Объяснение в лирической форме

Питон, он же Python – язык программирования, получивший в последнее время огромную популярность.

Я написал хорошую книгу по языку программирования Питон. Книга эта была рассчитана только на тех, кто уже программирует, и программирует неплохо. Так она и задумывалась, но мне это не понравилось.

Теперь лирическое отступление. Великий писатель земли русской Лев Толстой задумал сочинить роман о войне двенадцатого года. Задумал – и начал сочинять. Естественно, если роман о войне двенадцатого года, то с этого года он и начал – это когда всеми любимый поручик Ржевский скачет по степям с Наденькой или Сашенькой из кинофильма *Гусарская баллада*, а в это время про него сочиняют анекдоты. Потом Толстой задумался и решил начать с самого начала, то есть с совсем древних времён – сражения при Аустерлице, когда Наполеон немного разгромил русскую армию. Так будет честнее, решил солнце русской прозы и зеркало русской революции, начинать надо с поражений, а потом уже рассказывать, как мы им вдули. Я себя с Толстым не равняю, конечно, куда мне до него. Впрочем, и он программирования не знал, так что 1:1. Далее старый, но смешной анекдот:

Попал мужик под паровоз. Вылечили его. Лег он дома спать, а жена на кухне чайник поставила. Чайник вскипел — и давай свистеть. Мужик вскакивает и топтать чайник, топтать!
Жена офигела:
— Ты чего, Федя?
— Убивать их надо, пока маленькие!!! © Народное

Я решил, что не надо писать продвинутую книгу по языку программирования, пока я не написал книгу для начинающих по тому же языку. Вот когда эту книгу издатели издадут, читатели прочитают и программисты научатся программировать по ней, только тогда и наступит время для книги для очень продвинутых программистов.

Ещё я написал хорошую книгу по языку программирования Паскаль. Это был учебник не только языка, но и программирования, с самого начала, для тех, кто не знает вообще ничего, ни о Паскале, ни о Программировании. Разумеется, у меня возникла мысль предложить Дорогому Издателю™ эту же замечательную книгу, только заменив программный код на Паскале на программный код на Питоне. Но нет, оказалось, что я на такую халтуру не способен, и книга сама собой стала писáться с самого начала, заново, и совсем другая.

Я не буду объяснять, чем хорош Питон, и доказывать, что программировать надо именно на нём, по очень простой причине. Если вы выбрали именно эту книгу и заплатили за неё деньги – вы ведь заплатили, правда, не расстраивайте меня – значит вы уже уверены, что программировать стоит именно на Питоне.

И ещё. Возможно, сначала это прозвучит не совсем понятно, но у создателя языка были свои планы в отношении того, где и как Питон будет применяться. Мы не обязаны следовать за его указаниями и соответствовать его ожиданиям. Мы будем использовать Питон так, как хочется нам. Конкретно мне хочется использовать Питон как совершенно обычный язык программирования. Опять непонятно? Ничего страшного. Постепенно всё прояснится.

Кое-что о Питоне

Бесполезная информация. Кто такой питон, знаете? В смысле, одноимённую змею помните? Забудьте. Имя языку Питон дано не по названию змея-аспида, а в честь английской комик-группы Monty Python начала семидесятых. Юмор чисто английский, для нас непонятный, а сейчас, наверное, непонятный и для самих англичан. Впрочем, на иконках всё равно рисуют какую-то гадюку. По-русски имя группы традиционно произносится *Монти Пайтон* – в точности, как и по-английски. А вот змею у нас зовут *Питон*. Так что выбирайте сами, что вам дороже – правда или истина.

Однажды я решил освоить какой-нибудь нетрадиционный язык программирования. *Нетрадиционный* – в хорошем смысле слова, а не тот, которым пользуются исключительно лица нетрадиционной ориентации. Я перепробовал пять или шесть языков. Я не буду называть их имена, это

очень уважаемые имена. Процесс шёл вполне предсказуемо. Я скачивал собственно программную часть – компилятор, оболочку, интегрированную среду, библиотеки и что там ещё бывает. В комплекте шли какие-то инструкции по применению и встроенная справка и контекстная помощь или как оно теперь называется. Потом, само собой, я искал учебники и руководства, для начинающих и для продвинутых.

Закончилась посадка в самолёт. В динамике голос: "Здравствуйте, дорогие пассажиры, вас приветствует командир корабля! На борту во время полета вы можете посетить два бара, ресторан, бассейн, бильярд и сауну. А сейчас со всей этой фигней мы попытаемся взлететь" © Шутка

А потом я пытался со всей этой фигнёй взлететь, то есть хоть что-то запрограммировать, и чтобы, я вас умоляю, оно заработало. Нет, оно не работало. Обычно оно даже и не транслировалось. То, что учебники не соответствовали компилятору, не очень и удивляло. Встроенная справка не соответствовала ничему вообще. Видимо, компилятор уже поменяли – а справку-то зачем? И это не было отдельными, изолированными случаями – это было системой.

исторический момент
Летом 1942 года по просьбе американцев Советский Союз передал им для изучения танк Т-34. Его испытывали на Абердинском полигоне. Танк этот на тот момент был, безусловно, лучшим танком в мире. Однако были нюансы. Производили танк тогда только на двух заводах, один из которых – Харьковский, к тому моменту давно уже был захвачен немцами, теперь – украинцами. Второй завод, Сталинградский, ещё работал, но под немецкими бомбами. Производство танков стремительно перемещалось в Сибирь. Качество было вполне ожидаемым и предсказуемым.
Хотя у американцев на тот момент своих танков почти и не было, советский танк был подвержен самому критическому разбору. Что-то похвалили, что-то поругали. Но вот воздушный фильтр, заявили американцы, наверняка проектировал вредитель!
А вы что думаете, зря этих вредителей в 37-м году пачками расстреливали?
конец Исторического момента

Так вот, все разработчики всех этих нетрадиционных языков программирования, все – от первого до последнего – все вредители. Все эти языки спроектированы и реализованы в точности, как воздушный

фильтр на Т-34. Хотя, меня внезапно посетила внезапная мысль, может они и не вредители? Кстати, поищите в Яндексе по словам *анекдот может она телеграмму не получила?* Может, они не вредители, а им просто денег не платили? Может, они просто работали бесплатно? Это я не про Т-34, это про бесплатные языки программирования. Обдумайте.

А ещё более потом я решил опробовать Питон. И всё заработало. И сразу. Встроенная справка соответствовала интегрированной среде. Компилятор компилировал. Программы транслировались и работали. Иногда правильно, иногда неправильно, но это уже проблема программиста. Что удивительно, учебники, стареы и нестарые, содержали вполне адекватные тексты программ – эти программы работали тоже. On-line справка, для извращенцев, тоже была и тоже вполне соответствовала.

Я удивился и решил остановиться на Питоне. И что я о нём узнал, здесь я вам и расскажу.

Формальная информация, очень короткая

Питон – язык программирования. Питон неуклонно набирает популярность и достиг того, что его используют в ЕГЭ, Выдумал Питон Гвидо Ван Россум, он голландец, это многое объясняет, в Голландии легализованы лёгкие наркотики. Первая версия появилась относительно давно, в 1994 году. Питон работает под очень многими операционными системами, или, говоря иначе, платформами. Нас будет интересовать, само собой, в основном Windows.

Питон безостановочно развивается – выходят всё новые и новые версии. Мы за новизной не гонимся, поэтому использовать будем версию 2.7.14. Это самая стабильная подверсия из второй версии и она всем настоятельно рекомендуется. Возможно, когда вы будете читать эту книгу, её место займёт какая-нибудь из третьих версий.

Ещё у Питона есть, вы не поверите, философия. У Дельфи философии нет. У фортрана тоже нет, я проверял. А у Питона есть, и она даже в нём прошита – в смысле вы набираете в интегрированной среде (о среде позже) фразу `import this` и получаете в ответ всю философию. Если вам это интересно, философию я поместил в приложения.

Что, кроме этой книги, почитать?

Само собой разумеется, даже если эта, моя, книга – лучшая в мире книга по Питону, надо обязательно прочесть что-то ещё. Любая, даже самая лучшая книга, даёт взгляд – или проекцию – только с одной стороны. Неплохо бы посмотреть и с другой и составить объёмную картину. Так что дальше несколько книг для рекомендованного прочтения. Точнее, несколько авторов, потому что названия у всех книг сходные до полной неразличимости. Ещё одно обшее у авторов – они повторяются. Не в смысле пишут одно и то же в разных книгах. В смысле поворяют одну и ту же мысль на протяжении книги несколько раз немного разными словами, чтобы лучше дошло до читателя. Можно, я тоже так буду?

Обязательно и без всякого сомнения следует прочитать Гвидо нашего Россума. В конце концов, это его язык, он его придумал, он его хозяин и мнение своё высказать имеет полное право. Хотя, по сути, его книги это только введение и базовые концепции языка.

Лутц (Mark Lutz) – написал очень много книг, буквально десятки, некоторые книги просто введение в язык, некоторые на очень глубокие специфические темы, которые далеко не всякому программисту и понадобятся. Объекты, в смысле создания своих объектов, изложены очень неуверенно и путано. Это я так нагло его критикую потому, что считаю – в Объектно Ориентированном Программировании (ООП) я разбираюсь лучше. Заодно займусь рекламой – прочитайте мою книгу по ООП. В идеале, заодно и купите.

Так вот, я очень советую прочесть его книгу *Изучаем Питон (Learning Python)*. Это самое лучшее введение в язык, хотя местами я бы подсократил. Кроме того, Лутц всё время сравнивает Питон с другими языками, причём всегда в пользу Питона. Как-то это неспортивно. Впрочем, лично мне это не обидно, потому что обычно сравнивает он с C++, не с Дельфи. А C++ я не люблю. Кстати, кто бы объяснил, почему на обложке его книги не змея, а крыса?

Прочтите Пилгрима *Вглубь языка Питон*. Технология ООП описана у него хорошо. Всё остальное, к сожалению, ограничивается введением в язык.

Ещё один автор – Сузи Роман Авревич. ООП изложено хорошо. Всё остальное тоже хорошо, но только ни разу не для начинающих. По стилю напоминает справочник. Справочники я уважаю, но начинать знакомство с предметом лучше с чего-то написанного доступнее.

В общем, структура типовой книги по Питону выглядит так:
Основные понятия
Техника программирования
Ещё что-то, обычно Объектно Ориентированное программирование
И множество узкоспециальных тем.

Я хочу сосредоточиться на первых двух пунктах.

И ещё – очень советую прочитать несколько старых книг просто по программированию, не на Питоне, а просто. Классика, можно сказать.

Б.Керниган, Ф.Плоджер «Элементы стиля программирования», «Радио и связь», 1984, «The Elements of Programming Style»

До Питона было ещё очень далеко, но прочитать обязательно – чтобы понять где хорошая программа, а где плохая.

Лу Гринзоу «Философия программирования Windows 95/NT», Санкт-Петербург, «Символ»,1997, Lou Grinzo Zen of Windows 95 Programming.

Питон уже где-то рядом. Если хотите писать реальные программы под реальный Windows, то это читать обязательно.

Роджерс «Алгоритмические основы машинной графики», Мир, 1989, «Procedural Elements for Computer Graphics».

Эта книга, безусловно, о программировании. Но во многом, скорее, математическая. Поэтому она не устаревает. О том, как оно всё на самом деле рисуется на экране.

Э.Дейкстра «Дисциплина программирования».

Величайший программист всех времён и народов, кроме шуток. Объектно Ориентированное Программирование он не очень любил. В смысле, совсем его не любил.

Нужно ли что-то уже уметь?

Нет, ничего не надо! Эта книга именно для тех, кто программировать не умеет, от слова *вообще*. Как я уже упомянул – сам не похвалишь, никто не похвалит – я уже написал очень хороший учебник по Паскалю. Так вот, ваше знакомство с этой моей замечательной книгой *не* предполагается, точно так же и не предполагается знакомство с программированием вообще.

Разумеется, я буду ожидать умения общаться с файлами, в общечеловеческом смысле слова – то есть уметь открыть, сохранить, сохранить как… Мне кажется, это уже давно умеют все. А вот как создать файл изнутри программы, я вас научу.

Математику желательно знать вплоть до решения квадратного уравнения. Ну, или хотя бы до знания, что такое квадратный корень И что такое возведение в степень. Из геометрии неплохо знать, чем прямоугольный треугольник отличается от равностороннего. Из тригонометрии – что есть такое *синус*. Кто он такой, знать не обязательно, просто знать, что он есть.

Если вас интересует Питон сам по себе, то на этом всё. Если вы думаете о связи вашей программы с другими программами – не важно, через файлы, по сети или непосредственным обменом данных через память – то надо чётко понимать, как переменные устроены внутри. Если оставаться внутри Питона, то это знание лишнее, а если вы хотите в дальнейшем выйти за его пределы, то знание является обязательным, Лишнее я вас учить не заставляю, прочитайте два приложения к этой книге – о байтах и о переменных.

А вот ещё хорошая цитата:

Если что-то из сказанного выше вам показалось непонятным, не волнуйтесь © Лутц

Вот в таком духе и будем продолжать.

Осознанный выбор, или Информированное согласие

Если вдруг, чего, конечно, вам не желаю, вы попадёте в больницу и Добрые Врачи™ захотят у вас отрезать что-то лишнее, то сначала вам подсунут на подпись документ под названием *Информированное Согласие*. Сейчас его подсократили, а раньше он был куда смешнее. Помимо прочего, вы разрешали студентам осматривать вас в живом и мёртвом виде, использовать ваши фотографии в любом ракурсе, а также изготавливать из вашей тушки любые препараты, только для обучения, само собой.

Весь смысл в том, чтобы вы чётко понимали, что вас ждёт в будущем. Тут у нас в программировании препараты из вас делать не будут, но тем не менее. Если вы хотите программировать на Питоне, то должны понимать, действительно ли вы хотите программировать на Питоне. Очень неплохо будет, если сначала вы бросите взгляд на Питон и другие языки и выберете тот, который вам больше нравится. Этим мы сейчас и займёмся.

Как вы, конечно, знаете, у нас в стране группой альтернативно одарённых лиц по предварительному сговору внедрено ЕГЭ – единый государственный экзамен по разным предметам, в том числе и по информатике. Информатика это вовсе не программирование, но и немного программирования в заданиях присутствует. Использовать при этом сейчас можно целых пять языков. Вот они: Бейсик, Школьный Алгоритмический Язык, Python, C++, Pascal.

Возьмём не очень сложную задачу и посмотрим, как она решается на этих пяти языках. Не очень сложная задача – это *пузырьковая сортировка*. Что такое сортировка вообще? Есть числа, например 10,22,11,50,30. На выходе имеем те же числа, но уже в другом порядке: 10,11,22,30,50. Для невнимательных поясняю – числа теперь идут по возрастанию. Если чисел только пять, то их можно отсортировать, только взглянув на них один раз и записав в правильном порядке. Если чисел сотни и тысячи, применяют какой-нибудь алгоритм, часто – пузырьковую сортировку. Считается, что применяют её только плохие программисты, потому что она медленная. Всё верно, но при современных скоростях процессоров это имеет

значение, только если надо отсортировать десятки тысяч чисел, да и то не один раз. Зато она простая и понятная.

В чём идея? Идём по числам и сравниваем. Если первое больше второго, меняем их местами. Если второе больше третьего, меняем местами их. Обратите внимание – если на первом шаге мы уже поменяли числа местами, то нынешнее второе – это бывшее первое. И так до конца списка. Если за весь проход никого местами не поменяли, значит наступило счастье и все числа отсортированы. Если кого-то поменяли, повторяем проход по списку чисел ещё раз. Обдумайте. Почему сортировка называется пузырьковой? Создателю алгоритма виделся образ пузырька, всплывающего в бокале с шампанским.

В примерах принципиально отсутствуют ввод и вывод данных. Входные данные задаются непосредственно руками. Выходные не выводятся вообще никак.

1. Бейсик. Придуман в 1964-м. Авторы Томас Курц и Джон Кемени, американцы. Использована встроенная в Microsoft Office версия Visual Basic. Выносится искренняя признательность геофизику Григорию, сваявшему это код, скучая под тенью кедра в недрах тундры на буровой в окрестностях Нижневартовска.

```
Dim i, j, ind1, ind2, t As Integer
   Const n As Integer = 5
   Dim a(n - 1) As Integer

   Rem заполнение
   a(1) = 10
   a(2) = 22
   a(3) = 11
   a(4) = 50
   a(5) = 30

   Rem сортировка
   For i = 0 To n - 1
      For j = 0 To n - 1 - 1
         If a(j + 1) < a(j) Then
            t = a(j)
            a(j) = a(j + 1)
            a(j + 1) = t
         End If
      Next j
   Next i
```

2. Школьный алгоритмический язык. Придуман в 1980-е. Автор Андрей Ершов, русский. Использована реализация в среде Кумир.

```
алг Сортировка
нач
. цел таб a[1:5]
. лог ok
. цел tmp
. цел i
. | заполнение таблицы
. a[1]:=10
. a[2]:=22
. a[3]:=11
. a[4]:=50
. a[5]:=30

.
. | сортировка
. ok:=нет
. нц пока не ok
. . ok:=да
. . нц для i от 1 до 5-1
. . . если a[i] > a[i+1]
. . . . то
. . . . . tmp:=a[i]
. . . . . a[i]:=a[i+1]
. . . . . a[i+1]:=tmp
. . . . . ok:=нет
. . .   все
. . кц
. кц
.
кон
```

3. Python. Придуман в 1991-м. Автор Гвидо Ван Россум, голландец. Использована стандартная версия 2.7.14.

```
#  заполнить список значениями
a = [ 10,22,11,50,30 ]

# отсортировать
ok = False

while not ok:
    ok = True
    for i in xrange(0,len(a)-1):
        if a[i] > a[i+1]:
            a[i],a[i+1] = a[i+1],a[i]
            ok = False
```

4. C++. Придуман – вопрос философский – 1972–1983. Авторы – вопрос философский. Или Денис Ритчи и Кен Томпсон, американцы. Или Бьёрн Страуструп, датчанин. Использован Turbo C++ 3.0.

```
int a[5];
int ok,tmp;


// заполняем массив
a[1] = 10;  a[2] = 22; a[3] = 11;  a[4] = 50; a[5] = 30;

// сортируем
ok = 0;
while (!ok)
{
   for (int i = 1; i < 5; i++)
   {
   ok = 1;
   if (a[i] > a[i+1])
      {
      tmp = a[i];
      a[i] = a[i+1];
      a[i+1] = tmp;
      ok = 0;
      }
   }
}
```

5. Pascal. Придуман в 1970-м. Автор Никлаус Вирт, швейцарец. Использовано Delphi 7.

```
const
   N = 5;
var
   a          : array[1..N] of integer;
   ok         : boolean;
   tmp        : integer;
   i          : integer;

{  тут мы заполняем массив значениями  }
a[1]:=10; a[2]:=22; a[3]:=11; a[4]:=50; a[5]:=30;

{  а тут мы его сортируем  }
ok:=false;
repeat
   ok:=true;
   for i:=1 to N-1 do begin
      if a[i] > a[i+1] then begin
         ok:=false;
         { поменять местами  }
         tmp:=a[i];
```

```
            a[i]:=a[i+1];
            a[i+1]:=tmp;
        end;
    end;
until ok;
```

Выводы с высоты птичьего полёта. То есть мы не вникаем в смысл программного кода, а только смотрим на внешний вид. Как мы видим, текст на Паскале самый длинный, а текст на Питоне самый короткий. Само по себе это, конечно, ни о чём не говорит. Текст на Паскале длинный потому, что я Паскаль знаю очень хорошо и потому программирую не очень аккуратно и, следовательно, длинно. Заметьте, что в использованной версии C++ тогда ещё не было булевских переменных. Смотрите, сравнивайте и выбирайте, что хотите. Но учтите, раз эта книга по Питону, выбрать вам придётся именно его.

Несколько почему

Почему я везде пишу *вы* с маленькой буквы? Мне так кажется гораздо более правильным. Когда я пишу письмо конкретному человеку, я пишу Вы с большой буквы. Когда я обращаюсь к неопределённому кругу заранее неизвестных мне лиц, то пишу с маленькой.

Почему в книге простых человеческих слов больше, чем программного кода? Отвечаю и извините, если кому-то напомню о грустном. Когда я был молодой и … Нет, не в смысле трава была зелёная и девки давали. Когда я был молодой и лечил зубы, каждый знал, что лечение зубов заключается в том, что будут сверлить, сверлить и сверлить. А в конце быстро набросают цемента. Сейчас сверлят совсем мало, но долго делают непонятно что. Результат сейчас в целом лучше. С обучением программированию в целом то же самое.

Книга, на девяносто процентов состоящая из кода, никого не научит программировать. В лучшем случае она сможет служить справочником для уже умеющих программировать.

Почему я по несколько раз обращаюсь к решению одной и той же задачи? Потому что я решаю её другим способом. Решать одну и ту же задачу разными способами мне кажется очень и очень полезным. То, что называется *предметная область*, то есть – о чём программа, вам уже знакомо. Теперь можно сосредоточиться на методах её решения. Понятно,

что в реальном программировании, в том, которое за деньги, всё не так. Если вы в совершенстве выучили один способ, зачем учить другой? Но если вы ещё не выучили ни одного, лучше попробовать изучить все.

И ещё одно почему.

почему
Почему некоторый текст написан вот именно так, как написан вот этот. То есть в этих вот решёточках? В Питоне решётками выделяются комментарии, то есть такой текст, который на выполнение программы никак не влияет, а пишется только для самого программиста. При исполнении программы он игнорируется.
Вот и тут то же самое. Вы смело можете пропустить всё, что внутри решёток, на понимание дальнейшего пропуск никак не повлияет. А для чего это написано? Для расширения вашего кругозора и открытия перед вами сияющих перспектив.
конец Почему

О главном – о том, чего в этой книге нет

Никакой игры слов, здесь речь действительно пойдёт о том, чего в моей книге нет, но что я считаю действительно важным и заслуживающим внимания.

Здесь нет ни слова об Объектно Ориентированном программировании. Это может показаться странным, потому что, с одной стороны, в Питоне буквально всё – объекты, а, с другой стороны, я в этой области являюсь, скромно говоря, специалистом. Противоречие кажущееся. В Питоне сплошные объекты и применение их совершенно естественно и незаметно для программиста. Но разработка своих собственных объектов – не сказать, чтобы очень просто и очевидно. Оставим тему для другой книги. А тема действительно очень важная и для реального профессионального программирования необходимая.

Кстати, Лунтц в своих книгах упорно демонстрирует ООП на примере пиццерии и только пиццерии. Натянуть сову на глобус занятие не из простых, но здесь не об этом. Один из объектов – сотрудник. У него есть методы *повысить зарплату*. Методов *понизить зарплату* или *уволить* нет. Это теперь называется *позитивное мышление*.

Другая тема – итераторы, генераторы и прочие чрезвычайно полезные возможности, резко сокращающие длину программы. Но тут такое дело – сокращать не значит упрощать.

Очень маленькая тема, но это гордость Питона. Что называется, фирменный лейбл. Называется это *анонимные функции*, они же lambda-функции. Долго думал, какая от них может быть польза, кроме как показать уровень своего интеллекта, но не понял. Отложим до следующей книги.

И ещё такое понятие, как *исключения*. В том или другом виде и под разными названиями они существуют в большинстве языков программирования с незапамятных времён. Это очень полезная концепция и достаточно понятная. Но только я их не люблю и никогда ими не пользуюсь. Ну не нравятся они мне, не нравятся.

А ещё в книге нет ничего о программировании GUI – Graphic User Interface, ориентированном на события. По-простому – нажал на кнопку и что-то случилось. А ещё ничего не будет о базах данных, сокетах, интернетах и многом, многом другом…

Глава первая. Начало

Где скачать и как установить

Сначала – что скачать. Как вы прекрасно знаете, у каждой программы есть версия, чем выше версия, тем новее программа. Версии обычно обозначаются цифрами и часто через точку. Например, 1.11 или 2.22. Первая цифра – это основная версия (main version), а то, что после точки – minor version, переведите сами. Число одиннадцать после точки, как правило, обозначает, что после выхода основной версии появилось ещё одиннадцать версий с мелкими улучшениями.

В случае с Питоном, есть основная третья версия, но самой распространённой является версия 2.7.14. Как видите, появилась ещё одна точка.

Теперь забудьте об этом всём, просто войдите в Яндекс и наберите *где скачать Питон 2.7.14*. Скорее всего, первая же ссылка приведёт вас на *python.org*. Это самый авторитетный, официальный и уважаемый источник. Скачайте то, что надо, вы справитесь.

Установите. Вы ведь умеете устанавливать программы. В Главном Меню появится пункт <Python 2.7>, а в нём несколько подпунктов. Лично я туда не заглядываю, потому что у меня на рабочем столе появились ещё и иконки. Если у вас их нет, создайте или запускайте программы через Главное Меню, смотря что вам больше нравится.

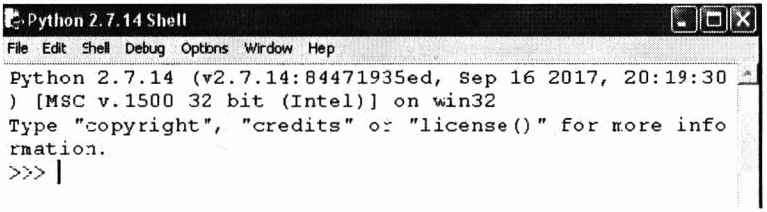
Иконок, как минимум, будет две, возможна и третья – это справка по языку. Под левой иконкой написано Python (command line). Это так называемая *консоль*. Все книги по Питону обязательно начинают рассказ с освоения консоли. Мне это кажется почти бесполезным, и я отправил консоль в приложения. Там вы её и найдете. Начнём сразу с настоящего программирования.

Программа, которая ничего не делает

Сейчас на рабочем столе у нас должна быть вот такая иконка с вот такой подписью.

Вот по ней щёлкаем и запускаем, говоря по научному, интегрированную среду программирования Питон. Называется эта интегрированная среда IDLE. Тут игра слов. С одной стороны, IDLE расшифровывается как Python's Integrated Development and Learning Environment. С другой стороны, *idle* значит *ленивый, бесполезный, не работающий*. Выбирайте, что вам больше нравится.

Выглядит среда, скажем прямо, не очень. Уступает не то что Дельфи или аналогам, но даже и древнейшему Турбо Паскалю.



Для начала, максимизируйте окно. К сожалению, я не знаю, как заставить его запускать максимизированным сразу. С чего мы начнём? Обычно программируют что-то вроде программы, выводящей *Hello, word!,* но лично мне это кажется слишком сложным и нелогичным. Начинать надо с программы, которая не делает *ничего*. Или даже так – *НИЧЕГО!*

Как выглядят программы, ничего не делающие, в других, как это называется, традиционных языках? Вот так выглядит абсолютно бесполезная программа на Паскале:

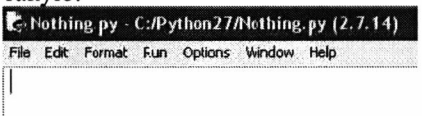
```
program Nothing;
begin
end.
```

А вот такая же, ну очень полезная программа на C++

```
void main ()
{
}
```

Питон в этой части достиг полной гармонии с природой. Программа, делающая *ничего*, сама по себе является *ничем*. Но сначала даже в Питоне мы должны проделать несколько несложных манипуляций. Выбираем в главном меню пункт File\New File. Получаем новое окно, тоже не очень впечатляющее:

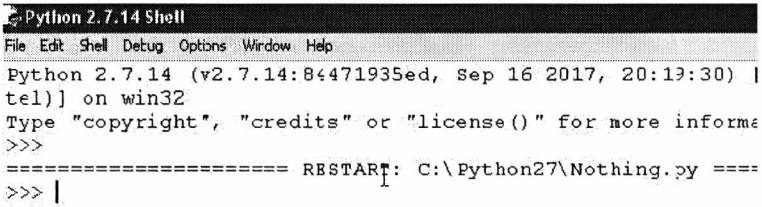
Теперь информация для любознательных. То, первое окно, называлось Shell Window, будем звать его *Оболочка*. Второе – Editor Window, его будем звать *Редактор*. Выбираем в меню File\Save as, получаем запрос о имени файла, пишем, разумеется, Nothing – картинка меняется на вот такую:



На что обратить внимание? На то, что вверху окна появилось имя файла в совокупности с именем каталога. Каталог именно тот, в котором установлен Питон. Пусть пока так и будет. Расширение у файла *.py. Это надо запомнить. Теперь возвращаемся к вопросу о том, как выглядит в Питоне пустая программа. А никак. Пустая, она и есть пустая. Другими словами – пустая, ничего не делающая программа, текста не имеет. Или можно сказать и так – если у нас есть совершенно пустой файл, то в нём содержится она, совершенно пустая программа, уже готовая к употреблению. Считайте, что дальше эта секретная программа напечатана – белыми буквами на белом фоне:

Арестант секретный, фигуры не имеет © Тынянов «Поручик Киже»

Тем не менее, даже не имеющая текста программа нуждается в том, чтобы её кто-то выполнил. Чтобы выполнить – нажимаем <F5>. Делаем это, напоминаю на всякий случай, находясь в Редакторе. Если мы внесли какие-то изменения в код и не сохранили, нам об этом напомнят и предложат сохранить. Соглашаемся и сохраняем. После чего мы автоматически, без нашего желания, переходим в Оболочку и видим вот такой экран:



Это и есть результат выполнения программы, которая ничего не делает. Ещё раз, на всякий случай – программа не делает абсолютно ничего, то, что написано сверху, – имя файла, в котором мы сохранили программу.

Если мы сейчас выйдем из IDLE и немедленно войдём снова, то, разумеется, увидим чистое окно Оболочки. Мы можем либо выбрать пункт File\Open и выбрать любой питоновский файл, либо выбрать File\Recent files и выбрать файл из недавно открывавшихся или редактировавшихся – в нашем случае, понятно, будет открыт тот самый единственный файл, который мы только что сочинили.

Программа, которая опять ничего не делает. Но творчески

Встреча с ветераном:
– Дедушка, расскажите о Ленине.
– Захожу я в общественный туалет, а там Ленин. Он делал то же, что и мы, но как просто, как человечно! © простой человечный анекдот

Программа состоит из операторов, или команд, или назовите это как угодно. В любом языке программирования есть оператор, который не делает ничего. *Я не знаю зачем и кому это нужно* ©. Питон ничем не лучше и не хуже, есть такой оператор, и в нём наша программа с его использованием выглядит вот так:

```
Pass
```

Результат выполнения будет в точности тот же самый, что и раньше, то есть – никакой. То, что мы написали, бессмысленно и бесполезно. Говоря по-другому – не несёт никакой информации. Но можно написать код, который бесполезен, но информацию несёт.

Поговорим о комментариях. Комментарии – это, как легко догадаться, когда что-то комментируют. То есть написали вы программный код и боитесь, что через неделю и не вспомните, о чём там речь шла. И вы оставляете пометку на память. С точки зрения языка, компьютера, эффект от комментариев тот же, как если бы их не было. То есть перед выполнением программы все комментарии просто выбрасываются. Другими словами – комментарии не для компьютера, комментарии для вас. Примеры.

```
# какая-то неведомая фигня
for i in xrange(0,11):  # а это цикл
    doSomething
    doSomethingElse
```

Каждая строка комментариев имеет впереди решётку. Всё, что от решётки и до конца строки, никому не интересно. Точнее не интересно компьютеру, программист комментарии может и должен прочитать. Комментарии я буду выделять курсивом.

Довольно часто возникает другая потребность – временно отключить часть программного кода, чтобы он не выполнялся. Зачем? Обычно потому, что в этой части программы есть определённые сомнения и хочется посмотреть, что будет без неё. Как это сделать? Если по правилам, то так:

```
# какая-то неведомая фигня
#for i in xrange(0,11):
#    doSomething
#    doSomethingElse
```

Всё верно, но скучно и утомительно. Есть другой способ, он неправильный, но все так делают:

```
'''
# какая-то неведомая фигня
#for i in xrange(0,11):
#    doSomething
#    doSomethingElse
'''
```

Всё, что между тройными кавычками – как бы комментарий. На самом деле, это не совсем так, но это работает и все так делают. Значит – можно и вам.

Выглядит она вот так:

```
print 'Is there anybody out there?'
```

Как результат, будет выведена в окне Оболочки вот эта самая строка, та, которая в кавычках. Если быть точным, будет выведена та строка, что мы набрали в кавычках, но выведена она будет без кавычек. Можно усложнить программный код, совсем немного:

```
print 'Is there anybody out there?'
print
print
print 'Is there anybody out there?'
```

Результат тоже будет чуть интереснее:

```
Is there anybody out there?

Is there anybody out there?
```

Как несложно догадаться, оператор `print` без параметров выводит пустую строку. И ещё один нехитрый фокус:

Оператор `print '-~=' * 10`
Результат `-~=-~=-~=-~=-~=-~=-~=-~=-~=-~=`

Иногда бывает надо отделить одну часть вывода от другой, чтобы легче было читать. *Звёздочка десять* в данном случае просто выводит строку десять раз подряд.

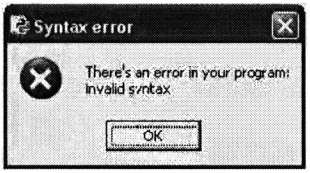
И теперь очень и очень важная тема, которую пропустить нельзя. Строка сама по себе заключена в кавычки. А что делать, если в составе самой строки уже есть кавычка? Пусть, например, мы хотим вывести такую строку: We don't need no education

старческое разъяснение
Обе английские строки, эта и выше, из Альбова *The Wall* группы *Pink Floyd* От 1979 года. Простые люди, вроде меня, считают этот альбом лучшим альбомом всех времён и народов. Эстеты-извращенцы не согласны и предпочитают *Dark Side of the Moon* тех же авторов.

конец Разъяснения

Что произойдёт при попытке просто и без затей вывести такую строку:

```
print 'We don't need no education'
```



А почему? А потому, что Питон воспринимает, как строку, только `'We don'`, после чего следует непонятный хвост и какая-то совсем ни к чему ещё одна кавычка. Я обещал не сравнивать Питон с другими языками, но иногда можно отступить от своих правил. В *традиционных* языках вместо одной кавычки в строке пишут две, вот так: `'We don''t need no education'`. Выводится в таком случае только одна кавычка. Питон против такого кода возражать не будет, но отреагирует по-другому:

```
print 'We don''t need no education'
We dont need no education
```

Если немного задуматься, всё становиться очевидным – Питон видит здесь две строки, записанные рядом, `'We don'` и `'t need no education'`. А если рядом, да ещё и без пробела, записаны две строки, почему бы не склеить их в одну? Вот и склеил. Так как же вывести эту несчастную кавычку? Питон здесь, как и во многих других случаях, пошёл своим уникальным путём. В Питоне строки можно записывать или в одинарных, по-русски одиночных, кавычках, или в двойных, по-русски точно так же. Извините, если вы это и так отлично знаете, но двойные кавычки – это не две одиночные кавычки подряд. Это совершенно отдельный символ, причём на той же клавише, что и одиночная кавычка. Никакой смысловой нагрузки это различие не несёт, никакой разницы нет – вообще.

```
print 'За что Герасим утопил Муму?'
print "За что Герасим утопил Муму?"

За что Герасим утопил Муму?
За что Герасим утопил Муму?
```

Однако! Внутри одинарных скобок совершенно свободно могут употребляться двойные, а внутри двойных – одинарные. То есть:

```
print 'We don"t need no education'
print "We don't need no education"

We don"t need no education
We don't need no education
```

Можно использовать для этой цели и тройные кавычки, в качестве заключающих в себя строку.

```
print '''a'b'''
a'b
```

С кавычками покончено. Ещё одно замечание и ещё один фокус. Выведем не одну строку, а две, и двумя разными способами:

```
print 'abcd','efgh'
print 'abcd'
print 'efgh'

abcd efgh
abcd
efgh
```

Что имеем? Если мы выводим две строки одним оператором, то они разделяются пробелом. Если мы выводим две строки двумя операторами, то они оказываются на разных строках вывода. А если мы хотим двумя операторами, но на одной строке? Пожалуйста:

```
print 'abcd',
print 'efgh'

abcd efgh
```

Да, весь секрет в этой самой запятой в конце первого оператора. Да, лично мне это не нравится.

Сеня, это же не эстетично! © Бриллиантовая рука.

И, наконец, две строки одним оператором, но чтобы вывод был сплошь, без разрывов:

```
print 'abcd' + 'efgh'
abcdefgh
```

Обсудив вывод, неплохо обсудить и ввод. Но вводить надо *что-то*. Это что-то называется *переменная*. Это, кроме шуток, самое важное понятие программирования. Поэтому сначала займёмся переменными, самими по себе, пока без их ввода.

Короткое, но важное добавление

В любой уважающей себя среде программирования есть встроенная справка. Есть она и в Питоне. Сначала надо нажать пункт меню <Help>. Далее возможен выбор.

Если выбрать <Hrlp\About IDLE>, то вы узнаете, в какой версии языка вы работаете. Это очень полезно – а вдруг у вас какая-то другая версия и то, что я пишу в книге о своей версии, не вполне относится к вашей.

Дальше. <Help\IDLE Help>. Это справка по интегрированной среде IDLE. Она очень короткая.

Третий пункт основной. Он называется <Help\Python docs>. Там невероятное количество полезной – и бесполезной – информации. Слева обычный поиск по справке. Работает он очень плохо, что делать, не мы писали. Почти наверняка сначала вы получите совсем не то, что хотели.

Формально правильно, а по существу издевательство © В. И. Ленин

Только не подумайте, что я сторонник учения Карла Маркса, Фридриха Энгельса и Владимира Ильича Ленина. Ни в коем случае, просто мысль очень верная и к месту. А так я твёрдый сталинист.

Справа список тем – они все написаны на удивление хорошо, толково и понятно. По-английски, само собой.

Особенность справки в том, что она совершенно незаметно и непринуждённо переходит в on-line режим. Пример:

Запускаем справку, как указано выше.
Выбираем в указателе Tkinter – это очень важная и ходовая тема

Нам предлагают

Tkinter

(module)

При выборе первого пункта мы попадём во встроенную справку и, пока будем бродить по ней, неожиданностей не будет. Впрочем, и узнаем мы оттуда не слишком много. Если мы выберем вторую строку, то нам предложат богатейшую информацию по Tkinter – но вся она будет уже on-line.

Глава вторая. Переменные

Переменные вообще. И целые в частности

Что такой переменная? Переменная – это очень просто. У переменной есть имя и у неё есть значение. Пример: *Дима – чудак на букву М.* В данном случае, *Дима* – имя переменной, а всё остальное – значение. Но мы пока ещё не дошли до таких высот абстракции и временно ограничимся числами. *Дима – полный ноль*, где-то так. Теперь пишем в Редакторе, практически дословно. Но, для наглядности, одновременно заводим и переменную с более математическим именем.

```
Dima = 0
x = 3
print x
```

У нас, просто ниоткуда, возникла переменная `x`, и мы присвоили ей значение 3. Ещё, разумеется, возникла переменная `Dima`, которой присвоен круглый ноль.

о простоте Питона
Когда я говорю, что переменная возникла *из ниоткуда*, я имею в виду следующее – в отличие от традиционных языков, нам не надо производить с именем этой переменной какие-то предварительные манипуляции. Нам не надо объявлять переменную – нам не надо уточнять, что в этой переменной будет храниться именно число.
В питоновской переменной может храниться что угодно – число, строка, список. Причём по ходу выполнения программы может меняться не только значение переменной, что нормально и понятно, но и тип этого значения.
Разумеется, такой подход очень упрощает и ускоряет процесс внесения ошибок в программу.
конец Простоты

Если быть занудными, мы написали имя переменной `x` слева от оператора присваивания в виде равенства, и переменная возникла, а после этого мы присвоили ей значение 3. Пока можете об этом не задумываться Задумаетесь потом, а теперь обратите внимание, что после присваивания переменной значения мы это самое значение переменной ещё и вывели. Проверьте, попробуйте – выведется именно три.

```
print x
3
```

Теперь немного усложним:

```
x = 3
y = 2
z = x + y
print 'z = ', z
```

Результат:

```
z =  5
```

Как нетрудно догадаться, мы присвоили значения переменным X и Y и присвоили их сумму переменной Z, значение которой потом вывели. Вывели не просто так, а красиво, с подписью. На всякий случай, повторим всю процедуру для вычитания:

```
x = 2
y = 3
z = x - y
print 'z = ', z

z =  -1
```

Обратите внимание, отрицательные числа появились и вывелись автоматически, без нашего особого внимания. Умножение немного интереснее –

```
x = 2
y = 3
z = x * y
print 'z = ', z

z =  6
```

Интереснее умножение тем, что оно легко расширяется на возведение в степень, которое по сути есть многократное умножение, для целых чисел, по крайней мере.

```
x = 2
y = 3
z = x ** y
```

```
print 'z = ', z

z = 8
```

Числа те же, результат другой. Теперь деление. С делением намного сложнее – одно целое число не обязано делится на другое. Если мы поделим 23 на 5, то получим частное 4 и остаток 3. Вроде бы, раньше так учили в начальной школе, не знаю, как сейчас. Как это записать в Питоне?

```
x = 23
y = 5
z = x / y
ost = x % y
print 'z = ', z
print 'ost = ', ost

z =  4
ost =  3
```

Косая черта делит нацело. Знак процента даёт остаток. Теперь решим не очень сложную задачу, которая, хотя и кажется оторванной от жизни, программистам довольно часто встречается. Есть трёхзначное число. Надо разобрать его на цифры. То есть на входе 321, на выходе 3,2.1. К решению будем подползать мелкими шагами – сначала решим задачу попроще. Пусть у нас на входе всего-навсего *двузначное* число. Разберём его на цифры. Задача настолько проста, что сразу приведу решение:

```
x = 32

c1 = x % 10
c2 = x / 10

print c2,c1
```

Пора ещё раз напомнить, что у переменной обязательно есть имя. Имя не обязательно состоит из одной буквы, например – x,y,z. Как вы только что увидели, имя может состоять из букв и цифр. Если совсем аккуратно – имя может состоять из букв, цифр и знака подчёркивания, при условии, что первым символом не будет цифра. Примеры правильных имён – x, y123, a_plus_b, _pi_na_tri. Примеры неправильных имён я приводить не буду, вы сами справитесь. Очень и очень важная особенность изучаемого нами языка – большие и маленькие буквы отличаются и не одно и то же. То есть x и X совсем разные переменные. Как легко догадаться, это служит

неисчерпаемым источником ошибок. Впрочем, ошибки эти легко вылавливаются.

Пример вычислений сложнее:

```
V = ((3.1415*H)/3) *(R**2 + R*r + r**2)
```

Это вычисление объёма усечённого конуса, если что. R большая и r маленькая – совсем разные переменные и обозначают, соответственно, верхний и нижний радиусы конуса.

Теперь, вернувшись к нашему примеру – который ниже в тексте, обратите внимание – нумерация цифр начинается с самой правой. Потому что если начинать с самой левой, надо заранее знать, сколько их, цифр, будет. Обдумайте.

В остальном всё в примере понятно. Если взять остаток от деления на 10 числа 32, то получим 2, что и является последней цифрой. А если поделить число нацело на 10, то получим первую и предпоследнюю цифру. Десять здесь присутствует не просто так, а как основание десятичной системы счисления.

Переходим к трёхзначному числу. С последней цифрой ничего не изменилось – это остаток от деления на 10. А что будет, если мы поделим число, пусть будет 321, на 10 нацело? Мы получим 32, двухзначное число, с которым мы уже умеем обращаться. Подставляем результат деления нацело в предыдущий код и получаем:

```
x = 321

c1 = x % 10
c2 = (x / 10) % 10
c3 = (x / 10) / 10

print c3,c2,c1
```

Результат правильный, я проверял. Последнюю строку вычислений можно записать по-другому, и, может быть, так будет даже и понятнее:

```
c3 = x / 100
```

Однако первый вариант обладает неоспоримым преимуществом с точки зрения математики – мы не решали задачу заново, а свели её к предыдущей, уже решённой. Если есть возможность, то поступайте так всегда. На самом деле, это совет не из области математики, это совет из области программирования. Ещё одна деталь, на которую следует обратить внимание, – скобки. Вроде бы всё понятно, но иногда в математике, кроме обычных круглых скобок, используют и квадратные, и фигурные. В других языках программирования при попытке их применения у вас немедленно возникнут серьёзные проблемы – вам сообщат, что вы использовали недопустимый символ. В Питоне всё будет гораздо хуже. Он всё сжуёт, но результат, скорее всего, будет не совсем тем или совсем не тем, что вы хотели. Дело в том, что в Питоне вполне допустимы во многих случаях и другие скобки, кроме круглых. Поэтому запомните – в вычислениях и в выражениях используйте *только* круглые скобки.

А теперь об очень важном, что мы незаметно пропустили. Пока это для вас не очень важно, но там, потом, в реальной программистской жизни вырастет до гигантских, монстрообразных размеров. Мы предположили и приняли на веру, что числа наши будут именно двухзначными или трёхзначными. Если они такими не окажутся, то программа наша выдаст неверный результат. А они такими не окажутся, просто по закону подлости – он, закон этот, беспощадно работает. Поэтому в реальной, а не учебной жизни, все данные на входе надо проверять.

Кстати, вопрос – вот есть у нас очень простая формула $\rho = \frac{m}{V}$. Это, если вы не знали, да ещё и забыли – расчёт плотности вещества. Понятно, что числа там, вообще-то, дробные, о них мы будем говорить в следующем разделе. Но если взять их целыми и V неожиданно окажется равным нулю? Мне кто-то говорил, что на ноль делить нельзя. Что будет? Проверьте.

Переменные дробные.
Или, по-программистски, плавающие

Дробные переменные у нас уже встречались, там, где мы считали объём усеченного конуса, но введение этих чисел в оборот прошло как-то незаметно. Сейчас мы подойдём к этому вопросу основательно и систематически.

Сейчас будет длинное математическое вступление. Вы можете его пропустить и перейти сразу к практической части, ничего не потеряете.
В чистой математике сначала рассматривают только целые числа. Не то чтобы была ещё и какая-то *грязная* математика, так принято выражаться, ничего личного. Просто есть ещё и *прикладная* математика. Далее понятие числа распространяют на рациональные числа, то есть числа, которые можно представить в виде натуральной дроби, например $\frac{1}{3}, \frac{3}{13}, \frac{127}{19}, \frac{5}{125} = \frac{1}{25}$. Последнюю дробь можно сократить, но это совершенно не важно. Потом в дело вступают *иррациональные* числа. Иррациональное число – число, как ни странно, то, которое не рациональное. Иначе говоря – не представимое в виде натуральной дроби, например $\sqrt{2}$. Почему именно корень из двух? Потому что ещё древние греки под командованием Пифагора доказали иррациональность этого числа. И немедленно, от радости, принесли в жертву сто быков. Тут для меня остаются нерешёнными некоторые вопросы, как то – чему радоваться? – чем быки виноваты? – что с мясом сделали? Далее надо понимать один несложный момент. Вот есть у нас банка мёда, и добавили к ней, скажем так, ложку дёгтя. Вы знаете, что такое дёготь? Путаетесь в показаниях? Это народная русская пословица. Ну пусть не дёгтя. Пусть известно чего. Мёд уже не мёд. А если к ведру меда? А если к бочке мёда? Я понимаю, что в реальности никто не заметит, но пословица утверждает иное.

Философский вопрос
Есть куча песка. Мы отнимаем из неё одну песчинку. Осталась ли куча песка кучей песка? Мы продолжаем отнимать из кучи по одной песчинке. Когда куча перестанет быть кучей?
Как ни странно, вопрос этот был задан в мохнатые семидесятые в журнале *Техника – Молодёжи*, и там же был дан на него вполне разумный и обоснованный ответ.
Обдумайте.
конец Философского вопроса

Короче, если у нас есть иррациональность, то любое выражение, в котором она присутствует, абсолютно точно иррациональным и останется. Иррациональность заражает. То есть число $\frac{213\sqrt{2}}{42}10!$ неизбежно будет

иррациональным, хотя всё в нём, кроме корня из двух, – рациональнее некуда.

Математическое уточнение, последнее. Все вместе числа, и целые, и рациональные, и иррациональные, в математике называются *действительные*. В программировании немного не так – числа делятся на *целые*, с ними мы уже ознакомились, и *плавающие*. Почему они так называются, не важно, это было давно и все забыли, по сути это и есть наши математические действительные числа.

Теперь переходим к практической части. Как выглядят плавающие числа? Вот старый целый пример с результатом:

```
x = 7
y = 2
z = x/y
print 'z = ', z

z = 3
```

Теперь внесём небольшое изменение.

```
x = 7.0
y = 2
z = x/y
print 'z = ', z

z = 3.5
```

Если вместо 2 написать 2.0, результат будет тем же – то есть вполне разумным и ожидаемым. Всё, что для этого понадобилось – употребить в записи числа десятичную точку. Все математические операции работают после этого вполне ожидаемым способом – то есть именно так, как они работают в обычной арифметике. Сложение:

```
z = 2.3 + 1.2
print 'z = ', z

z = 3.5
```

Вычитание:

```
z = 1.234 - 3.99
print 'z = ', z
```

```
z = -2.756
```

Деление у нас уже изучено, займёмся умножением, точнее, его расширением – возведением в степень.

```
z = 3 ** 2.5
print 'z = ', z

z = z =  15.5884572681
```

Напоминаю, что $x^{2.5} = x^2\sqrt{x}$. Обратите внимание на невнятный формат вывода – его можно окультурить, но пока это не главное. Главное – если одно из чисел в выражении имеет десятичную точку, то есть является плавающим, значит и результат вычисления выражения тоже будет иметь плавающий тип.

В математике есть понятие функции. В программировании тоже есть понятие функции, но это совсем другие функции. Обычные, всем известные математические функции – синус, тангенс, логарифм. Все эти функции, разумеется, в Питоне есть. Однако если мы напишем просто и без затей:

```
x = 1
z = sin(x)
print 'z = ', z
```

то результат будет не вполне удовлетворительным:

```
Traceback (most recent call last):
  File "D:\start.py", line 5, in <module>
    z = math.sin(x)
NameError: name 'math' is not defined
```

В переводе на человеческий – Питон понятия не имеет, что это за синус и где его искать. Надо поправить, добавлением всего одной строки и исправлением другой:

```
import math  #!!!

x = 1
z = math.sin(x)  #!!!
print 'z = ', z
```

```
z =  0.841470984808
```

Слово **import** означает, что мы подключаем какой-то модуль. Слово math – что это конкретно математический модуль с математическими функциями. Подробнее это всё обсудим позже. А пока запомните, это придётся делать часто.

А пока – то, без чего можно обойтись, но иногда это оказывается полезным. Можно присваивать значения переменным так:

```
a = 2; b = 3
print 'a = ',a,' b = ',b

c,d = a,b
print 'c = ',c,' d = ',d

a =  2  b =  3
c =  2  d =  3
```

Здесь мы видим целых два новшества. В первой строке сразу два оператора присваивания, разделённые точкой с запятой, – так можно. Но это именно два отдельных оператора присваивания. А дальше мы присваиваем в одном операторе значения двум переменным сразу – тем двум, что слева от равенства значения, тех двух, что справа. Никакой пользы, кроме вреда и путаницы, я от этого не вижу, разве что в следующей ситуации:

```
a,b = b,a
print 'a = ',a,' b = ',b

a =  3  b =  2
```

Одно движение руки – и две переменные обменяли свои значения. Быстро, просто и наглядно.

Строки

Строковые переменные здесь чисто для полноты изложения. Строки – в первом приближении – это очень просто. Потом, как и всегда, вылезают подробности, но это потом:

Потом вам будет плохо, но это ведь потом © Макароныч

```
a = 'abcd'
```

```
b = 'efgh'
s = a + b
print 's = ', s

s =  abcdefgh
```

Это мы сложили две строки. Обратите внимание, что результирующая строка после вывода кавычек не имеет, ни внешних, ни внутренних. Вычесть или разделить строки, к сожалению, нельзя, но можно умножить – но только на целое число.

```
a = 'abc '
s = a * 3
print 's = ', s

s =  abc abc abc
```

Всё должно быть понятно. Со строками пока закончим.

Ввод

Вывод уже был, теперь ввод. С глубокой печалью должен признать, что с вводом данных не всё в Питоне хорошо и не всё в Питоне удобно. В Паскале и других традиционных языках это реализовано проще. Но – *а куда ты денешься с подводной лодки* © Анекдот. Кроме шуток – другой системы ввода у меня для вас нет.

Начнём с самого простого – попросим ввести имя, отчество и фамилию, а потом всё это объединим и выведем. Чтобы не отвлекаться на не относящиеся к делу подробности, всё будет происходить на американском языке. Питон изначально заточен под латинскую кодировку. Все остальные языки, хоть кириллица, хоть иврит, возможны, однако требуют некоторых дополнительных заклинаний. Пока отвлекаться не будем.

```
firstName  = raw_input('first name = ')
secondName = raw_input('second name = ')
lastName   = raw_input('last name = ')

print 'person is ', firstName + ' ' + secondName + ' ' + lastName

first name = John
second name = F.
last name = Kennedy
person is  John F. Kennedy
```

Что мы видим? Видим вот такую схему:

```
something = raw_input('введите что-то = ')
```

Слева от равенства имя строковой переменной. Справа от равенства вызов стандартной функции ***raw_input***. В скобках у неё текст – приглашение к вводу, которое будет автоматически выведено. В ответ на приглашение вводим строку и нажимаем <Enter>. Всё понятно и, может быть, даже удобно – хотя надо всю конструкцию запомнить наизусть.

вставка об Американском языке и генетике
Причём тут генетика? Как объясняют популяризаторы науки, часть генов человека и других животных не используется. Причём часть эта составляет где-то 99%. Просто появляется в цепочке ДНК определённая последовательность аминокислот, означающая – *дальше мусор, в размножении не участвует*. А потом другая последовательность – *отсюда работает и в размножении участвует*. Абсолютно точный аналог комментариев. И таким образом закомментировано девяносто девять процентов генетического кода человека, что, конечно, является перебором. Хорошие программисты рекомендуют избегать избыточных комментариев. С другой стороны, интересно, а что же там такое хитро закомментировано? Третий выпуклый глаз? Способность к левитации? Зелёная пупырчатая кожа? Достойный человека хвост?

Теперь об американском языке: существует ли он и как мы должны с этим бороться. Был такой американский лингвист — Сводеш. Знаменит он тем, что составил список ста понятий, которые предположительно есть в каждом языке, по крайней мере, в каждом индоевропейском языке. Список так и называется – Список Сводеша. Список составлен, само собой, на английском.
Теперь, если мы хотим математически, то есть численно, оценить расстояние между двумя языками, мы переводим список на оба эти языка. Чем больше пар слов непохожих и откровенно разных – тем языки дальше. Для тех, кому это кажется очень грубым методом, Сводеш рекомендует давать бóльший вес словам из начала списка.
Так вот, вычисленное по этой методике расстояние между британским и американским диалектами английского языка равно единице. Отличие в слове *камень*. В британском это *stone*, а в американском *rock*. На образование этой большой разницы потребовалось двести лет.

```
# конец Вставки
```

Теперь усложняем. Введём два целых числа, пусть x и y, и выведем $z = x^y$. Питону всё равно, в какую степень возводить, целую или дробную, но для наших учебных целей ограничимся целыми числами, да и проверять результат на бумажке или калькуляторе будет проще. Проверять надо всегда – какой бы простой ни была программа, программист всегда найдёт способ ошибиться. Вот что у нас получилось:

```
x = int(raw_input('x = '))
y = int(raw_input('y = '))

z = x**y
print 'z = ', z

x = 2
y = 10
z =  1024
```

Всё работает, калькулятор не нужен, программист обязан знать степени двойки до... – ну до какой-то степени. В предыдущем опыте мы ввели строку и на этом всё закончилось, потому что именно строка нам была и нужна. Теперь мы вводим строку и применяем к ней функцию **int**, которая делает из строки целое число. Мне такой подход не нравится, но так уж оно в Питоне устроено. Заучите наизусть эту конструкцию, использовать её придётся много раз. Теперь то же самое, но для расчёта площади круга по радиусу. Радиус имеет право быть и дробным. Для тех, кто настолько окунулся в программирование, что забыл о геометрии, напоминаю: $S = \pi R^2$. Кстати, напоминаю, что указ Петра Первого о запрете жениться не знающим геометрию никто формально до сих пор не отменял.

```
import math

R = float (raw_input('R = '))
S = math.pi * R**2

print 'S = ', S

R = 2.5
S =  19.6349540849
```

Легко заметить, что место функции **int** заняла функция **float**, переводящая строку в число с плавающей точкой. Ещё раз обращаю внимание на строку

`import math`, позволяющую совершенно бесплатно использовать константу π.

надоело – почему она называется плавающая точка
Всё-таки надо объяснить, почему числа называются плавающими и что такое эта плавающая точка. Объяснять буду чудовищно примитивно, зато понятно. Возьмём – чисто для конкретности примера – число длиной в четыре байта. Туда легко можно затолкать шестизначное число – например, 123456 – и ещё останется очень много места. Могли бы записать и больше знаков – девять или десять, а сколько, кстати, – посчитайте. Хотя нет, считать не надо.
Не мелочись, Наденька! © Ирония судьбы
Мы пойдём другим путём!
© Владимир Ильич Ленин Надежде Константиновне Крупской
У нас есть шесть цифр числа. Кстати, это называется красивым словом *мантисса*. Из неиспользованного места мы выделим две цифры под так называемый *порядок*. Как теперь понять, что за число у нас написано? Очень просто. $\textit{число} = \textit{мантисса} \times 10^{\textit{порядок}}$.
Если порядок равен нулю, то имеем в результате исходное значение мантиссы, то есть $\textit{число} = 123456 \times 10^0 = 123456 \times 1 = 123456$.
Если порядок плюс два, то $\textit{число} = 123456 \times 10^2 = 12345600$.
Про плюс сказано не просто так, порядок может быть и отрицательным, пусть минус четыре, $\textit{число} = 123456 \times 10^{-4} = 123456 \times 0.0001 = 12.3456$.
Как видите, записать можно почти любое число – только значащих цифр будет всё равно только шесть. С другой стороны, больше точных цифр бывает только в денежных расчётах. Всё остальное измерено с точностью плюс-минус пол-лаптя.
конец Надоела

В Питоне есть ещё и другие числа – рациональные, комплексные и совсем другие, но, чтобы не нарушать плавность потока изложения, я перенёс всё это в приложения.

Игра случая

Возможно, вы слышали, есть такая наука – теория вероятностей. Когда я учился математике, она мне очень нравилась. В программировании она тоже применяется. Если вы, прямо и в лоб, спросите меня – а где она применяется? – то я буду в некоторой задумчивости от избытка

возможных ответов. Бывают собственно вероятностные задачи, где требуется определить вероятность некоторого события. Чаще мы имеем какое-то измеренное значение, на точность измерения которого влияет много разных случайных помех. Надо определить диапазон, в котором на самом деле наше значение находится.

Само собой, теория вероятностей применяется в программировании игр, там без этого вообще никак. Ещё она применяется в автоматической генерации тестов, но это скучно и уныло. А ещё есть такой метод Монте-Карло. Вообще-то Монте-Карло – это такой городишко, куда все приезжают играть в азартные игры на деньги. Звучит не очень серьёзно. Однако в математике это очень серьёзный метод, применяемый для решения очень серьёзных задач. В первую очередь – для моделирования процессов радиоактивного распада.

Знать основы теории вероятностей в любом случае для любого программиста абсолютно необходимо. Настоятельно рекомендую к прочтению мою книгу того же издательства:

Простая математика для простых программистов.

Теперь несколько очень простых примеров.

```
import random

x1 = random.randint(0,5)
x2 = random.randint(0,5)
x3 = random.randint(0,5)
x4 = random.randint(0,5)
x5 = random.randint(0,5)

print x1,x2,x3,x4,x5
```

```
0 1 5 2 1
```

Чисто случайно – нет, я не жульничал – выпал очень удачный и наглядный результат. Мы заказали случайное целое число в диапазоне $\{0,5\}$. И мы получили, и ноль, и пять. Причем единица повторилась дважды, но ни одной тройки не было. Нам, разумеется, пришлось импортировать модуль `random`, который и обеспечивает работу со случайными числами. Хотя я и пишу эту книгу для тех, кто, предполагается, никаких других языков программирования не знает, должен заметить – работа со случайными

числами в Питоне устроена много лучше, проще и логичнее, чем в других языках.

Показывать полезность случайных чисел мы будем постепенно, по шагам. Для начала напишем программу, которая выводит на экран два случайных целых числа и предлагает их сложить. Потом программа выводит правильный результат. Заметьте, программа не информирует клиента, является ли ответ правильным или как. Этим мы займёмся в следующей главе. Это называется *условный оператор*. А пока предлагаю вот такой код:

```
import random

x = random.randint(10,99)
y = random.randint(10,99)

print x, '+', y, '= ?',
raw_input()
print 'result is ', x+y
```

```
81 + 22 = ?
result is  103
```

На что ещё обратить внимание? На незаметную запятую в конце первого оператора вывода `print x, '+', y, '= ?',` – это благодаря ей не происходит немедленный переход на новую строку. Также интерес представляет строка `raw_input()`. Видимого и немедленного результата она не даёт – зато она молча ожидает нажатия клавиши <Enter> . Запомните, это вам ещё много раз пригодится.

Случайные числа бывают не обязательно целыми.

```
import random

x1 = random.random()
x2 = random.random()

print x1, x2
```

```
0.726732416853 0.967583829512
```

Эта функция возвращает действительное, то есть плавающее, число в диапазоне от нуля до единицы. А что делать, если вам надо, к примеру, от нуля до ста, или, того хуже, от ста до трёхсот?

Однажды к товарищу Сталину пришёл высокопоставленный работник Центрального Комитета, отвечавший за работу с творческой интеллигенцией. Он стал жаловаться на писателей – пьют, уводят друг у друга жён, потом бьют морды и пишут доносы.

– Других писателей у меня для вас нет, – прервал его Сталин, – работайте с тем, что есть. © Быль

Вот и у нас то же самое – других случайных чисел у нас нет, придётся работать с тем, что есть. Запрашивать случайные числа по новой мы не будем, используем те, что есть. Если нам нужен диапазон пошире, то, может быть, поможет умножение? А если диапазон хочется сдвинуть – то сложение? Сдвинуть – не в смысле уменьшить, это значит, что у нас есть на руках случайное число в диапазоне (0,1), а нам надо число в диапазоне (10,11). Далее примеры. Сдвинуть, расширить, сдвинуть и расширить.

```
import random

x1 = random.random()

x_10_11 = x1 + 10
x_0_100 = x1 * 100
x_100_300 = x1*200 + 100

print x_10_11
print x_0_100
print x_100_300

10.7360001241
73.600012407
247.200024814
```

Обдумайте – если вам придётся работать со случайными числами, то это самые основы. Кстати. В модуле `random` ещё очень, очень много функций для получения случайных чисел на все случаи жизни. Поинтересуйтесь.

Глава третья. Условный оператор

Что это такое

До сих пор наши небольшие программы выполнялись несколько предсказуемо – от начала до конца. Если в программе было десять строк, выполнение программы начиналось с первой строки и заканчивалось на десятой. Такие программы просты и понятны, но, к сожалению, мало полезны. Настоящие программы выполняются заранее непредсказуемым способом. Вот сейчас, мало-помалу, мы и начнём писать настоящие программы.

Традиционно начнём с самой простейшей из простых задач. Вводим число, целое, плавающее, неважно. Если оно больше нуля – так и пишем – положительное. Если меньше или равно нулю – ничего не пишем.

В то время мы ничего, кроме шампанского, не пили; не было денег — ничего не пили, но не пили, как теперь, водку.
© Л. Н. Толстой «После бала» – пример простого условного оператора

```
x = int(raw_input('x = '))

if x > 0:
    print ('x positive')
```

Заметьте – не я поставил пробелы во второй строке. Они сами поставились. Редактор Питона, узнав в лицо условный оператор, автоматически при нажатии клавиши <Enter> вставляет в начало следующей строки несколько пробелов. Сколько именно конкретно – не важно, пусть будет четыре, я согласен. Далее будет не очень серьёзный разговор об очень серьёзных вещах.

Надо усвоить очень важную идею. В других языках программирования количество пробелов никакой роли не играет – если можно поставить один пробел, можно поставить их сколько угодно. Ту же роль играет переход на следующую строку – в начале следующей строки можно поставить сколько угодно пробелов, это ничего не меняет. Разумеется, пробелы программисты ставят, и их – пробелы – от них – программистов, начальники даже требуют ставить.

В каждой уважающей себя фирме есть документ, описывающий стандартный вид программного кода, то есть сколько пробелов надо вставить в следующей строке после условного оператора. Тех, кто не подчиняется, наказывают – это только в хороших фирмах, разумеется. Но всё это только для того, чтобы программный код легко читался и выглядел красиво. Некрасивый самолёт не полетит, некрасивый код работает известно через что. В Питоне всё очень по-другому – некрасивый код не заработает.

Хорошо оформленный код на Паскале, он же Дельфи, выглядит так:

```
if x > 0 then begin
   y:=x*x;
   z:=y - 1;
   if z div 2 = 0 then begin
      ShowMessage('Всё пропало!');
   end;
end;
```

Мы можем его как угодно испортить – с внешней точки зрения. Например, так:

```
if x > 0
then
begin
          y:=x*x;
        z:=y - 1;
              if z div 2 = 0
then begin
ShowMessage('Всё пропало!');
  end;
        end;
```

Не надо так делать, пожалуйста! Тем не менее, выглядит это ужасно, а работает хорошо, точно так, как и прежде.

Так вот, в Питоне так нельзя. Если вы испортите код вот таким образом

```
if x > 0:
print ('x positive')
```

то при попытке трансляции немедленно получите сообщение об ошибке. Ещё раз – так нельзя, пробелы имеют принципиальное, очень важное значение. Запишем условный оператор в условной, абстрактной форме:

```
if <УСЛОВИЕ>:
<ПРОБЕЛЫ><ДЕЛАТЬ ЧТО-ТО>
```

Обратите внимание, обязательной частью здесь является слово **if**, двоеточие – это важно, и пробелы! А вот остальные два пункта – условие и выполнение чего-то могут быть разными, или говоря по-научному – вариабельными.

Однако вы уже проверили нашу небольшую программу и убедились, что она работает. Теперь будем усложнять задачу.

Условный оператор. Сложнее

А что, если нам надо в случае выполнения условия выполнить не один оператор, а несколько? Например, увеличить переменную X на единицу? Сначала объясняю словами – все операторы, записанные – графически, визуально – на том же уровне, будут выполнены. Непонятно? Теперь на примере:

```
x = int(raw_input('x = '))

if x > 0:
   print ('x positive')
   x = x +1

print x

x = 3
x positive
4
```

В случае положительного числа выполняется и оператор вывода и арифметический оператор, который увеличивает икс на единицу. А теперь о главном. Что будет, если немного, совсем чуть-чуть, изменить программный код:

```
if x > 0:
   print ('x positive')

x = x +1
```

Проверьте. На всякий случай, подскажу ответ – переменная X будут увеличена на единицу, независимо от того, положительная она или

отрицательная. Это потому, что теперь арифметический оператор находится *на одном уровне* с условным, и выполняется независимо от него. Потренируйтесь – поглядите на этот текст, и ответьте, чему будет равен X:

```
x = 1

if x > 0:
    x = x + 2
    x = x + 3

x = x + 4

if x > 9:
    x = x - 10

print x
```

Ответ: ноль.

Там, где в общей форме условного оператора написано `<СДЕЛАТЬ ЧТО_ТО>`, может быть написано почти что угодно, лишь бы оно было записано с отступом и, желательно, с одним и тем же. В частности, там может быть и ещё один, вложенный, условный оператор.

```
if x > 0:
    if x > 10:
        print 'I''m here!'
```

Результат понятен – при X больше, чем десять, будет выдано сообщение. Есть ли смысл в такой записи? Очевидно, нет – второе условие сильнее и можно оставить только его. Но это только потому, что других условий мы пока не видели. Вот простой, абсолютно бесполезный и бессмысленный пример, включающий все возможные условия:

```
if x > 0:
    if x < 0:
        if x >= 0:
            if x <= 0:
                if x == 0:
                    if x <> 0:
                        pass
```

Отступы, ясное дело, добавлял редактор, причём автоматически. В самом конце нас ожидает, если вы его не забыли, пустой оператор – всё равно мы никогда туда не доберёмся – а почему? Сочетание знака *больше* и знака

равенства обозначает – никогда не догадаетесь – *больше или равно*. Предпоследнее условие, два знака равенства подряд – всего лишь проверка на равенство. Дело в том, что в Питоне символ просто равенства уже занят под оператор присваивания. А последнее условие, знаки *меньше* и *больше*, вместе обозначают *не равно*. Справа не обязательно должно быть конкретное число, можно и так:

```
#  ввести x - это комментарий, не забыли?
#  ввести y

if x <> y:
    print 'не равно!'
```

Точно так же, вместо отдельно взятых переменных могут быть и выражения, например:

```
if x**2/a**2 + y**2/b**2 == 1:
```

Те, кто силён в математике, узнают здесь проверку на принадлежность точки к эллипсу. А теперь проверка делимости числа, целого разумеется, на 13:

```
if x % 13 == 0:
    print 'Ok'
```

Вернёмся к нашему бесполезному примеру со вложенными условными операторами:

```
if x > 0:
    if x > 10:
        print "I'm here!"
```

Если мы его слегка подправим, то смысл появится.

```
if x > 0:
    if x < 10:
        print 'I"m here!'
```

Теперь мы проверяем выполнение условия $1 \le x \le 9$, что может пригодиться – и наверняка пригодится. А нельзя записать это как-нибудь попроще? Разумеется, можно.

```
if (x > 0) and (x < 10):
    print 'Ok'
```

```
#  или так

if (x >= 1) and (x <= 9):
    print 'Ok'
```

Если между двумя отдельными условиями стоит волшебное слово **and**, то условие в целом считается выполненным, тогда и только тогда, когда выполнены оба условия. Обратите внимание на *обязательные* скобки вокруг условий. А если наши интересы совсем противоположны, то есть нам нужно не число в диапазоне $1 \le x \le 9$, а вне его? То есть X меньше единицы или X больше девяти? Так и пишем.

```
if (x < 1) or (x > 9):
```

Задумайтесь над тем, что любое сложное условие можно записать в разной форме – как минимум в двух. Запись может меняться, а смысл остаётся прежним. Пример двух равносильных условных операторов :

```
x = 1

if (x > 0) and (x < 5):
    print 'Ok'

if not ((x < 0) or (x > 5)):
    print 'Ok'

Ok
Ok
```

Сейчас мы совершено непринуждённо познакомились с ещё одним зарезервированным словом, которое очень часто оказывается нужным:

```
if x > 1:

if not (x > 1):   #  совершенно то же самое, что if x <=1
```

Казалось бы, зачем это вообще нужно? Иногда это упрощает программирование, избавляя от необходимости думать. Чем больше программист думает, тем больше ошибок мы в итоге будем иметь. Вспомним пример, приведённый чуть ранее, и обратное к нему условие:

```
if (x >= 1) and (x <= 9):
if (x < 1) or (x > 9):
```

Пришлось думать. А можно и так, немного громоздко, но без напряжения головного мозга.

```
if not ((x >= 1) and (x <= 9)):
    pass
```

То есть берём все вместе предыдущие условия в скобки и ставим впереди `not`. И думать не надо. Хотя всё-таки немного надо, и вот о чём. Значение слова `or`, оно же *если*, отличается у нас от общепринятого бытового. В быту *или-или* означает что-то одно и только одно.

Или я её веду в ЗАГС, или она ведёт меня к прокурору
© к/ф Кавказская пленница

Ещё проще, вот *или налево, или направо* – тут уже точно надо выбирать что-то одно. Математическое *или* работает совсем по-другому. Оно означает – или налево, или направо, или и то и другое вместе. То есть и *в ЗАГС,* и *к прокурору* одновременно. Как страшно жить математикам. Обдумайте. Пробудите свое воображение.

Условный оператор. Ещё сложнее

Пока мы обсуждали содержание условного оператора. Поговорим и о форме. Вспомним, с чего мы начали – программа, определяющая, является ли число положительным:

```
x = int(raw_input('x = '))

if x > 0:
    print ('x positive'
```

Самое первое желание, приходящее в голову при взгляде на это творение, – пусть оно сообщает, когда число *не положительное*, то есть число меньше или равно нулю. Примитивное, но работающее решение:

```
x = int(raw_input('x = '))

if x > 0:
   print ('x positive')
if x <= 0:
   print ('x NOT positive')
```

Если вам кажется, что это *слишком* примитивно, то я с вами согласен. Зачем вторая проверка? Вариантов всего два – или положительное, или *не* положительное. Мы проверили число на положительность. Зачем ещё раз проверять его на отрицательность, точнее не положительность? Ведь если первый ответ был *да*, то второй ответ будет *нет*, и наоборот. Встречаем расширенную форму условного оператора.

```
if x > 0:
   print ('x positive')
else:
   print ('x NOT positive')
```

В обобщённом виде это выглядит так:

```
if <УСЛОВИЕ>:
<ПРОБЕЛЫ><ДЕЛАТЬ ЧТО-ТО>
else:
<ПРОБЕЛЫ><ДЕЛАТЬ ЧТО-ТО ДРУГОЕ>
```

Разумеется, после **`else`** тоже могут быть записаны не один, а много операторов. И не забывайте двоеточие. И подумайте о том, что выполняется или то, или другое. Или то, что после **`if`**, или то, что после **`else`**. Теперь немного усложним задачу. До сих пор мы только определяли, положительное число или наоборот. Задача несколько выдуманная из головы, но уж раз мы её поставили, скорее всего, она бы стояла так – является ли число положительным, отрицательным или нулём. В формат **`if-else`** это не укладывается. Писать три отдельных условных оператора тоже не хочется. Если вам кажется, что это ерунда, не заслуживающая внимания, то вы глубоко неправы.

Вариантов может быть не три, как у нас, а больше – семь, восемь. Почему не больше? Потому что, если их больше, пора применять другие средства, а не условный оператор, сколь угодно продвинутый. Об этом мы поговорим позже. Проверка условия может быть весьма затратной по времени, но это редко и не главное. Главное то, что такой код – с многими условными операторами на одном уровне – очень способствует размножению ошибок. Теперь вариант решения с привлечением новых средств языка:

```
if x > 0:
   print ('x positive')
elif x < 0:
```

```
   print ('x negative')
else:
   print ('zero')

pass
```

Присмотритесь и обдумайте. Загадочное словечко **elif**, скорее всего, является склейкой понятных слов **else** и **if**. Что будет, если X больше нуля? Правильно, после вывода соответствующего сообщения выполнится пустой оператор **pass**. В смысле – выполнится и ничего не сделает, как ему и положено. Если X меньше нуля – сообщение и пустой оператор. И только если оба условия не сработали, попадаем в секцию **else**. Обратите внимание на важный момент – в любом случае всё кончается пустым оператором. Говоря по-другому, у нашей конструкции только один выход. Это хорошо.

А теперь доработаем предыдущую программу – ту, которая выводила два случайных числа, просила ввести их сумму, а потом выводила правильный ответ. Новая логика её работы такая – если ответ правильный, то программа так и говорит, а если нет, то опять-таки так и говорит и, в придачу, выводит правильный ответ.

```
import random

x = random.randint(10,99)
y = random.randint(10,99)

print x, '+', y, '= ?',
z = int(raw_input())

rightZ = x + y

if z == rightZ:
    print 'you are Ok'
else:
    print 'something''s wrong'
    print 'sum is ', rightZ
```

В общем и целом, всё должно быть понятно. Задумайтесь над тем, как небольшое изменение алгоритма работы программы заметно меняет её текст.

Не очень сложное задание. Два

Есть целое число, это номер года нашей эры – например, 1937. Напишите условный оператор, который определяет, является ли год високосным. Для тех, которые думают, что високосный год – тот, который делится на четыре, объясняю.

Год високосный, если:

делится на 4
при этом не делится на 100
или
при этом делится на 100 и первые две цифры делятся на 4

То есть:

1899	нет
1900	нет
1902	нет
1903	нет
1904	да
…	
2000	да

Вот ещё одна, очень хорошая, годная задача на целые числа и условный оператор. Кроме того, неплохо бы и подумать о спасении души. Задача называется – вычисление даты Пасхи. Поскольку у нас тут сплошная, извините, толерантность, то вспомним, что Пасхи бывают разные – православные, католические и, прошу прощения, еврейские. Это важно. Ещё тридцать лет назад, когда в книгах по программированию – например многотомнике Кнута – ставилась эта задача, в описании алгоритма обязательно уточнялся важный нюанс.

Если, по несчастливой случайности, дата христианской пасхи совпадёт с датой еврейской, то христианская пасха переносится на неделю вперёд. По нынешним временам это как-то нетолерантненько. Такие формулировки недопустимы. Перенос даты теперь прошит в алгоритм каким-то неявным способом. Ещё – алгоритм вычисления выдумал Карл Фридрих Гаусс, тот, которого называли *Король Математиков*. А для изучавших теорию вероятностей – тот, который *гауссово распределение*.

Алгоритм есть в Википедии, но там он настолько подробно расписан на чём-то, очень похожем на Паскаль, что даже неинтересно. Этот вариант я взял с сайта *mooseum.ru*, у них ещё почему-то лось на заставке. Вот алгоритм:

```
Разделить номер года на 19 и определить остаток от деления a.
Разделить номер года на 4 и определить остаток от деления b.
Разделить номер года на 7 и определить остаток от деления c.
Разделить сумму 19a + 15 на 30 и определить остаток d.
Разделить сумму 2b + 4c + 6d + 6 на 7 и определить остаток e.
Определить сумму f = d + e.
Если f ≤ 9, то Пасха будет праздноваться (22+f) марта;
если f > 9, то Пасха будет праздноваться (f-9) апреля.
```

Далее понятно:

```
Для перевода на новый стиль дату, как известно, нужно сдвинуть вперёд на
13 дней в 20-м и 21-м веках.
```

А вот важное уточнение, я об этом и не знал:

```
Если Пасха совпадает с праздником Благовещения (7 апреля), то она
называется Кириопасха (Господня Пасха)
```

© *mooseum.ru*

Закодируйте. Для католической и еврейской пасхи алгоритмы найдите сами. Обязательно запрограммируйте. Мы же толерантные.

Очень сложное задание

Я однажды условия этого задания уже написал. Это было в моём учебнике по Паскалю. Это моя первая книга и, с точки зрения Дорогого Редактора™, лучшая. Потому что эта книга лучше всех продаётся. Сейчас я не буду выдумывать ничего нового. Если вы ту книгу не читали, то вам абсолютно всё равно, что там было написано, а если читали – то вам будет интересно сравнить Паскаль и Питон.

Начинается всё достаточно безобидно. Квадратное уравнение, как всем, конечно, известно, это вот такая штуковина: $ax^2 + bx + c = 0$. В идеальном случае, оно имеет два корня, их традиционно обозначают x_1, x_2. На всякий

случай – корень уравнения, это такое число, при котором уравнение обращается в равенство, то есть левая часть становится равной нулю. Проще квадратного уравнения может быть только линейное, это такое, в котором даже второй степени икса нет:

$$3x-15=0$$

Ответ очевиден. Корень уравнения

$$x=\frac{15}{3}=5$$

При этом значении икса левая часть уравнения становится нулём и равняется правой. Извините, если мои пояснения кажутся вам обидными. Много общаясь с молодыми программистами, я уже очень давно обнаружил, что они не то что книг не читают – а зачем? – они ещё и по-русски писать не умеют, хотя в этом есть какая-никакая практическая польза. Впрочем, по-английски они писать не умеют тоже. Новая волна программистов, похоже, не знает уже и математики. Извините, отвлёкся.

– Извините, что помешал вам ваши деньги прятать
© к/ф Любовь и голуби. А неплохой фильм, посмотрите, мне сначала очень не нравился, а потом понравился. *Наша кошечка сначала не любила пылесос, а потом ничего – втянулась.*

Теперь то же самое, но для квадратного уравнения. Есть уравнение

$$2x^2+5x+2=0$$

Его корни $x_1=-0.5, x_2=-2$. Проверьте.

Теперь самая главная формула. Её изобрели ещё в древнем Вавилоне. Вы знаете, как жили древние люди в древнем Вавилоне? И я нет. А квадратные уравнения они решать умели.

Его пример – другим наука,
Но, Боже мой, какая скука... © А.С. Пушкин «*Евгений Онегин*»

Вот она, эта волшебная формула, хотя древние вавилоняне записывали её клинописью на глиняных табличках:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Быстро программируем.

```
A = 2; B = 5; C = 2

d = B**2 - 4*A*C

x1 = (-B + d**0.5)/ (2*A)
x2 = (-B - d**0.5)/ (2*A)

print A,B,C
print 'x1 = ', x1, '  x2 = ', x2

2 5 2
x1 =  -0.5   x2 =  -2.0
```

Здесь обязательно надо кое-что прокомментировать. В реальном программировании, это когда программируют ради денег, а не почему-то ещё, никогда не пишут программ для решения квадратного уравнения. Нет, не сомневайтесь, находить корни квадратного уравнения приходится во время выполнения одной программы тысячи и даже миллионы раз, но это всегда оформляется как *функция*. Но по плану, который я составил для этой книги, функции у нас будут позже. Так что пока пишем так, как пишем. Потом ещё раз вернёмся и перепишем.

Хорошая новость – программа работает и вычисляет правильные корни. Теперь плохие новости. Та штуковина, которая под радикалом, то есть под квадратным корнем, называется *дискриминант*. Если дискриминант больше нуля, то всё отлично – как в нашем случае. Если он равен нулю, то корней не два, а один, или, если больше нравится, – два, но совпадающих. Если дискриминант меньше нуля, то всё пропало – корней нет вообще. Или говоря по-другому, корни есть – но *комплексные*, а нам это не нужно. Но по сути проблема не в том, что корней нет. Проблема в том, что программа наша в этой ситуации немедленно рухнет, упадёт и завершится аварийно. А правильная программа не имеет права падать ни при каких обстоятельствах.

Продолжаем разговор. А что будет, если $a = 0$? Хочется немедленно объявить, что корней нет, потому что на ноль делить нельзя. Это неправильный ответ – это всего лишь означает, что наша формула не совсем подходит. Другими словами – это проблема формулы, а не уравнения. Уравнение оказывается не квадратным, а линейным. То есть уравнение $ax^2 + bx + c = 0$ при $x = 0$ немедленно превращается в уравнение $bx + c = 0$ и мы легко находим его единственный корень $x = \frac{-c}{b}$.

Третий абзац и третий случай. А что будет, если $a = 0$ и $b = 0$? Что будет, если, говоря языком формул, $c = 0$? Это зависит от того, чему равно c. Если $c \neq 0$, то корней нет. Если равно нулю, то корнем нашего уравнения является любое число, то есть – абсолютно любое.

Теперь, перед тем, как приступить к программированию, напишем для нашей ещё не написанной программы тестовые примеры. Я уже говорил – в других книгах – и ещё раз повторю в этой. Тесты, на которых вы будете проверять вашу программу или функцию, надо обязательно писать *до*, а не после того, как будто написана сама программа. Почему это должно быть обязательно так, здесь я объяснять не буду. Пока просто поверьте на слово.

Задача – есть коэффициенты A,B,C. Надо найти корни. Проверить на вот таких примерах, ответы вычислены заранее:

2	5	2	$X_1 = -2$ $X_2 = -0.5$
2	4	2	$X_{1,2} = -1$
2	1	2	корней нет
0	3	6	$X_{1,2} = -2$
0	0	3	корней нет
0	0	0	тождество

Приступаем к программированию. Как вы заметили, программа наша может двигаться по одному из трёх путей.

```
if A <> 0:
    x1 = 0;  x2 = 0
elif B <> 0:
    x1 = 0;  x2 = 0
else:
    x1 = 0;  x2 = 0
```

К чему эти нелепые присваивания иксам нулей? Сначала я хотел написать в этих местах разумный комментарий и этим ограничиться, однако оказалось, что синтаксис Питона этого не позволяет. Комментарий – это как бы абсолютное ничто, а внутри условного оператора обязательно должно что-то быть. Далее на вид незначительный, а по сути весьма существенный вопрос – а как наша программа предоставит внешнему миру результаты своей работы? Если бы она была функцией, то вернула бы корни уравнения как список. Но мы, напоминаю, пока ничего не знаем ни о функциях, ни о списках. Так что пока – корни будут просто выводиться на экран. Приступим.

Первый случай, $a \neq 0$. Хороший, годный случай, но радоваться рано. В этом случае мы должны сначала вычислить дискриминант $dis = \sqrt{b^2 - 4ac}$.
Если дискриминант больше нуля, то мы имеем два разных корня.
Если дискриминант равен нулю, то корень только один.
Если дискриминант меньше нуля, то корней нет.

Как сообщить пользователю, что корней два, или один, или вообще нет? Мы вернёмся к этому вопросу позже, обогащённые новыми знаниями, но пока, на этом уровне знаний, просто введём текстовую переменную, которая будет содержать ответ и которую можно будет вывести на экран. И первая версия нашей программы приобретает вот такой вид, незаконченный и не вполне полезный. Если вам стало скучно это читать, то сейчас я учу вас не языку Питон. Я учу вас программировать вообще. Всё сказанное относится к любому языку программирования.

```
result = 'вообще ничего не понятно'

if A <> 0:
    #  квадратное уравнение, дискриминант вычисляем
    x1 = 0;  x2 = 0
elif B <> 0:
    #  линейное уравнениё
    x1 = 0;  x2 = 0
else:
    #  особый случай
    x1 = 0;  x2 = 0

print 'a = ', A, 'b = ', B, 'c = ', C
print result
print 'x1 = ', x1, '  x2 = ', x2
```

А почему коэффициенты объявлены как переменные большими буквами, а выводятся на экран как маленькие? А это чтобы напомнить вам, мои маленькие друзья, что большие и маленькие буковки в Питоне это совсем разные буковки.

Снова вернёмся к нашему первому из трёх случаю, который, напоминаю, непринуждённо разветвляется на ещё три.

```
dis = B**2 - 4*A*C;

if dis > 0:
    x1 = (-B+dis**0.5)/(2*A);
    x2 = (-B-dis**0.5)/(2*A);
    result = 'два корня'
elif dis == 0:
    x1 = (-B)/(2*A); x2 = x1
    result = 'один корень'
else:  //  дискриминант меньше нуля - корней нет
    x1 = 0; x2 = 0
    result = 'корней нет'
```

А почему в случае, когда корней нет вообще, мы записываем два нулевых корня? А потому, что мы пока по-другому не умеем. Ну, или умеем, но очень коряво. Если в каком-то другом традиционном языке мы вычислим значение только одного корня, или вообще ни одного, то получим при попытке вывода вместо значений неизвестных корней, в худшем случае, какой-то мусор. Скорее всего, однако, будут выедены на экран просто нули. В Питоне не так. Если переменной не присвоено значение, а мы хотим её вывести, то программа завершится катастрофой.

И ещё, мне даже неудобно об этом упоминать, но здесь мы не производим при делении проверку на ноль – догадайтесь с одного раза, почему. Справедливости ради, мне тоже очень неудобно об этом упоминать, но с первого раза этот код не заработал – строчки я тупо размножал и подправлял, получилось вот так:

```
x1 = (-B+dis**0.5)/(2*A);
x1 = (-B-dis**0.5)/(2*A);
```

Хорошо, что я программу немедленно протестировал. Впрочем, впереди нас ждут и другие сюрпризы. Это не какие-то специальные, педагогические ошибки. Это ошибки совершенно реальные. Насколько простой ни была бы программа, ошибки в ней будут обязательно, готовьтесь.

Теперь второй случай, случай линейного уравнения. Здесь всё просто. Или кажется простым.

```
#  линейное уравнениё
x1 = -C/B;  x2 = x1
result = 'один корень'
```

А почему только *кажется*? Напомню исходный вид нашей программы, с заданными входными данными.

```
A = 0; B = 5; C = 2
result = 'вообще ничего не понятно'

if A <> 0:
    #  квадратное уравнение, дискриминант вычисляем
    #  это мы уже запрограммировали
elif B <> 0:
    #  линейное уравнениё
    x1 = -C/B;  x2 = x1
```

Чему равен корень уравнения? Очевидно $x = \frac{-C}{B} = \frac{-2}{5} = -0.4$. При запуске программы, однако, получаем нечто неожиданное:

```
a =  0 b =  5 c =  2
один корень
x1 =  -1   x2 =  -1
```

Если немного задуматься, то ничего неожиданного здесь нет – это Питон, со всем его особенным отношением к целым числам и операциям с ними. Коэффициенты A, B, C заданы как (0, 5, 2) соответственно. Это целые числа – следовательно, результат от деления тоже *обязан* быть целым. А почему всё сработало отлично в первом случае? А потому, что там была операция извлечения квадратного корня, или, говоря по-другому, операция возведения в степень 0.5. Результат этой операции может быть целым, но только чисто случайно, поэтому Питон предполагает его дробным.

Это просто особенность Питона. *Наши недостатки продолжение наших достоинств* © Кто сказал, не знаю, пусть будет Пушкин.

Переменная в Питоне может иметь любой тип и менять его много-много раз. Это удобно. Неудобно то, что вы должны тип переменной всё равно иметь в голове и отслеживать, как он повлияет на выполнение вашей программы. В нашем случае мы должны с самого начала задать исходные данные немного по другому.

```
A = 0; B = 5.0; C = 2.0
```

Результат тоже немедленно изменится и приобретёт смысл:

```
a =  0 b =  5.0 c =  2.0
один корень
x1 =  -0.4   x2 =  -0.4
```

Если вы думаете, что на этом наши проблемы закончились, то вы оптимист. Мы ещё вернёмся к этой теме в главе о функциях.

Теперь третий случай – уравнение вида $C = 0$. Вы считаете, что этого не может быть? Вы неправы – очень даже может. Обработка этого случая даже сложнее, чем предыдущего. Вариантов два – или последний коэффициент равен нулю, или нет.

```
#  особый случай
    if C == 0:
        x1 = 0;  x2 = 0
        result  = 'тождество'
    else:
        x1 = 0;  x2 = 0
        result = 'корней нет'
```

Почему присваивание нулевого значения корням повторяется два раза? Чтобы в случае модернизации программы не потерять эту операцию. А программа модернизироваться будет, я обещаю. Для начала неплохо было бы обеспечить вывод именно того количества корней, сколько их есть. Если один корень – выводить один, а не всегда два, как сейчас.

Первый, временный вариант

```
print result
if result == 'два корня':
    print x1,x2
elif result == 'один корень':
    print x1
else:
```

```
pass
```

Чем плох этот вариант? Плох он тем, что в случае изменения текста одного из сообщений вся схема рушится. Немного позже мы попытаемся это исправить, а пока пусть будет то, что есть. А вы попытайтесь из этих отдельных блоков собрать законченную и работоспособную программу.

Глава четвёртая. Циклы

Вступление и о главном. Введение в цикл for

Язык Питон – мощный и гибкий, как одноимённая змея. Язык Питон может всё. То, что мы видели раньше, – это применение самых простых его возможностей к самым простым задачам. Проблема Питона в том, что иногда он *слишком* мощный. *Слишком* – для наших скромных задач. Как математик, я не люблю аналогий – в математике рассуждения по аналогии неприемлемы. Тем не менее, один раз можно. Вот есть у вас дача. И вы захотели на этой даче воздвигнуть строение.

Строго на север порядка пятидесяти метров расположен туалет типа «сортир», отмеченный на плане буквами «Мэ» и «Жо»...
© к/ф Бриллиантовая рука

Если вам надо вырыть яму для вышеупомянутого строения, вряд ли вам поможет шагающий экскаватор, лучше нанять группу лиц какой-то национальности с лопатами. Если вам непонятно, о чём это я вообще, то сейчас мы займёмся циклами. Что такое цикл? В простейшем случае – это когда мы что-то одинаковое делаем несколько раз, причём абсолютно одинаковое, вот так, например:

```
print 'Stop!'
print 'Stop!'
print 'Stop!'
print 'Stop!'
print 'Stop!'
```

```
Stop!
Stop!
Stop!
Stop!
Stop!
```

Как-то скучно писать одну строку в программе пять раз, чтобы пять раз вывести один и тот же текст. Чтобы избавиться от этой печальной необходимости, циклы и выдуманы. Цикл, в простейшем варианте, повторяет одно и то же. Пишем цикл:

```
for i in xrange(1,5):
    print 'Stop!'
```

Всё понятно? Что, ничего вообще не понятно? Начинаем разбираться. Со второй строкой более-менее ясно – это то, что мы хотим повторять. Первая строка кончается двоеточием, а вначале второй строки сами собой появились четыре пробела – точь-в-точь, как в условном операторе. И это не случайно, логика и механизм работы те же самые.

В первой строке виднеются цифры *один* и *пять*. Если мы на пальцах посчитаем, загибая их, от одного до пяти, то получим целых *пять* загнутых пальцев. Но ведь мы ещё и не запустили программу на выполнение! Запускаем, наконец. Вот результат:

```
Stop!
Stop!
Stop!
Stop!
```

Аккуратно пересчитываем – четыре! Почему, если мы попросили цикл выполниться от одного до пяти, он выполнился только четыре раза? Получать простой ответ на сложный вопрос будем медленно – слона едят по частям. Обратите внимание – в первой строке у нас задействована переменная I, которая, вроде бы, не имеет никакого применения. На самом деле, это очень важная переменная – ещё она называется *переменная цикла*, ещё – *итератор*.

Георгий Иванович, он же Гога, он же Гоша, он же Юрий, он же Гора, он же Жора, здесь проживает? © к/ф Москва слезам не верит

Есть переменная цикла I, и, что важно, и я вас уверяю, внутри цикла она всё время изменяется. Говоря по-другому, есть понятие *шаг цикла*, это значит, что переменная цикла изменилась. То есть сначала $i = 1$, а потом $i = 2$. Это и есть шаг цикла. Все эти рассуждения ведут к одному – а не попробовать ли нам вывести переменную цикла I внутри этого самого цикла? Выводим.

```
for i in xrange(1,5):
    print 'i = ', i
```

```
i =  1
i =  2
i =  3
i =  4
```

Итак, если мы явно и чётко пишем (1,5) – циклй выполняется только от единицы до четырёх, то есть – для последнего числа он не выполняется. Это надо запомнить.

Урок русского языка в грузинской школе.
Учитель:
- Дети, слова учител, победител *пишутся с мягким знаком на конце, а слова* вилька, тарелька, *пишутся бэз мягкого знака.*
Гоги:
-Учител, но как это понять?
Учитель:
-Гоги, это понять невозможно, это надо запомнить.
© старый анекдот из тех времён, когда в грузинских школах преподавали русский язык

Это действительно надо запомнить. Авторы учебников по Питону пытаются растолковать эту загадку так: мы начинаем движение с первого числа, но останавливаемся *перед* последним. Мне такое объяснение не кажется очень понятным, но это пока не важно. На самом деле, цикл **`for`** в Питоне способен на большее, здесь мы видим только самый его простой и частный случай. Цикл **`for`** на самом деле может сделать *почти всё.*

— Что это за человек?
— О, это большой талант; из своего голоса он делает всё, что захочет.
— Ему бы следовало, сударыня, сделать из него себе штаны
© А. С. Пушкин

Нам пока не надо *всё* – нам нужно самое простое применение цикла, для многократного выполнения совершенно одинаковых или очень похожих действий. Чтобы продвинуться дальше, зададим себе вопрос – что означает загадочное, отсутствующее в американском языке слово **`xrange`**? Вместо **`xrange`** можно написать и просто **`range`**, по-английски – диапазон, так уже понятнее. То есть range(1,5) означает числа в диапазоне от 1 до 5, *но!* Последнее число в диапазон не входит, вы ведь уже запомнили это, правда?

примечание для любознательных

Если вам хочется, можно всегда писать просто ***range*** и не задумываться ни о чём. ***xrange*** делает в точности то же самое, но при этом экономит память. Если вы закажете цикл от единицы до миллиона, то, скорее всего, для его выполнения будет затребовано несколько мегабайт памяти – это если использовать просто ***range***. А если ***xrange*** – то нет.
Конечно, для наших игрушечных программ это совсем неважно.
конец Примечания

В качестве границ интервала можно использовать и переменные, а не конкретные числа.

```
start = 1
stop  = 2

for i in xrange(start+1,stop*2-1):
    print 'i = ', i
```

Быстро определите в уме, сколько раз отработает оператор печати. Цикл может двигаться и в обратную сторону, но это немного сложнее. Что будет, если мы напишем просто и без затей

```
for i in xrange(5,1):
    print 'i = ', i
```

А ничего не будет. Цикл выполнится ровно ноль раз. Причина в том, что по умолчанию, если не задано иного, переменная цикла увеличивается на единицу на каждом шаге цикла. Или, говоря иначе, шаг приращения цикла равен плюс единице. Это даёт нам некоторую свободу маневра – если можно плюс один, наверное, можно и минус один? Совершенно правильно, можно и минус один, вот так:

```
for i in xrange(5,1,-1):
    print 'i = ', i

i =  5
i =  4
i =  3
i =  2
```

В скобках мы видим нечто новое, третий параметр – шаг приращения. Он, конечно, не обязан быть равен плюс или минус единице. Пока что там может стоять любое целое число. Результат иногда может быть неочевиден.

```
for i in xrange( -10,-1,+4):
    print 'i = ', i
```

Сколько раз и для каких значений переменной цикла I будет выполнен этот цикл? Знак плюс здесь исключительно из педагогических соображений, он ничего не меняет. Правильный ответ:

```
i =  -10
i =  -6
i =  -2
```

Моё мнение – цикл должен двигаться только с шагом плюс один. В крайнем случае – минус один, но и то, по моему мнению, это лишнее и портит в остальном хорошую программу. Послушайтесь моего совета – только плюс один! И ещё о важном. Первый параметр можно не писать, вот так

```
for i in xrange(5):
    print i, '^ 2 = ', i**2

0 ^ 2 =  0
1 ^ 2 =  1
2 ^ 2 =  4
3 ^ 2 =  9
4 ^ 2 =  16
```

Вывод – если в скобках написать только одно число, то отсчёт пойдёт, начиная с нуля. Запомните. А теперь повторение пройденного и дополнительное напоминание. Как вы уже запомнили, цикл выполняется от первого параметра **xrange** до второго минус единица. То есть следующий код работает так:

```
for i in xrange(1,5):
    print i

1
2
3
4
```

Это понятно, это мы уже обсуждали. Для пяти цикл не выполняется. А если цикл в обратную сторону, от пяти до единицы?

```
for i in xrange(5,1,-1):
```

```
    print i

5
4
3
2
```

Абсолютно то же самое, но теперь цикл не выполняется и для единицы. Для осознания этого требуются некоторые умственные усилия.

Цикл for – практика и подробности

Сначала несложный пример – выведем таблицу первых трёх степеней первых десяти чисел. То есть на выходе мы должны получить что-то вроде

```
1       1      1
2       4      8
3       9     27
--------------
10    100   1000
```

Начинаем думать. Поскольку нам надо вывести степени для чисел от 1 до 10, надо будет написать ***xrange(1,11)***. Пишем так и получаем вот такое:

```
for i in xrange(1,11):
    print i,
    print i**2,
    print i**3

1 1 1
2 4 8
3 9 27
------------
10 100 1000
```

Таблицу я сократил до первых трёх строк. В принципе, всё неплохо – содержание правильное. Форма несколько неэстетичная и требует выравнивания столбцов. Реализовать это средствами оператора **print** затруднительно, поэтому отложим задачу форматирования до лучших времён.

Простая задача – сумма арифметической прогрессии. Сначала о том, что такое эта *арифметическая прогрессия*. Самый простой случай: 1,2,3,4... Или сложнее:10,13,16,19... Главное в арифметической прогрессии то, что каждый последующий член равен предыдущему плюс некоторая

неизменная величина. Как посчитать сумму первых N членов прогрессии? Очень просто. $\sum_{i=1}^{n} a_i = \frac{a_1 + a_n}{2} n$. Однако мы сделаем вид, что этой формулы не знаем. Посчитаем сумму тупо и в лоб – а потом проверим по этой самой формуле. План работ такой:

```
ввести N
посчитать сумму через цикл
вывести
посчитать сумму по формуле
вывести
сравнить
```

Затруднений вроде бы не предвидится, пока. Будем считать сумму самой простой арифметической прогрессии 1,2,3,4...

```
N = int(raw_input('N = '))

sumIter = 0
for i in xrange(1,N+1):
    sumIter = sumIter + i

print 'sumInter = ', sumIter

N = 4
10
```

Теперь посчитаем по формуле. И сравним две суммы.

```
sumForm = ((1 + N)/2)*N

print 'sumForm = ', sumForm

if sumIter == sumForm:
    print 'Ok'
else:
    print 'very, very bad!'
```

Выглядит неплохо, однако всё же проверим.

```
N = 3
sumInter =  6
sumForm =  6
Ok
```

Всё хорошо, для числа три. Упростим задачу до невозможности – посчитаем сумму для двух членов.

```
N = 2
sumInter =  3
sumForm =  2
very, very bad!
```

Что-то пошло не так. А почему? А потому, что деление на два выполняется нацело. Если в скобках один плюс два, то есть три, и мы поделили это нацело на два, то получили единицу. Ещё осталась единица в остатке, но что нам с этого толку? Получается, что наша программа неверно работает со всеми нечётными числами. Обидно. Что делать? Первая идея – ввести число два не как целое, а как плавающее: 2.0. Идея неплоха, но этому помешает функция ***int*** в операторе ввода. Несложно заменить её на ***float***, но это как-то не очень красиво. Вторая идея – поправить формулу, вот так:

```
sumForm = ((1 + N)*0.5)*N
```

```
N = 2
sumInter =  3
sumForm =  3.0
Ok
```

Вроде бы хорошо, но нельзя ли изменить формулу так, чтобы результат остался целым? Обдумайте.

Теперь об очень важном и вечном. Нам надо посчитать сумму – не по формуле, а непосредственно. Мы присваиваем сумме перед циклом ноль, а в цикле добавляем слагаемые, это так всегда – когда мы считаем сумму. А бывает по-другому? Да, бывает – когда нам нужна не сумма, а что-то другое – например, произведение. Следующий пример именно об этом, но с одним большим отличием.

Это ещё одна стандартная задача, с той только разницей, что простого пути для её решения до сих пор не придумано, только непосредственно вычислять. О чём речь? Сумму арифметической последовательности можно вычислить непосредственно. Если для вас это скучно и раздражает, можно вычислить по короткой формуле. Следующую задачу можно решить только в лоб, короткой формулы нет. Задача называется расчёт факториала. Что такое факториал? $N!=1\times2\times3\times...\times N$ – очень просто. И программа очень простая:

```
N = int(raw_input('N = '))

F = 1
for i in xrange(2,N+1):
    F = F * i

print N,'! = ', F

N = 6
6 ! =  720
```

Пробел между шестёркой и восклицательным знаком выглядит не очень красиво, исправьте сами. Вычисление факториала интересно тем, что на нём можно демонстрировать применение самых разных технологий программирования, рекурсии, например.

Ещё одна задача с циклами, но оперирующая плавающими числами. Задачу возьмём у знаменитого Перельмана, не того, которому миллион долларов не нужен, а у того, предыдущего Перельмана, который всё-таки не отец последующему. Запутались? Разберитесь!

А теперь цитата из книги Перельмана «*Живая математика*», а внутри цитата другая, вложенная цитата:

Предоставляю читателю самостоятельно решить следующую задачу, почерпнутую из «Господ Головлевых» Салтыкова:

Порфирий Владимирович сидит у себя в кабинете, исписывая цифирными выкладками листы бумаги. На этот раз его занимает вопрос: сколько было бы у него теперь денег, если бы маменька подаренные ему при рождении дедушкой на зубок сто рублей не присвоила себе, а положила в ломбард на имя малолетнего Порфирия? Выходит, однако, немного: всего восемьсот рублей.

Предполагая, что Порфирию в момент расчета было 50 лет, и сделав допущение, что он произвел вычисление правильно (допущение маловероятное, так как едва ли Головлев знал логарифмы и справлялся со сложными процентами), требуется установить, по скольку процентов платил в то время ломбард.

Пересказываю своими словами и привожу в порядок. Есть великий русский писатель Салтыков-Щедрин. У нас в городе стоит очень плохой

памятник этому великому писателю. Зато его очень любят голуби, памятник, я имею в виду. На нём всегда сидят голуби. Самый главный голубь сидит у писателя на голове. У него – у писателя – есть роман про нехороших людей – «*Господа Головлёвы*». Самый плохой из нехороших – Порфирий. Когда он родился, то получил в подарок 100 (сто) рублей, но маменька заныкала и растратила. Порфирию сейчас пятьдесят. Он напрягся и высчитал, что если бы маменька положила сто рублей в ломбард (или банк) под проценты, то за пятьдесят лет они превратились бы в восемьсот. Перельман спрашивает, сколько процентов годовых тогда платили такие учреждения. Юридическая и финансовая стороны понятны.

Теперь математика и программирование. Перельман предлагает решить задачу математически, с помощью бумаги, карандаша и логарифмических таблиц. Это потому, что у него не было калькулятора, не говоря уже о компьютере. А у нас есть. Решать будем, исходя из тех знаний Питона, которые у нас уже есть. Напишем программу, на вход которой даются три величины – сумма денег, годовой процент, количество лет. На выходе – сколько денег стало.

Напоминаю, как рассчитывается прирост денег. Пусть у нас в начале, как и в романе, 100 руб. 00 коп. Пусть в год начисляется 1%. Имеем:

в конце первого года	100.00 + 100.00×1% = 100.00 + 1.00 = 101.00
в конце второго года	101.00 + 101.00×1% = 101.00 + 1.01 = 102.01
в конце третьего года	102.01 + 102.01×1% = 102.01 + 1.02 = 103.03

Как легко заметить, за три года, кроме трёх рублей, набежали ещё и копейки. Ясно, что если начислять будут не 1%, а больше, лишних денег набежит больше. Хотя, конечно, денег лишних не бывает. Как мы этой программой будем пользоваться? Очень просто – запустим для 1%, посмотрим на результат, наверняка будет мало. Запустим для 10% – наверняка будет много. Так, постепенно, и подберём правильный ответ.

Приступим к программированию. Сначала ввод.

```
sumBefore = float(raw_input('сумма = '))
procent   = float(raw_input('годовой процент = '))
numYears  = int(raw_input('сколько лет прошло = '))
```

Всё должно быть ясно. Количество бессмысленно прожитых лет вводится как целое, потому что оно обязано быть целым – ведь оно управляет циклом. Диапазон цикла, разумеется, от 1 до `numYears+1`.

```
sumAfter = sumBefore

for i in xrange(1,numYears+1):
    sumAfter = sumAfter + (sumAfter*procent)/100.0

print 'наши денюжки = ', sumAfter
```

Любую, саму несложную программу, надо проверить. К счастью, мы уже рассчитали руками накопленные суммы для первых трёх лет, теперь можно – и нужно – сравнить их с полученными программно.

в конце первого года	`наши денюжки = 101.0`
в конце второго года	`наши денюжки = 102.01`
в конце третьего года	`наши денюжки = 103.0301`

Очень похоже на то, что мы и ожидали. А теперь несколько опытов, для попытки нащупать ответ на исходный вопрос. Исходная сумма – 100 рублей, срок – пятьдесят лет. Меняется только годовой процент.

```
 1%        164.43
10%      11739.08
 2%        269.16
 3%        438.39
 4%        710.67
 5%       1146.74
```

Заметно резкое – по экспоненте – возрастание с ростом процента. Нам ведь нужно где-то восемьсот рублей? Подберите нужный процент, мы уже близко. Однако возникает естественный вопрос – а нельзя ли как-то процедуру оптимизировать, избавившись от ручного подбора? Можно, но этим вы займётесь сами, освоив *другие циклы* чуть позже.

Разговор о переменных, особенно – о логических

Ещё в этом разделе будет довольно много рассуждений о том, как должна быть организована программа. Много, разумеется, только для того уровня знания языка, который я успел до вас донести.

Языки программирования без переменных редко, но случаются. Лично я сходу вспомнил LISP, на котором немного программировал и APL, которого не видел даже издалека. Да и то, переменных не было только в исходном, расово чистом Лиспе, а потом что-то подобное всё-таки завелось. В APL, переменные как бы были, но специфические. Так вот, во всех *традиционных* языках программирования переменные есть. Питон – язык *нетрадиционный*, но переменная в нём тоже есть. В чём подвох?

Поговорим о переменных – в Паскале или C++. Переменная в этих языках обязательно имеет *тип*. Когда мы говорили о Питоне, мы обсуждали переменные целые, плавающие и строковые. В других, обычных, языках всё то же самое. Но! – переменная получает тип с момента своего появления и никогда не меняет его. Не может целая переменная стать строковой. Более того, невозможно присваивание между переменными разных типов. Нельзя целой переменной присвоить плавающую переменную или выражение плавающего типа. Нельзя строке присвоить целое или плавающее значение. Есть одно повсеместное исключение, без которого программировать стало бы неимоверно трудно. Исключение такое – плавающей переменной можно присвоить целое значение. Но! – опять-таки это но! – плавающая переменная от этого, само собой разумеется, не станет целой. Если плавающей переменной присвоить целую единицу и затем поделить на два, то в итоге мы получим честные пол-литра.

Если иметь в виду только то, что мы уже знаем о Питоне, то картина имеет некоторое сходство. Я честно рассказал о целых, плавающих и строковых переменных. И вам показалось, что они есть. Я вас обманул – их нет! Да, то есть нет, нет в Питоне переменных каких-то там типов. Есть просто переменные, без типа. Что это значит? Вот очень простой, несложный программный код:

```
integer = 44
single  = 3.14
string  = 'Мама мыла раму'
```

Казалось бы, мы объявили три переменных – целого, плавающего и строкового типа. А что будет, если мы напишем что-то вроде этого, то есть продлим предыдущий код?

```
integer = 44
```

```
single  = 3.14
string  = 'Мама мыла раму'

integer = 'Мама, мама, что мы будем делать?'
string  = 3.14158 ** 3

print integer
print string
```

```
Мама, мама, что мы будем делать?
31.005902024
```

Что мы сделали – мы присвоили целой переменной строковое значение, а строк – плавающее. И вывели. И всем всё равно.

Запомним – в Питоне типов нет! Это мы ещё обдумаем. Потом.

Хотя переменные не делятся по типам, тем не менее, всегда и везде переменные делятся по назначению. В первом приближении есть переменные, значения которых вводят, и переменные, значения которых выводят – потому что они нам интересны и ради того, чтобы узнать эти значения, программа и написана. А какие ещё бывают переменные с точки зрения их назначения? Поглядим на примере – вычислим объём усечённой пирамиды. Вот формула из геометрии, $V = \frac{h}{3}(S_1 + \sqrt{S_1 S_2} + S_2)$, где h высота пирамиды, а S_1 и S_2 – площади верхнего и нижнего основания соответственно. Пусть у нашей пирамиды верхнее и нижние основания – это прямоугольные треугольники с катетами a_1, b_1 и a_2, b_2. То есть $S = \frac{ab}{2}$.

Запрограммируем.

```
a1 = float(raw_input('a1 = '))
b1 = float(raw_input('b1 = '))

a2 = float(raw_input('a2 = '))
b2 = float(raw_input('b2 = '))

h = float(raw_input('h = '))

S1 = (a1*b1)/2
S2 = (a2*b2)/2

V = (h/3) * (S1 + (S1*S2)**0.5 + S2)
print 'V = ', V
```

Проверьте – программа считает правильно, по формуле. Есть переменные, которые мы вводим, – a_1, b_1 и так далее. Есть переменная, которую мы выводим, – V. А ещё есть переменные S_1 и S_2. Это то, что обычно называют *промежуточные* или *временные* переменные. Называют с некоторым неодобрением. Во многих книгах по программированию повторяется разный по форме, но одинаковый по содержанию совет – *избегайте промежуточных переменных*! Мне кажется, советчики не вполне правы. В нашей небольшой программе две промежуточные переменные по два раза каждая. Написать вместо них непосредственно выражения – можно, но это замусорит текст. Прочитать и, главное, понять его, код, будет трудно и сделает его длиннее и сложнее. Так что пусть будут вспомогательные переменные.

Теперь посмотрим на применение переменных с другой стороны – со стороны переменных *логических*, они же *булевские (boolean)*. Логические переменные почти никогда не вводятся в программу, то есть – не вводятся через оператор ввода. По крайней мере, как у нас, в текстовом интерфейсе – точно никогда. Зато если вы видите какую-то форму для заполнения, а на ней, говоря на американском программистском языке, *checkbox* – по-русски *флажок* или *галочка* или *птичка* – то это оно. Или он.

Прислали Чапаеву цистерну спирта. Как уберечь, чтобы не выпили? Подумал Чапаев и написал: $C_2H_5(OH)$. *Наутро смотрит – цистерна пустая, все пьяные. Спрашивает Петьку – как дело было?*
– Смотрим, чушь какая-то написана, а в скобках – ОН. Попробовали – и точно – ОН!
© Анекдот для высокообразованных, высококультурных школьников, сдавших ЕГЭ по химии на 146 баллов.

Чаще всего такой флажок появляется рядом с вопросом – *Принимаете ли вы лицензионное соглашение?* Пока её, птичку, не поставишь, дальше не продвинешься

Американские производители ПО проанализировали, с какой скоростью русские читают тексты. Оказалось, что русские читают в тысячи раз быстрее американцев. Подсчет производился по отрезку времени от начала чтения лицензионного соглашения и до нажатия кнопки <Согласен>.
© Из Интернета, возможно, не шутка.

Однако сосредоточимся и не будем отвлекаться. Каждому флажку на экране в программе соответствует логическая переменная. То есть вы их всё-таки вводите, но не так, как обычные числовые переменные. Вывод таких переменных на экран или печать, опять-таки, никогда не производится непосредственно. Что я имею в виду? Если у нас есть целая или плавающая переменная, значение которой равно трём, то на экране мы увидим 3 или 3.0, или, в самом плохом случае, что-то вроде 3.0000079. Логические переменные никогда не выводятся в том виде, как они есть, – говоря по-другому, конечному пользователю их значения безразличны. Поясню, чтобы меня правильно поняли. Никакому нормальному и даже ненормальному пользователю неинтересны надписи на экране ***True*** или ***False***. Но вполне ясны тексты ***Всё в порядке*** или ***Я согласен*** или ***Денег нет***.

Другая принципиальная особенность логических переменных. Сколько разных значений может иметь целая переменная? Откровенно говоря, много. А плавающая – ещё больше. Здесь мы вступаем в область интересных математических и компьютерных проблем, но это для другой книги. Логические переменные отличаются тем, что допустимых значений у них в точности два. Однако пора приступить к программированию.

```
yes = True
no  = False

print yes, no

True False
```

Что мы видим? Две переменные, которым присвоили два разных логических значения, – потому что их всего два. Потом их вывели – на экран. Увидели то же самое. Других, кроме как эти два, значений, присвоить им невозможно. И зачем всё это? Немного усложним.

```
a = 5;  b = 2
plus = a > b

if plus:
    print 'a > b'

a > b
```

Логической переменной можно присвоить результат вычисления условия. Или, другими словами, то, что в условном операторе после `if` – это и есть

логическое выражение, которое можно присвоить логической переменной. Обдумайте. Теперь случай позаковыристее. Чуть раньше я просил вас закодировать условный оператор для проверки високосного года. Закодировали? Врёте, наверное. Если таки да, то вот вам возможность проверить правильность вашего результата:

```
year = int(raw_input('year = '))

leap = (year % 4 == 0) and ((year % 4 <> 0) or ((year/4) % 4 == 0))

if leap:
    print 'leap year'
else:
    print 'not leap'
```

Это к вопросу о полезности промежуточных переменных. Очень не хотелось бы каждый раз выписывать эту строку – в случае, если эта проверка понадобится в программе более одного раза. Разумеется, эту проблему, как и большинство других, можно решить более чем одним способом. К этому мы ещё вернёмся.

Циклы. Особенности

О мелких, но важных вещах. Условные операторы могут быть вложенными, это вы знаете. Точно так же вложенными могут быть и циклы. Разумеется, мелким это явление можно назвать только потому, что об этом можно было бы догадаться и так. Вложенные циклы – очень и очень важная вещь в любой программе. Далее пример, бесполезный практически, но вполне себе актуальный математически. Выведем список простых чисел, не превышающих некоторого, заранее заданного значения. Простое число, напоминаю, – такое целое число, которое не делится без остатка ни на какое другое число, кроме единицы и самого себя. Оговорку *без остатка* пришлось добавить только потому, что у нас в программировании есть и деление с остатком, математикам это вообще непонятно. Кроме того, опять же – спасибо математикам – единица не считается простым числом. Если единицу признать за целое число, рушится вся теория чисел. На всякий случай – если число не простое, то оно *составное*.

Общая структура программы видится где-то так:

```
ввести предел числа , до которого ищем
```

```
цикл по числу до предела
   цикл по делителю до предела
      если число делится на делитель без остатка, значит оно не простое
   если число простое, то вывести
```

Переводить это псевдокод в код реальный можно почти построчно. Почти – потому что потребуется инициализация логической переменной, отвечающей за самое главное – за признак простоты.

```
howMany = int(raw_input('howMany = '))

for i in xrange(2,howMany+1):
    prime = True
    for k in xrange(2,i):
        if i % k == 0:
            prime = False
    if prime:
        print(i)
```

Программа работает, я проверял, с вот таким результатом:

```
howMany = 10
2
3
5
7
```

Что не совсем хорошо в этой программе? Я никогда не был сторонником оптимизации программного кода. Я всегда был противником улучшения и особенно ускорения. Но одно дело, когда программа стала работать в десять или даже в сто раз. Оно не сто́ит усилий, обычно. Тут ситуация несколько иная. Когда мы ищем простые числа в пределах первого десятка, вопросов нет. Но что будет, когда в пределах первого миллиона? Ответ – внутренний цикл выполнится в среднем 500000 раз. Необходимо ли это? Ответ – совсем нет, имеет смысл для числа N проверять только до $\sqrt{N}$, кстати, почему? Так что не надо выполнять цикл больше чем ≈707 раз. Наша программа настолько неэффективна, что это, пожалуй, слишком даже для меня. Поправим всего одну строку:

```
for k in xrange(2,int(i**0.5)+1):
```

Обратите внимание на +1. Обдумайте, почему отсутствие этого абсолютно безразлично в исходном коде и жизненно необходимо в улучшенном.

очевидное замечание
Впрочем, очевидное оно на то и очевидное, что очевидно не всегда и не для всех. Переменная I определена только внутри внешнего цикла, обращаться к ней снаружи этого цикла нельзя. Переменная K определена только во внутреннем цикле. Поскольку внутренний цикл по чистой случайности находится внутри внешнего, в нём – внутреннем цикле, можно обращаться и к I и к K.
конец Очевидного замечания

Пока что ни с чем новым мы не познакомились, только вложенные циклы, но до того, что один цикл может находиться внутри другого, догадаться можно было и без подсказок. А теперь выучим новое слово, даже два – полезное и бесполезное. Начнём с полезного, а заодно опять ускорим нашу программу, и опять заметно. Сколько раз выполняется внутренний цикл? Правильно, $\sqrt{N}$. Необходимо ли это? Конечно нет, ведь как только мы обнаружили делимость нацело, то есть первый делитель, то проверять дальше не надо, число составное. Меняем внутренний цикл с помощью нашего нового слова:

```
    for k in xrange(2,int(i**0.5)+1):
        if i % k == 0:
            prime = False
            break
```

Что делает оператор **break**? Этот оператор немедленно завершает выполнение цикла. Но, важная деталь, завершает выполнение только в*нутреннего* цикла. Ещё точнее – он завершает выполнение именно того цикла, в котором находится. Если бы внутри внешнего цикла было бы несколько внутренних, и не внутри один другого, а на одном уровне, то завершился бы только один из них. Смоделируйте и проверьте. А что будет, если написать так?

```
for i in xrange(2,howMany+1):
    prime = True
    for k in xrange(2,int(i**0.5)+1):
        if i % k == 0:
            prime = False
            break
    if prime:
        print(i)
        break
```

Правильно, будет выведено первое простое число и на этом всё закончится. Есть ли в этом смысл? Никакого, мы и так знаем, что первое простое число – 2. А вот если бы мы вводили не диапазон, в пределах которого ищем простые числа, а число, *начиная с которого* искать наименьшее простое число, то смысл был бы и вполне математически корректный. Измените программу для этой цели. Когда программа заработает, добавьте снаружи условный оператор, чтобы программа по выбору работала в любом из двух режимов – *до* числа и *от* числа. Не жалуйтесь! *Тяжела и неказиста жизнь простого программиста* © Народное.

Теперь представляю обещанное второе, аналогично бесполезное слово. Для тренировки аналогично бесполезная программа:

```
for i in xrange(1, 5+1):
    print 'i = ', i, i**0.5
```

```
i =  1 1.0
i =  2 1.41421356237
i =  3 1.73205080757
i =  4 2.0
i =  5 2.2360679775
```

Чтобы не быть совсем уж бесполезной, она выводит квадратные корни из первых пяти натуральных чисел. Хитрая запись 5+1 упрощает понимание. Теперь добавляем волшебное слово:

```
for i in xrange(1, 5+1):
    if i % 2 == 0:
        continue
    print 'i = ', i, i**0.5
```

```
i =  1 1.0
i =  3 1.73205080757
i =  5 2.2360679775
```

Что делает оператор, имени которого я даже не хочу вспоминать? Он игнорирует всё, что находится внутри цикла после него, и переходит к выполнению цикла с самого начала, но уже с увеличенной на единицу переменной цикла. Я считаю, что этот оператор бесполезный и даже вредный. Немедленно забудьте.

Довольно-таки сложная задача

На всякий случай, предупреждаю – это не я придумал. Изобразить таблицу умножения, вот в таком виде:

```
     1  2  3  4  5  6  7  8  9

1    1  2  3  4  5  6  7  8  9
2    2  4  6  8 10 12 14 16 18
3    3  6  9 12 15 18 21 24 27
4    4  8 12 16 20 24 28 32 36
5    5 10 15 20 25 30 35 40 45
6    6 12 18 24 30 36 42 48 54
7    7 14 21 28 35 42 49 54 63
8    8 16 24 32 40 48 56 64 72
9    9 18 27 36 45 54 63 72 81
```

Говорят, это таблица Пифагора и придумал её Пифагор – почему, не знаю. Врут, однозначно. Развивает базовые навыки вывода текстовой информации. Начинаем с центрального блока. Понятно, что где-то должен быть вложенный цикл. Нам нужны взаимные произведения чисел от 1 до 10. Поэтому пишем:

```
for i in xrange(1,11):
    s = str(i).rjust(3) + '  '
    for k in xrange(1,11):
        s = s + str(i*k).rjust(3) + ' '
    print s
```

Обратите внимание, мы сначала клеим всё, что позже на экране будет в одной (экранной) строке, в одну (символьную) строку, а затем эту символьную строку выводим на экран. Второе – это функция `rjust`. Или метод `rjust`, как вам больше нравится. `str(i*k)` превращает произведение `I*K` в строку, а `rjust(3)` добавляет к этой строке необходимое количество пробелов слева, чтобы общая длина строки составила три символа. Теперь у нас есть центральная часть таблицы. Добавляем строку сверху:

```
s =  ''
for i in xrange(1,11):
    s = s + ' ' +str(i).rjust(3)
print s
```

`print`

`print` без параметров вставляет пустую строку. Добавляем строку слева. Прежде этого придётся слегка модифицировать основной цикл:

```
for i in xrange(1,11):
    s = str(i).rjust(3) + '  '
    for k in xrange(1,11):
        s = s + str(i*k).rjust(3) + ' '
    print s
```

Теперь осталось только заменить самую первую строку, которая `s = ''`. Добавьте нужное количество пробелов, чтобы всё стало красиво.

Другие циклы

В Питоне есть ещё один, другой цикл. Сначала напишем его так, как написать его можно, но смысла никакого в этом нет – зато всё понятно.

```
i = 1
while i < 5:
    print i
    i = i + 1

1
2
3
4
```

Совершенно очевидно, что предыдущий цикл эквивалентен последующему:

```
for i in xrange(1,5):
    print i
```

Трудно не согласиться, что второй вариант короче и, главнее, понятнее. Тогда зачем нам нужен первый? В случае применения цикла `for` мы всегда знаем, сколько раз выполнится цикл – если не использовать `break`, разумеется. Если нам потребовался цикл `while` – мы не знаем заранее, сколько раз он будет выполняться. Ещё одно усложнение – переменной цикла нет. Или её нет в явном виде и переменную эту пришлось бы вводить искусственно, как в примере выше.

Самая частая ситуация применения цикла **while** – активное и непредсказуемое участие пользователя в работе программы. Пример – любая игрушка. Мы ведь не знаем заранее, когда игрок нажмёт на клавишу или когда всё игроку надоест, и он захочет выйти из игры. Но до этой категории задач нам ещё далеко, выберем что-нибудь попроще. Что-нибудь *попроще* будет из области высшей математики, конкретно – математического анализа. Там таких задач сколько угодно.

краткий курс Теории рядов

Все знают формулу про длину окружности и площадь круга: $P = 2\pi R; S = \pi R^2$. Константа $\pi \approx 3.14158\ldots$ А как они догадались, что она равна именно этому? А они почти вычислили сумму бесконечного ряда. Почти – потому что ряд бесконечный. Вот пример бесконечного ряда: $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots$ Складывать эти числа не надо, потому что математики доказали, что его сумма равна в точности единице.

А вот гораздо более полезный ряд: $\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots$ Всё те же математики доказали, что сумма этого ряда равна $\frac{\pi}{4}$. Теперь, чтобы узнать, чему же равно π, остался сущий пустяк – сложить несколько членов ряда. Ну там, десять, сто, тысячу. Скорее, миллион, если мы хотим получить хоть немного точный результат. Кстати, это называется формула Лейбница.

Для наших целей мы возьмём ряд, изобретение которого приписывается много кому. А если говорят *много кому* – значит, никому вообще. Вот он: $\pi = \sqrt{12(1 - \frac{1}{3 \times 3} + \frac{1}{5 \times 3^2} - \frac{1}{7 \times 3^3} + \ldots)}$. Говорят, этот ряд гораздо лучше – вот и проверим.

Очень важное понятие – формула общего члена ряда. Это уже очень близко к программированию. Для первого ряда это $a_n = \frac{1}{n}$, где n – номер члена в последовательности. Для второго ряда $a_n = \frac{1}{(2n-1) \times (-1)^{n-1}}$. А для третьего?

А куда вы летите, я пока не знаю © Брат-2

конец краткого курса.

Теперь надо решить две очевидно выделяемые подзадачи. Первая подзадача – как записать формулу общего члена. Задача, повторюсь, по форме математическая, а по существу – чисто программистская. Вторая подзадача – по какому признаку завершить цикл. Несмотря на слово *цикл* в вопросе, эта задача математическая и довольно сложная. Для того чтобы решить, на каком шаге остановиться при суммировании ряда, математики вывели формулы, узнать значения которых часто сложнее, чем сосчитать сам ряд. Из всех возможностей мы выберем самую математически ненадёжную. Зато она легко программируется. Наше условие записывается так: $a_n < \varepsilon$. Другими словами, очередной, последний член ряда меньше какой-то заранее заданной величины. Её, эту величину, тоже придётся ввести.

Общий член, мне кажется, выглядит вот так: $a_n = \frac{(-1)^n}{(2n-1)\times 3^{n-1}}$. Как я до этого додумался? Нет, я не подглядывал и не списывал. Нет, не потому, что я слишком умный, это потому, что я очень давно программирую. Обычная работа для программиста – посмотреть на последовательность, выявить закономерность и вывести формулу. Причём это не только в математических задачах. Теперь о проверке условия. Оно должно выглядеть где-то так `while An < eps:` Переменная называется `eps` потому, что та смешная закорючка в формуле по-гречески называется эпсилон. Маленькая проблема. `An` мы ещё не вычислили, оно будет вычислено только внутри цикла. В нашем случае мы с первого взгляда знаем, что первый член ряда равен единице, но так бывает не всегда. Часто приходится присваивать проверяемой переменной какое-то искусственное значение, к примеру, заведомо очень большое. Продолжаем программировать, однако.

Ошибки нужны только для того, чтобы на них учиться. Я написал вот такой текст и получил бесконечно длинный список чисел, бесконечно долго бегущий по экрану.

```
eps = float(raw_input('eps = '))

An = 1
S = 0
n = 1

while (An > eps):
    An = ((-1)**(n+1)) / ((2*n-1)*3**(n-1))
```

```
    S = S + An
    print S
```

Что делать в таком случае? Нет, понятно, что надо немедленно найти ошибку. Но ещё раньше надо остановить бесконечно долгое выполнение программы. А для этого надо в том окне, которое *Оболочка*, не *Редактор*, выбрать пункт меню <Shell>\<Interrupt Execution Ctrl+C> – и всё немедленно остановится. Можно проще – нажать совместно клавиши <Ctrl> и <C>. Так в чём же ошибка? Ошибок много, и не все из них глупые.

Первая. Наш ряд *знакопеременный*. Звучит красиво и загадочно, означает всего лишь, что его члены оказываются то положительными, то отрицательными. При первом же отрицательном члене цикл `while (An > eps):` закономерно заканчивает работу. Заменяем на `while (abs(An) > eps):`

Вторая ошибка гораздо более умная и интересная, если так можно говорить об ошибках. И ошибка эта уже чисто питоновская. Тут придётся немного подумать. Посмотрите на самую длинную строку, ту, где вычисления. Там делится что-то на что-то. То, что слева от знака деления, зависит только от целой переменной N и целых чисел. То, что справа, – точно так же. Если целое число поделить на другое число – получим опять целое число, потому что деление будет нацело с остатком. Это надо запомнить. Слева от оператора присваивания у нас плавающее число `An`. Если ему присвоить целое число, `An` немедленно станет тоже целым числом. Это надо понять. Это чисто особенность Питона. Как исправить? Можно вместо целых констант написать плавающие, то есть заменить: 1 ➔ 1.0. А можно явно указать, что мы хотим иметь плавающее число, в знаменателе, например: `An = ((-1)**(n+1)) / float(((2*n-1)*3**(n-1)))`. Скобок становится всё больше и больше, но тут уж ничего не поделаешь, всё, как в анекдоте – *<обидное слово>, не жалейте заварку*!

Третья ошибка до того простая, что перестаёт быть ошибкой и превращается в особенность. Я забыл умножить результат на $\sqrt{12}$. Исправляем всё разом и дорабатываем вывод результата и промежуточных переменных – для контроля процесса.

```
eps = float(raw_input('eps = '))
print

An = 1.0
S = 0.0
```

```
n = 1

while (abs(An) > eps):
    An = ((-1)**(n+1)) / float(((2*n-1)*(3**(n-1))))
    print 'An = ', An, 'eps = ', eps, 'abs(An) > eps = ', abs(An) > eps
    S = S + An
    n = n +1
    pi = S * (12**0.5)
    print 'pi = ', pi
```

А вот результаты с точностью до одной тысячной:

eps = 0.001

```
An =  1.0 eps =  0.001 abs(An) > eps =  True
pi =  3.46410161514
An =  -0.111111111111 eps =  0.001 abs(An) > eps =  True
pi =  3.07920143568
An =  0.0222222222222 eps =  0.001 abs(An) > eps =  True
pi =  3.15618147157
An =  -0.00529100529101 eps =  0.001 abs(An) > eps =  True
pi =  3.1378528916
An =  0.00137174211248 eps =  0.001 abs(An) > eps =  True
pi =  3.14260474566
An =  -0.000374111485223 eps =  0.001 abs(An) > eps =  False
pi =  3.14130878546
```

Понадобилось всего шесть итераций. Для сравнения, запрограммируйте ряд Лейбница и посмотрите, *насколько* плохо он сходится.

И ещё, вспомните задачу о процентах, ту, которая на основе Салтыкова-Щедрина. Обдумайте, как можно применить цикл **while** к нахождению её возможно точного решения.

Глава пятая. Списки

Списки – что это такое

Сначала я хотел начать с объяснения того, что такое список вообще. Показать списки в их всемогуществе, если можно так выразиться. Потом я решил, что это пугает неподготовленных, слабых духом программистов. Списки настолько всесильны, что сначала даже и непонятно, к чему их можно приспособить. Так что начнём с малого. Напишем очень короткую программу:

```
a = [11,12,13,14,15]
print a

[11, 12, 13, 14, 15]
```

Что мы видим? Мы вывели одну переменную по имени A. Вывелось пять чисел. Другими словами – имя у переменной одно, а значений у переменной пять. Всё вместе называется *список*. Список записывается в квадратных скобках. То, что внутри, – *элементы списка*. Элементы списка разделяются запятыми. У нашего списка все элементы имеют один и тот же тип. Это не обязательно, но пока будет именно так.

Как поступить, если нас интересует не весь список целиком, а конкретно какой-то один элемент? Вот мы выводим один элемент, с номером один:

```
print 'a[1] = ', a[1]
a[1] =  12
```

Сюрприз! Просили первый элемент, а получили не 11, а 12 – то есть физически второй. Причина проста и это надо запомнить навечно – элементы в списке нумеруются, начиная с нуля. Для нашего случая вот так:

```
a[0] =  11
a[1] =  12
a[2] =  13
a[3] =  14
a[4] =  15
```

Ещё раз – элементы в нашем списке, в котором пять элементов, нумеруются от нуля до четырёх. Возможно, вы это прекрасно усвоили, но приходится повторять снова и снова – настолько это важно. Что будет,

если мы попросим вывести элемент вне этого диапазона? Например, одиннадцатый. Ничего хорошего, проверьте и убедитесь. Ещё – то, что после имени списка в квадратных скобках, называется *индекс* списка. А теперь совсем несложный код:

```
n = 1
print a[2*n+1]

14
```

А зачем я это вообще написал? А чтобы показать, что индексом списка может быть не только целое число, но и переменная, и даже выражение. Главное, чтобы всё это было целого типа. Само собой разумеется, что вот так `val3 = a[3]` можно присвоить переменной элемент списка. Занудно напоминаю, что совершенно неважна история этой переменной – она могла вот только что сейчас появиться в коде, иметь раньше целый тип или быть строкой, а хотя бы даже и списком.

Теперь вопрос интереснее и важнее. Как изменить значение списка? Это я специально спросил так нечётко и расплывчато, а на самом деле, вопрос распадается на два. Первый – как изменить значение всего списка сразу и целиком. Это просто:

```
a = [11,12,13,14,15]
print a

a = [-3,87,54566545]
print a

[11, 12, 13, 14, 15]
[-3, 87, 54566545]
```

Было в списке пять элементов, мы заменили весь список на новый, стало три. Самое важное здесь слово – *заменили*. И *весь*. Мы не меняли отдельные элементы нашего списка. Мы не удаляли из него элементы. Мы не добавляли в него элементы. Мы заменили его весь. Очень может быть, что пока это для вас разговоры ни о чём. Ничего, со временем поймёте.

страшная правда

А что случилось с первым списком, после того, как ему присвоили новые значения? Куда он делся?

Он умер. Его значения всё ещё лежат в черноте оперативной памяти, но на них не указывает ни одно имя. И совсем скоро-скоро к нему придёт

страшный ~~Фредди Крюгер~~ Сборщик Мусора и сметёт жалобно скулящие байтики в совочек и утопит нафиг. Честное слово, вот так оно всё в чёрном-чёрном компьютере и работает.
конец Страшной правды

Теперь поменяем элементы выборочно.

```
a = [11,12,13,14,15]
print a

a[0] = 101
a[1] = a[2] + a[3] + 1
print a
```

```
[11, 12, 13, 14, 15]
[101, 28, 13, 14, 15]
```

Тоже не сказать, чтобы очень сложно. А что, если мы захотим поменять количество элементов в списке? Добавить – вместо пяти шесть. Или убавить – вместо пяти четыре. Об этом в следующем разделе.

Списки – короче, длиннее

Чтобы добавлять в список или удалять из списка элементы, применяются *методы*. Что такое метод с высоты орлиного полёта? Снаружи вызов метода выглядит так: <имя объекта>.<имя метода(параметры)> . Что такое объект? В языке Питон почти всё. Для нашего случая, когда мы хотим добавить в список один элемент, и добавить его в конец списка, вызов метода реализуется так:

```
a = [11,12,13,14,15]
print a

a.append(-1)
print a
```

```
[11, 12, 13, 14, 15]
[11, 12, 13, 14, 15, -1]
```

Разбираем подробно. `a` – объект, `append` – метод. После имени объекта и перед именем метода обязательно ставится точка. Метод `append` добавляет элемент в конец списка. У метода есть параметры, в нашем случае один параметр. Это то, что в круглых скобках. В круглых скобках элемент,

который мы добавляем в список. Самый первый возникающий вопрос – а если нам надо добавить не в конец, а в середину? Легко.

```
a = [1,2,3]
print a

a.insert( 2, -1)
print a

[1, 2, 3]
[1, 2, -1, 3]
```

Минус единица – то, что мы вставляем. Она потому и минус единица, чтобы её появление легко было заметить. Теперь сосредоточьтесь. Первый параметр – два – это номер элемента, *перед* которым мы вставляем нашу отрицательную единицу. Нумерация у них от нуля, значит элемент №2 это физически элемент номер три. Вот туда мы и попали. Всё понятно? Обдумайте.

Добавили? Теперь будем удалять и здесь нас ожидает опять-таки сюрприз – всё пойдёт не так, а именно:

```
a = [1,2,3]
print a

del a[1]
print a

[1, 2, 3]
[1, 3]
```

По существу вопросов нет. Мы попросили удалить элемент с индексом один, то есть второй по счёту, его и удалили. Всё хорошо. Только удалили мы его как-то по-другому, не с помощью метода, а с помощью функции. Методом тоже можно, но не ясно, нужно ли нам это. Для наглядности слегка изменим исходный список.

```
a = [111,222,333]
print a

a.remove(222)
print a

[111, 222, 333]
[111, 333]
```

Теперь, когда мы попросили удалить элемент и указали в качестве параметра 222, это число обозначает не номер элемента в списке, а его значение. Вам это надо? Иногда да, это бывает надо, но чаще – нет.
Ещё две незамысловатые, но полезные операции. Проще не объяснять, а показать.

```
a = [1,23,11,77]
print a

a.reverse()
print a
```

```
[1, 23, 11, 77]
[77, 11, 23, 1]
```

Метод записывает список в обратном порядке. Иногда это полезно. Следующая манипуляция ещё полезнее:

```
a = [1,23,11,77]
print a

a.sort()
print a
```

```
[1, 23, 11, 77]
[1, 11, 23, 77]
```

Как легко заметить, наш список оказался отсортированным по возрастанию. А если надо отсортировать по убыванию? Можно отсортировать точно так же, по возрастанию, а потом применить метод `reverse()`. Но можно и просто, без затей.

```
a = [1,23,11,77]
print a

a.sort(reverse=True)
print a
```

```
[1, 23, 11, 77]
[77, 23, 11, 1]
```

Мы уже удаляли какой-то конкретный элемент списка по его значению. Нетрудно догадаться – если мы можем найти и удалить элемент по его значению, то мы ведь можем найти, но не удалять элемент. То есть указываем значении элемента и получаем его индекс.

```
a = [1,23,11,77,11]
print 'index = ', a.index(11)

index =  2
```

Элемент 11 входит в список два раза и найдено только *первое* его вхождение. Обратимся к другому методу:

```
a = [1,23,11,77,11]
print 'count = ', a.count(11)

count =  2
```

Обратите внимание. Ответ формально тот же – 2, но смысл его совсем другой. Теперь это количество вхождений числа 11 в список.

А теперь очень неброская, но очень важная функция:

```
a = [1,23,11,77,11]
print 'len = ', len(a)

len =  5
```

Функция считает длину списка – сколько в нём элементов. Это очень просто и очень полезно, и очень часто будет применяться дальше. Говоря ещё более простыми словами, теперь наш цикл работает со списком любой длины, а не только содержащим пять элементов.

А какая вообще может быть польза от списков? Списки полезны и сами по себе, но если нужна настоящая, весомая польза – соедините списки с циклами.

Списки плюс циклы. Начало. И философия

Об использовании списков вместе с циклами можно сказать столько, что придётся говорить медленно и по частям. Список содержит много элементов, обращаться к ним по одному скучно и утомительно. Цикл предоставляет к нашим услугам переменную цикла, легко пробегающую все индексы списка. Сначала просто выведем в цикле все элементы списка:

```
a = [1,23,11,77]
```

```
for i in xrange(0,4):
    print i,'.   ', a[i]

0 .    1
1 .    23
2 .    11
3 .    77
```

Всё очевидно. Четыре – это количество элементов в списке. А что делать, если количество элементов в списке другое, или изменится, или вообще нам заранее неизвестно? Всё правильно, это мы уже умеем. Делать надо вот так:

```
a = [1,23,11,77]

for i in xrange(0,len(a)):
    print i,'.   ', a[i]

0 .    1
1 .    23
2 .    11
3 .    77
```

Здесь мы должны тщательно обдумать то, что видим. Нет, в целом всё понятно. Лично мне сначала казалось как-то непонятно – ведь цикл останавливается перед вторым параметром `xrange`, цикл для него не выполняется. Тем не менее, мы просто пишем `len(a)`, как собственно и подсказывает нам здравый смысл – и всё работает идеально. Потом я вспомнил, что начинаем цикл мы с нуля, так что одна странность компенсирует другую. Извините, что запутанно и расплывчато – это я показываю, как непросто давалось мне постижение Питона.

Теперь немного философии, можно пропустить, но лучше прочитать, особенно, если вы хотите после прочтения этой книги не забыть немедленно её содержание, а ещё и немного программировать, не обязательно на Питоне. Есть такое понятие – жизненный цикл. Ну типа – лягушка откладывает икру, из икры образуются головастики, из головастиков получаются лягушки и всё по новой. Я ничего не перепутал? Или бабочка откладывает – что там откладывают бабочки? – дальше гусеница – кокон – опять бабочка. А теперь о жизненном цикле переменной.

Откуда берётся сама переменная, мы *пока* говорить не будем. Если в конце книги останется несколько неиспользованных страниц, я об этом

обязательно напишу. Но когда переменная уже существует, для неё откуда-то берётся значение. Потом это значение обрабатывается, возможно, результат получает та же самая переменная, возможно, исходной переменной наследует другая. Например, ввели радиус и присвоили одной переменной, сосчитали длину окружности и присвоили переменной номер два. А потом переменную куда-то отправили – на экран, на печать, в файл или, как теперь модно говорить – *в облако*. В любом случае, куда-то отправили – если результат никому не нужен, то кому вообще нужна эта программа?

Для простой переменной – целой или плавающей – всё просто. Мы вводим значение, ведь мы уже научились вводить? Мы что-то там считаем, даже с применением условных операторов и циклов. Мы выводим результат. И всё у нас получается.

Теперь списки. Начнём с конца. Вывести список мы уже умеем – смотри выше. Обрабатывать, честно говоря, мы ещё не очень умеем, хотя уже можем со списком что-то сделать – добавить элемент, удалить элемент, поменять порядок элементов. А теперь о главном – откуда, собственно, берётся список? Вопросов нет, если это список игрушечный, из трёх элементов, программа учебно-тренировочная и это список прямо в программе программным образом и задают. Если в списке десять элементов, то их можно попросить ввести пользователя. Как частный случай, рассматривается ситуация, когда пользователь – сам разработчик программы, он и будет вводить и мучаться. Это справедливо. Попробуйте ввести хотя бы десять дробных чисел – четыре знака до запятой, два после – и ни разу не ошибиться. Наказание за ошибку – начать ввод с самого начала.

Однако в реальных списках – сотни, тысячи и десятки тысяч элементов, а иначе зачем списки вообще нужны? И откуда же берётся всё это богатство? Берётся оно, как правило, из файлов. Файл здесь имеется в виду в очень широком смысле слова – это может быть и физический файл, о них ещё будет специальная глава в книге. Данные может предоставлять база данных, которая, безусловно, физически тоже файл и далеко не один. Данные могут приходить откуда-то по сети – из той же базы данных в этой же комнате, или через интернет за тысячи километров. Не обязательно именно через интернет, разумеется. Во всех этих случаях будущие элементы списка заведомо существуют до начала выполнения программы.

Так бывает не всегда. Иногда назначение программы заключается в том, чтобы принимать данные, приходящие из внешнего мира. Есть датчик, он десять раз в секунду снимает какие-то отсчёты – температуру воздуха и частоту бета-распада. Всё это пишется в файл, разумеется, а попутно – в список, и с этим списком параллельно тоже что-то происходит. Его элементы сортируются, пересчитываются, вводятся и выводятся.

Однако уже слишком много слов и ни одной строки программы. Напишем две программы для двух случаев возникновения списков. Первый – отчасти практический, когда список вводится руками, с клавиатуры. Второй – скорее, теоретический, когда список генерируется каким-то математическим способом, из формулы.

Но сначала просто создадим список с нуля, ведь пустого списка у нас ещё не было.

```
a = []
a.append(10)
a.append(20)

b = []
for i in xrange(0,11):
    b.append(i**2)

print a
print b
```

```
[10, 20]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Пустой список очень прост и очевиден: `[]`. Две квадратные скобки, внутри которых пустота. Первый пример – добавление элементов в список поштучно. Второй – в цикле. В первом случае добавить ещё сотню элементов вызовет некоторую усталость. Во втором – пустяк, главное знать формулу, по которой эти элементы вычислять.

Теперь спросим у клиента, сколько чисел он хочет ввести, введём и внесём в список, даже не обрабатывая. Список выведем – а иначе зачем все эти слепые телодвижения?

```
howMany = int(raw_input('How Many? = '))
a = []
```

```
for i in xrange( 0, howMany):
    what = int(raw_input(' what = '))
    a.append(what)

print a

How Many? = 3
 what = 11
 what = 12
 what = 13
[11, 12, 13]
```

Задайте себе вопрос – а что, если где-то наш клиент вместо целого числа введёт дробное, что будет? Ничего хорошего не будет, разумеется. А как с этим бороться? Пока никак, но мы к этому непременно вернёмся.

А теперь задача математическая. Заполним список случайными числами. Само по себе это редко нужно – то есть зачем нам десять тысяч случайных чисел? Но это часто бывает нужно, как отправная точка исполнения нашей программы. Говоря менее торжественно, ведь с чего-то надо начинать? Мы написали программу. Программа эта обрабатывает список. Программу обязательно надо проверить. Список надо где-то взять. Выдумать из головы? Трудно, голова, она одна. Сгенерируем список случайным образом.

```
import random

num = 5
a = []

for i in xrange(0,num):
    what = random.randint(1,20)
    a.append(what)

print a

[19, 16, 4, 12, 5]
```

Просто запомним это и повторять не будем. Если я дальше попрошу – *заполнить список случайными числами* – то вот это оно и есть.

Списки плюс циклы.
Стандартные ситуации

Этот раздел будет скучным. Или нет? В любом случае, этот раздел не даст никакой возможности для творчества. Вам не придётся думать, размышлять, фантазировать. Вам надо только запоминать. Есть типовые задачи – у них есть типовые решения. Не надо ничего выдумывать – просто сделайте так, как все и всё уже делали до вас! Если вам кажется, что я излишне эмоционален, извините. Я встречал очень большое количество программистов, которые категорически не хотели делать, как надо. Они изобретали свой собственный велосипед, подводный, с зонтиком и на вёслах. Так вот, те задачи, которые я предлагаю дальше, решаются именно так и только так, и, я таки вас умоляю, не надо ничего выдумывать.

Для краткости. Если список уже чем-то заполнен, то он заполнен случайными числами и количество этих чисел ровно `num`. Если вы просто скопируете код заполнения списка откуда-то сверху, то для вас `num = 5`.

1. Заполнить список одним и тем же значением, чаще всего это значение – ноль :

```
a = []
for i in xrange(0,num):
    a.append(0)

print a
```

Вопросы есть? Вопросов нет. Заполнить ноль любым другим числом вы, несомненно, сможете.

2. Заполнить список случайными разными значениями:

Почему эта задача снова здесь, ведь мы её уже решали? Во-первых, так положено, исходя из педагогических соображений, – возвращаться к решению уже решённой задачи другими способами. Во-вторых, задача на самом деле другая – раньше от нас не требовалось, чтобы значения были разными. В-третьих, мы легко и просто решим задачу, которую решить вообще-то довольно трудно. А почему у нас будет просто – потому что в стандартном модуле Питона `random` всё уже решили до нас. У нас есть список, пусть из десяти элементов. Нам надо выбрать из него несколько, пусть три случайных и *разных* элемента. Задача не то чтобы

невыполнимая, но требует некоторых усилий. Попробуйте. Проблема в том, что нельзя просто выбирать элементы со случайным номером из первого списка – они, совершенно случайно, могут и повторяться. А в Питоне всё уже реализовано до нас:

```
import random
a = [1,2,3,4,5,6,7,8,9,10]
b = random.sample( a, 3)
print b

[6, 10, 1]
```

3. Поместить в каждый элемент списка его индекс. Предполагается, что список изначально не существует:

```
a=[]
for i in xrange(0,num):
    a.append(i)
print a
```

Ничего нового, всё очевидно.

4. Увеличить каждый элемент списка на единицу:

```
for i in xrange(0,num):
    a[i] = a[i] + 1

print a
```

5. Вычислить сумму элементов списка:

```
sum = 0
for i in xrange(0,num):
    sum=sum + a[i]
print sum
```

Ничего нового, пройденный материал.

6. Определить среднее элементов массива – результат плавающий

```
sum=0
for i in xrange(0,num):
    sum=sum+a[i]
print sum

av=float(sum) / num
print av
```

7. Найти минимальный элемент массива и его заодно его индекс:

```
min=a[0]
index=0

for i in xrange(1,10):
    if a[i] < min:
        min=a[i]
        index=i

print min, index
```

8. Сосчитать количество отрицательных и неотрицательных элементов массива:

```
# это комментарий, если уже забыли
# тут было заполнение списка случайными значениями
#    и инициализация num = 5
numNeg=0
numNotNeg=0

for i in xrange(0,num):
    if a[i] < 0:
        numNeg=numNeg + 1
    else:
        numNotNeg=numNotNeg + 1

print numNeg
print numNotNeg
```

9. Сдвинуть все элементы, начиная с четвертого, налево на три. Хвост заполнить нулями.

```
for i in xrange(0,len(a)-3):
    a[i]=a[i+3]
for i in xrange( len(a)-3, len(a)):
    a[i]=0

print a
```

Для полной универсальности кода применена функция `len()`.

10. Определить индекс первого чётного элемента массива:

```
indexEven=0

for i in xrange(0,num):
    if a[i] % 2 == 0:
```

```
        indexEven=i
        break

print indexEven
```

Ещё раз напоминаю, что остаток от деления обозначается процентом, а проверка на равенство – двумя равенствами. Ничего личного, я сам долго забывал. Надо похвалить Питон – некоторые задачи из тех, что мы рассмотрели, в нём решаются заметно проще, чем в других языках.

Стандартные ситуации. Чуть сложнее

Немного более сложные случаи. Сложнее они тем, что в ситуацию вовлечены не один, а больше списков.

1. А начнём с самого простого, что только может быть, – есть список, мы хотим ещё один список, такой же. Звучит очень просто. В чём засада? Засада в том, что нас ожидают очень и очень серьёзные и, не побоюсь этого слова, пугающие вещи. Если вы что-то программировали на других языках, то в них такого эффекта надо добиваться, а здесь оно приползает само.

Пишем идеально простой код:

```
a = [1,2,3,4,5]

b = a
print a
print b
```

И получаем идеальный, ожидаемый и очевидный результат:

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

Всё хорошо? А теперь изменим по одному элементу – один, первый, в первом списке и один, последний, во втором.

```
a[0] = -101
b[4] = -105
print a
print b
```

```
[-101, 2, 3, 4, -105]
```

```
[-101, 2, 3, 4, -105]
```

Мы хотели изменить по одному элементу в каждом списке, а изменили по два элемента в обоих. Почему? Легко заметить, что списки наши как были вначале одинаковыми, так одинаковыми и остались. Более того, они навсегда останутся одинаковыми, что бы мы ни делали. Причина проста – никаких двух списков нет, есть только один список, который мы изменяем. И есть два имени – `a` и `b`, которые указывают на один и тот же список. Непонятно? Мы к этому ещё вернёмся.

Пока подойдём к задаче практически и потребительски. Нам надо скопировать список так, чтобы после копирования этот и новый список были *разными* списками. Как и любую задачу, и эту можно решить разными средствами. Мы выберем вариант, который, возможно, и не является лучшим, зато не требует от нас никаких новых знаний. Мы *создадим* пустой список и дополним его элементами из первого списка. Слово создадим не случайно выделено курсивом, к этому мы тоже ещё вернёмся.

```
a = [1,2,3,4,5]

b=[]
for i in xrange(0,len(a)):
    b.append(a[i])
print a
print b

a[0] = -101
b[4] = -105
print a
print b
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[-101, 2, 3, 4, 5]
[1, 2, 3, 4, -105]
```

Да, это выглядит примитивным. Но это работает.

2. Опять очень простая задача, но без подвоха. Сложить поэлементно два списка и отправить суммы в первый. Если длина списков разная, выполнить для минимального числа элементов.

```
a = [1,2,3,4,5]
```

```
b = [10,20,30]

for i in xrange(0, min(len(a),len(b))):
    a[i] = a[i] + b[i]

print a
print b

[11, 22, 33, 4, 5]
[10, 20, 30]
```

3. Есть список, в нём элементы, положительные и отрицательные. Переписать все положительные в новый список, но не на свои старые места, а сплошь, то есть: [1,2,-3,4,-5] ➔ [1,2,4]

```
a = [1,2,-3,4,-5]

b = []
for i in xrange(0, len(a)):
    if a[i] > 0:
        b.append(a[i])

print a, b
```

4. Задача сложнее. Два списка. Из первого мы хотим взять N1 элементов. Из второго, как нетрудно догадаться, N2 элементов. Всё это объединить и записать в первый список. Но чтобы записать в первый список что-то нужное, надо удалить из него что-то ненужное, то есть всё, что идет после N1-го элемента.

```
a = [1,2,3,4,5]
b = [101,102,103,104]

N1 = 2
N2 = 3

for i in xrange(len(a)-1,N1-1,-1):
    del a[i]
for i in xrange(0,N2):
    a.append(b[i])
print a
```

На что обратить внимание? Много на что. Второй цикл понятен, но почему первый идёт от конца к началу? А вы попробуйте наоборот, чисто в уме. Если вы хотите удалить элементы с третьего по пятый – для простоты я имею в виду их физическое размещение в списке, а не индексы, то, когда вы удалите третий, оставшиеся два сдвинутся налево. Удаляя четвёртый,

вы, на самом деле, удалите пятый. А удаляя пятый, которого уже и нет вовсе, программа завершится аварийно.

Присмотритесь ещё раз к этому же циклу. Вспомните, цикл выполняется от первого параметра, *включая* его, до второго параметра, *не* включая его. Именно так, от первого до второго, а не от меньшего к большему. И ещё – задачу удаления элементов из списка можно решить много проще. Мы к этому вернёмся, чуть позже.

Вот прямо сейчас посмотрел и задумался – а это ничего, что элементы в списках у меня, как правило, идут по возрастанию? Вдруг вы подумаете, что это обязательно надо? Или мы добавляем элементы в список, а они автоматически сортируются? Нет, это не так. Но на таких примерах удобнее проверять правильность работы программы – ошибки сразу видны. Обдумайте.

5. Не очень сложная задача. Есть список. Надо записать его в обратном порядке. Для начала в другой список, потом в самого себя. Первая подзадача решается двумя способами. Первый способ на основе уже изученного:

```
a = [1,2,3,4,5]

b = []
for i in xrange(len(a)-1,-1,-1):
    b.append(a[i])
```

Обратите внимание на минус единицу в качестве второго параметра – а как иначе, если нам надо, чтобы цикл выполнялся до нуля? Теперь применим другую технологию, новую и неопробованную:

```
b = []
for i in xrange(0,len(a)):
    b.append(a[-i-1])
```

Этот вариант тоже работает, но *почему* он работает? До сих пор мы использовали только положительные индексы при обращении к спискам. Положительный индекс указывает именно на то, чему он и равен. Почти. Равен пяти – указывает на шестой элемент. Равен нулю – на первый. Отрицательные индексы работают почти так же, но отсчёт ведётся *справа*. НО! Самый правый, самый последний элемент имеет индекс минус один.

А почему не ноль? А как тогда отличить его от самого левого? Итак – последний – минус один, предпоследний – минус два, и так далее. Это надо обязательно запомнить. А теперь меняем местами без второго списка:

```
for i in xrange(0,len(a)/2):
    a[i],a[-i-1] = a[-i-1],a[i]
```

Здесь мы пользуемся возможностью одновременно присваивать значения двум – и более переменным. А если бы такой возможности не было? Использовать промежуточную переменную? Для эстетов, код без дополнительных переменных:

```
for i in xrange(0,len(a)/2):
    a[i]=a[i] + a[-i-1]
    a[-i-1]=a[i] - a[-i-1]
    a[i]=a[i] - a[-i-1]
```

6. Предпоследняя задача пусть будет искусственно выдуманной и практически бесполезной. Заполним список случайными целыми числами в диапазоне до ста. Переписать в другой список только точные квадраты. Квадратичность числа будем проверять тупо – перебором.

```
import random

a=[]

for i in xrange(0,30):
    r=random.randint(0,100)
    a.append(r)
print a

b=[]
for i in xrange(0,30):
    for k in xrange(0,a[i]):
        if k*k == a[i]:
           b.append(a[i])
print b
```

Доработайте, упростите и оптимизируйте.

7. Последняя задача имеет некоторый практический смысл. Сама по себе она не встречается, но является частью других, более сложных задач. Итак, есть числа в количестве N, для конкретики пусть N=10. Сами числа это 1,2,3…N. Надо разместить эти числа в списке из десяти элементов в

случайном порядке и так, чтобы каждое число встречалось ровно один раз. Задача не выдуманная, вполне реальная. Как это реализовать?

Первый способ. Добавляем элемент в список, пусть нулевой, затем случайным образом подбираем такое его значение в заданном диапазоне, которого в списке ещё нет. То есть список проходит такие метаморфозы: []➔[7]➔[7,2]➔[~~7,2,7~~]➔[7,2,3]➔... Зачёркивание как бы намекает, что вариант плохой, негодный. Реализуемо, но требует умственных усилий.

Второй способ. Формируем список из *заведомо неправильных,* например отрицательных, элементов, а затем для каждого значения случайным образом выбираем пока ещё пустой элемент. Где-то так:[-1,-1,-1]➔ [-1,7,-1]➔[-2,7,-1]. Можно, но тоже будет нелегко. Подумайте о том, что оба способа не гарантируют, что процесс когда-то завершится. Нет, практически он наверняка закончится – но а вдруг? Поэтому –

Третий способ. Заполняем список числами подряд, по возрастанию – то есть [1,2,3...]. Затем много-много раз меняем два случайно выбранных элемента местами. Вот так:

```
import random

a = []
for i in xrange(0,10):
    a.append(i+1)

for k in xrange(0,1000):
    ind1 = random.randint(0,9)
    ind2 = random.randint(0,9)
    if ind1 <> ind2:
        a[ind1],a[ind2] = a[ind2],a[ind1]

print a
```

Философские замечания о методологии программирования. Первое – ещё раз настоятельно рекомендую прочесть хотя и старую, но не стареющую книгу:

Лу Гринзоу «Философия программирования Windows 95/NT», Санкт-Петербург, «Символ», 1997
Lou Grinzo Zen of Windows 95 Programming.

Второе. Я долго думал, не написать ли `for i in xrange(0,10):` Всё-таки вместо этого решил добавить единицу в следующую строку. Достаточно ли тысячи обменов для качественной случайности? А сколько обменов будет на самом деле, ведь в случае равенства индексов обмен не производится? И главный вопрос – а зачем я проверяю индексы на равенство? Неужели, если они равны, переприсваивание не сработает и сломается? Отвечаю – не знаю и знать не хочу. И проводить опыты не хочу. Я лучше проверю индексы. Такая у меня философия программирования. И ещё одно, уже не вопрос, а проявление удивления. Заполняем мы список в диапазоне (0,10), а случайный элемент из него же выбираем в диапазоне (0,9). Это претензия к разработчикам языка, если что.

Вложенные списки.
Или многомерные, как вам больше нравится

Все списки, которые мы создали, поменяли, переделали и вообще испортили, имели нечто общее – все они содержали элементы целого типа. Как вы, наверное, уже догадались – это не обязательно. Могут быть списки и плавающих чисел и строк:

```
b = [3.14, 3.14**2, 3.14**3, 3.14**4]
c = ['мой', 'дядя', 'самых', 'честных', 'правил']

print b
print c[0] + c[1] + c[2] + c[3] + c[4]
```

```
[3.14, 9.8596, 30.959144000000002, 97.21171216000002]
мойдядясамыхчестныхправил
```

К сожалению, пробелы между словами надо вставлять самому. Списки могут содержать и элементы разных типов:

```
d = [1, 99, True, 2.87**3.14, 3.14**2.87, 'It was many and many a year ago']
print d
```

```
[1, 99, True, 27.399911226570875, 26.680139129184198, 'It was many and many
a year ago']
```

комментарий для продвинутых

Строковый элемент является началом стихотворения американского поэта Edgar Allan Poe «Annabel Lee» на американском языке. Хотите произвести

впечатление на девушку, выучите, поможет. Если девушка американская, конечно. Хотя и на нашу няшу стихи на непонятном языке окажут стимулирующее воздействие.
Почему возведение в степень? В одной из книг Мартина Гарднера – прочитайте обязательно – я нашёл вопрос: что больше, e^{π} или π^{e} ? Наконец-то я нашёл время и место это узнать. И вам тоже пришлось.
конец Комментария

Что важнее и интереснее, список может содержать и другие списки. Простой пример. Есть два списка:

```
a = [1,2,3,4,5]
b = [101,102]
```

Добавляем второй к первому:

```
a = a + b
print a
```

```
[1, 2, 3, 4, 5, 101, 102]
```

Элементы второго списка были добавлены в хвост первого, добавлены по одному, как отдельные элементы. А теперь добавим второй список как отдельный элемент.

```
a.append(b)
print a
```

```
[1, 2, 3, 4, 5, [101, 102]]
```

Те же числа внутри первого списка, но сохранилось их объединение в тоже список. Как обратиться к такому элементу? Точно так же, как и к обычному, по индексу.

```
a0 = a[0]
a5 = a[5]
print a0, a5
```

```
1 [101, 102]
```

Но так мы получили весь список. А как обратиться к одному отдельному элементу того списка, который внутри? Это немного сложнее:

```
a50 = a[5][0]
print a50

101
```

Изменение значений совершенно аналогично, заменим внутренний список целиком, а потом отдельный элемент в нём:

```
a[5] = [0.99, 0.999, 0.9999]
print a
a[5][1] = 'ax+b'
print a

1, 2, 3, 4, 5, [0.99, 0.999, 0.9999]]
[1, 2, 3, 4, 5, [0.99, 'ax+b', 0.9999]]
```

Как видите, элемент вовсе не обязательно заменять элементом того же типа, а список – списком той же длины. Списки – это очень мощный, гибкий и на лету сам себя меняющий инструмент. Хотя я и не уверен, является ли похвалой определение «гибкий» в отношении инструмента – гибкий молоток вызывает определённые сомнения. Так что для начала ограничим, и сильно ограничим, наш инструмент. Займёмся списками, содержащими списки, которые содержат списки… которые в итоге содержат только числа.

Сначала маленькая, без особого смысла, вводная задача. Потом большая и почти практическая. Маленькая задача такая – а как, собственно, создать двумерный список? Прямоугольный, например пять строк по три элемента в каждой. То, что в математике называется *матрица*. Как создать руками, выписав все элементы, понятно. А как в цикле? Понятно только, что циклов должно быть два и они должны быть вложенными. Предлагаю создать временный список, соответствующий одной строке матрицы, и добавлять его. Показываю на пальцах – создали список [1,2,3] – добавили, еще список [4,5,6] – опять добавили.

```
a=[]
n=5
m=3

for i in xrange(0,n):
    wrk=[]
    for k in xrange(0,m):
        wrk.append(0)
    a.append(wrk)
```

```
print a

[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Список заполнен без затей, нулями. Нудные читатели могут задать вопрос – а почему я решил, что у нас пять строк по три элемента? Почему не пять столбцов по три элемента в каждом, ведь можно понять эту запись и так? Просто потому, что так принято. В большинстве языков программирования матрицы задаются по строкам. Единственно заметное исключение – Фортран. И ещё напоминание, хотя об этом уже и говорилось, но немного другими словами. Мы можем обратиться как к любому элементу в отдельности, так и ко всей строке целиком:

```
b = [[1,2,3], [4,5,6], [7,8,9]]
print b[1]
b[1] = [444,445,446]
print b

[4, 5, 6]
[[1, 2, 3], [444, 445, 446], [7, 8, 9]]
```

Теперь задача сложная, объединяющая в себе несколько задач попроще. Сначала придадим всем этим абстрактным строкам и столбцам конкретный физический смысл. Есть большой список – это год. В нём списки поменьше – это месяцы. В них, как легко догадаться, дни, точнее – дневная температура. Программа рассчитывает среднюю температуру за каждый месяц и выводит. Простой вопрос – а откуда возьмутся исходные данные? Простой ответ – мы их сгенерируем случайным образом, но с оттенком правдоподобия, чтобы в декабре было холоднее, чем в мае.

В разных месяцах разное количество дней, если вы не знали. И выводить данные по месяцам лучше рядом с названиями месяцев. И для генерации не совсем бредовых температур неплохо иметь под рукой средние температуры по месяцам. Это, разумеется, придётся вводить руками, вот этими собственными руками. Существование високосных годов мы игнорируем. Если что, далее – средняя температура *у нас*. Какая у вас средняя температура, я не знаю.

```
import random
mnum = 12

mnames = [ 'January', 'February', 'March', 'April', 'May', 'June', 'July',
'August', 'September',  'October', 'November', 'December']
mdays  = [ 21,28,31, 30,31,30, 31,31,30, 31,30,31]
```

```
mtemps = [ -8.0,-4.6,-0.8, 5.4,12.7,15.3, 18.2,17.6,11.2, 3.3,-2,-2.7]
```

А здесь мы правдоподобно заполняем весь год по месяцам псевдослучайными температурами:

```
year = []
for i in xrange(0,12):
    month = []
    for k in xrange(0,mdays[i]):
        month.append(random.uniform( mtemps[i]-5, mtemps[i]+5))
    year.append(month)

#print year
```

Я-то, конечно, вывод раскомментировал и проверил – данные выглядят более-менее правдоподобно, но уж очень длинные – здесь приводить не буду.

совет
Выясните, чем отличается равномерное распределение случайной величины от нормального распределения её же. Догадайтесь, какое из них у нас в живой природе. Поменяйте распределение на другое, функция есть в модуле **random**.
конец Совета

А теперь основная часть работы, считаем среднюю температуру по палате, извините, по месяцу. И красиво выводим, конечно. Результаты подсчёта, как вы понимаете, отправятся в отдельный список.

```
mavs = []
for i in xrange(0,12):
    mavs.append(0)
    for k in xrange(0,mdays[i]):
        mavs[i] = mavs[i] + year[i][k]
    mavs[i] = mavs[i] / mdays[i]

print mavs
```

Разумеется у меня не хватило трудолюбия, чтобы проверить правильность вычисления среднего. Но я посмотрел на результат – выглядит похоже на правду. Теперь красивый вывод – ну, не очень красивый, что получилось.

```
for i in xrange(0,12):
    print mnames[i].ljust(10),
    print mavs[i]
```

На что обратить внимание – на запятую после первой строки вывода. Это уже было, запятая запрещает переход на новую строку после первого вывода. Не очень красиво, конечно, зато просто. Интереснее и новее метод `ljust`. Как и любой метод, он вызывается через точку от имени объекта. Методы специфичны для типа объекта. Наше `mavs[i]` является объектом строкового типа, поэтому обладает методами строк. Сам по себе метод `ljust` очень прост. После обращения к нему с параметром 10, строка '123' превращается в строку '123 ', то есть получает семь дополнительных пробелов в зад. Это так и называется – выровнять по левому краю.

Общие замечания к теме. То, чем мы занимались чуть раньше, – списки из десяти списков, в каждом из которых по пять элементов, в математике называется *матрица*, а в традиционных языках программирования – *двумерный массив*. Разумеется, двухмерностью дело не ограничивается, массивы бывают и трехмерные, и большей размерности. Применение их настолько часто и массово, что к Питону прилагается специальный модуль для работы с матрицами, изучение которого, разумеется, выходит за пределы этой книги. Если вам это интересно, модуль для работы с матрицами называется `NumPy`.

В завершение две задачи для самостоятельного решения. Первая не очень сложная – повернуть квадратную матрицу набок. Было:

```
1  2  3
4  5  6
7  8  9
```

стало:

```
7  4  1
8  5  2
9  6  3
```

Всё реализуется одним циклом, точнее двумя вложенными. Вторая задача действительно сложная. Заполнить квадратную матрицу *змейкой*, начиная с левого нижнего угла. Для списка размерностью 5x5 должно получиться:

```
11 19 20 24 25
10 12 18 21 23
04 09 13 17 22
03 05 08 14 16
01 02 06 07 15
```

Сделайте это! Вы сможете.

Срезы.
Всякие странности и экзотичности

Заурядный скучный список с элементами по порядку и новое обращение к нему:

```
a = [100,101,102,103,104]
print a[1:3]

[101, 102]
```

По-русски это обычно называется *срез*. В оригинале – *slice*. В скобках два индекса, со смыслом, очень похожим на параметры функции **xrange**. Индексы, разумеется, отсчитываются от нуля. Мы выбираем элементы, начиная с первого индекса, включая его. Мы останавливаемся перед вторым индексом, *не* включая его. Срез – это тоже список, только маленький и с ним можно делать всё то же, что со списком большим.

Можно присвоить срез другой переменной или удалить:

```
b = a[0:4]
print b
del b[1:3]
print b

[100, 101, 102, 103]
[100, 103]
```

Что интереснее, можно присвоить срезу целый список, причём его длина не обязана совпадать с длиной среза. Вот три варианта: список той же длины и элементы просто заменяются; список длиннее среза и исходный список растягивается; список пустой и операция эквивалентна удалению элементов. Код я подсократил – предполагается, что после каждой операции исходный список восстанавливается.

```
a = [100,101,102,103,104]

a[1:3] = [1101,1102]
a[1:3] = [11,12,12.5,12.75]
a[1:3] = []
```

```
[100, 101, 102, 103, 104]

[100, 1101, 1102, 103, 104]
[100, 11, 12, 12.5, 12.75, 103, 104]
[100, 103, 104]
```

Можно записать и более экзотические срезы, исходный список тот же:

```
a = [100,101,102,103,104]

print a
print a[2:]
print a[:2]

[100, 101, 102, 103, 104]
[102, 103, 104]
[100, 101]
```

Немного подумав, несложно догадаться – если после двоеточия ничего нет, берётся список до самого его конца. Если до двоеточия ничего нет, список берётся от начала, а последний элемент определяется как обычно. Отсюда имеем простой способ удалить все элементы списка, начиная – и включая – с указанного и до конца:

```
del a[2:]  # будут удалены все элементы с физически третьего и до конца
```

А теперь логичное завершение:

```
print a[:]

[100, 101, 102, 103, 104]
```

Если вы считаете, что это бессмысленно и бесполезно, то вы не правы. Помните, мы пытались присвоить весь список целиком самым простым и очевидным способом `b = a`? В результате переменная `A` и переменная `B` стали указывать на один и тот же список, что не входило в наши планы. Теперь это можно записать вот так и тут же проверить:

```
a = [100,101,102,103,104]

b = a[:]
b[0] = -1
print a, b

[100, 101, 102, 103, 104] [-1, 101, 102, 103, 104]
```

Список по имени `B` теперь существует независимо от списка по имени `A`.

Теперь вопрос с подвохом. Что выведет вот такой код?

```
b = 123.45
c = [1,2,3, b]
b = 3.14
print c
```

То, что сразу кажется очевидным, – `[1, 2, 3, 123.45]`? Но ведь B это переменная и мы добавили в список именно её? Тогда `[1, 2, 3, 3.14]`. Правильный ответ первый. В этом случае Питон ведёт себя традиционно, ожидаемо и предсказуемо. А почему, собственно? Надеюсь, до конца книги ещё найдётся немного места для теории и я открою вам эту тайну – или постепенно вы всё поймёте сами по себе.

Кортежи – что такое и зачем. Очень коротко

Это очень короткий раздел. Кортеж – это по-русски. В оригинале это называется *tupple*. По-русски звучит гораздо лучше. Что это такое и зачем они вообще нужны?

Кортеж – это почти список. Сравните:

```
a = [1,2,3]   #  это список
b = (1,2,3)   #  это кортеж

print a

print b
print 'b[1] = ', b[1]
print 'b[1:]', b[1:]

[1, 2, 3]
(1, 2, 3)
b[1] =  2
b[1:] (2, 3)
```

Один в один список, только скобки другие. И доступ поэлементный есть, и срезы есть. Так в чём разница? Принципиальная разница в том, что кортеж *неизменяем*. Заменить целиком его можно, потому что с точки зрения Питона это будет другой кортеж. Отредактировать – нет, нельзя. Об неизменяемых объектах подробнее будет в главе о строках.

Ещё одна особенность кортежа. В арифметических выражениях часто используются скобки, причём без лишних ограничений – то есть если можно заключить что-то в скобки, то почему нет? Например, можно так: $(3+5+7)=(3)+(5)+(7)$. Смысла особого в этом нет, но никто и не запрещает. А как будет выглядеть кортеж из одного целого элемента? Правильно, вот так: (7). А как отличить, где кортеж, а где выражение? А кортеж из одного элемента, не важно какого, обязательно записывается таким образом: (7,)

Напрашивается очевидный вопрос – а зачем это вообще нужно? Никто не знает, но все делают вид. Самое частое объяснение – достоинство кортежей как раз в том и заключается, что их нельзя изменить. Это гарантия сохранения их целостности. Кроме того, с кортежами вы встретитесь при чтении данных из бинарных файлов.

Ещё раз квадратное уравнение

Если помните, с квадратным уравнением у нас были мелкие проблемы – или у квадратного уравнения были проблемы с нашим недостаточным опытом программирования. Неприятность заключалась в том, что корней может быть или два, или один, или ни одного. В финале программы корни выводились на экран, и для того, чтобы с выводом не случилось чего нехорошего, приходилось инициализировать обе переменные. Перечитайте, если забыли. Теперь решим проблему, применив для хранения найденных корней списки.

```
A = 0; B = 0; C = 0
roots = []

result = 'вообще ничего не понятно'

if A <> 0:
    #  квадратное уравнение, дискриминант вычисляем
    dis = B**2 - 4*A*C;

    if dis > 0:
        x1 = (-B+dis**0.5)/(2*A);
        x2 = (-B-dis**0.5)/(2*A);
        result = 'два корня'
        roots.append(x1)
        roots.append(x2)
    elif dis == 0:
        x1 = (-B)/(2*A); x2 = x1
        result = 'один корень'
```

```
        roots.append(x1)
    else:
        result = 'корней нет'

elif B <> 0:
    #  линейное уравнениё
    x1 = -C/B
    result = 'один корень'
    roots.append(x1)

else:
    #  особый случай
    if C == 0:
        result  = 'тождество'
    else:
        result = 'корней нет'

print 'a = ', A, 'b = ', B, 'c = ', C

print result
for i in xrange(0,len(roots)):
    print roots[i]
```

Лично мне показалось немного неудобной невозможность добавить в список оба корня сразу. Говоря другими словами, нельзя добавить просто так в список несколько элементов – добавить-то можно, но сначала надо создать из них список.

И да, это не последний подход к решению задачи. А пока проверьте новую программу на новых тестах.

Возможные варианты:
a <> 0

дискриминант положителен

3,10,3 два корня $-3, -\frac{1}{3}$

дискриминант отрицателен

2,4,3 корней нет

дискриминант равен нулю

5, 10, 5 один корень $-\frac{12}{3}$

a = 0, b <> 0

0, 2, -10 один корень 5

0, 5, 3 один корень $\frac{3}{5}$

a = 0, b = 0, c <> 0

0, 0, 10 корней нет

a = 0, b = 0, c = 0

0, 0, 0 тождество

Проверили? Всё ли хорошо? По глазам вижу, что не проверили. Хорошо не всё. Вспомните, типов в Питоне нет. Целые числа делятся нацело. Для теста (0, 5, 3) ответ будет неправильным. Если задать его как (0, 5.0, 3.0), то всё будет отлично. Хорошо ли это? Нет не хорошо. Можно ли это исправить? Можно. Но исправлять надо целиком, полностью и навсегда. Эти исправления мы внесём после постижения функций.

Но и это ещё не всё. В тесте (5, 10, 5) ожидаемый ответ $-1^2/_3$, реально полученный –1. Замена на (5.0, 10.0, 5.0) не помогает. Тщательно изучив исходный текст, никаких ошибок я не обнаружил. Тогда меня осенило и я пересчитал на бумажке корни уравнения. Оказалось, программа права, а я сначала ошибся. Бывает.

Глава шестая, короткая. Строки

Это будет недлинная глава о строках. Немного теории. Чем они отличаются от списков, а чем – нет. Какие у них есть методы. Какие для них есть функции. И, конечно, опыты и эксперименты. Для начала, главное, что надо запомнить, – строки очень похожи на списки. И понять, что в главном они от них отличаются.

Повторение пройденного и чем строки похожи на списки

Для простоты: строка – это то, что в кавычках.

Кавычки могут быть разными – одиночными, двойными, но это не важно – если внутри нет других кавычек. Если строка в одиночных кавычках, то внутри могут быть двойные. Если строка в двойных кавычках, внутри строки могут быть одиночные. Далее увидите на примерах.

Операции над строками – сложение и умножение, никому не нужное. Показываю:

```
s1 = 'Маленькой ёлочке '
s2 = 'холодно зимой'
s3 = s1 + s2

print s3

Маленькой ёлочке холодно зимой
```

А почему умножение бесполезно? Демонстрирую:

```
s4 = s2 * 3

холодно зимой холодно зимой холодно зимой
```

И кому это может понадобиться? Единственный смысл – провести по экрану линию поперёк, что-то вроде `print '-'*80`. Идём дальше. Строки состоят из отдельных символов. Количество символов возвращает функция `len(s)`. Как частный случай, в строке может быть ноль символов. Это называется пустая строка, а записывается так: `s = ''`

К символам можно обращаться отдельно, по индексу, так же, как к элементам списка. Значит, точно так же со строками можно работать в цикле. Простая задача – посчитать, сколько раз в строке встречается буква ‘a’.

```
s = 'мой дядя самых честных правил'

numA = 0
for i in xrange(0,len(s)):
    if s[i] == 'a':
        numA = numA + 1
print "num of A's = ", numA

num of A's =  2
```

Всё понятно, ничего нового. Но, чтобы хоть что-то новое всё-таки было, перепишем цикл немного по-другому.

```
for sim in s:
    if sim == 'a':
        numA = numA + 1
```

Результат будет тот же, проверьте.

теория
Теперь обдумайте, что такое `sim`. Правильный ответ – это символ строки. А какой именно? Все символы по очереди – от первого до последнего. Такой цикл возможен только потому, что у строки есть первый символ, есть последний и для каждого символа, кроме последнего, мы можем однозначно указать следующий за ним символ.
конец Теории

Надо признаться, такая запись выглядит проще и понятнее. А почему я раньше такой способ не советовал, а теперь одобряю? А потому, что мы двигаемся дальше, и многое, доселе неясное, выступает из тумана. Немного усложним. Теперь нам надо подсчитать не вхождение в строку отдельного символа, а целой подстроки сразу. Это небольшое изменение предлагает о многом задуматься. Для конкретики будем подсчитывать число вхождений слова из трёх букв – определённого артикля *the*.

```
s = '''
Ah, distinctly I remember it was in the bleak December,
And each separate dying ember wrought its ghost upon the floor.
'''
```

```
numThe = 0
for i in xrange(0,len(s)-2):
    if s[i:i+3] == 'the':
        numThe = numThe + 1

print "number of the's = ", numThe

number of the's =  2
```

На что здесь надо посмотреть? Тройные кавычки для содержания в них многострочного текста. Лаконичная форма организации цикла из предыдущего примера не используется, потому что нам категорически нельзя начинать поиск трёхбуквенного слова, начиная с последнего или предпоследнего символа. А вот срез очень даже успешно применим. Последняя строка заключена в двойные кавычки, а внутри неё есть одиночная кавычка. Срезы точно так же можно использовать от начала и до конца строки:

```
s = 'Белеет парус одинокий'
print s[10:]
print s[:10]

ус одинокий
Белеет пар
```

Чуть не забыл, допускаются и отрицательные значения, как в индексах, так и в срезах. На этом сходство со списками, и вообще позитив, заканчиваются. Теперь о грустном.

Чем строки не похожи на списки

Определим любую строку и попытаемся выполнить очень простой и невинно выглядящий оператор:

```
s[1] = 'x'

s[1] = 'x'
TypeError: 'str' object does not support item assignment
```

Если вы попытаетесь изменить строки, используя механизм срезов, результат будет аналогичным. Что бы это значило? К сожалению, это значит, что строка является *неизменяемым* объектом. А это что значит? Да

именно это и значит – строку нельзя изменить. Этому вроде бы противоречит следующий код:

```
s = 'Белеет парус одинокий'
s = 'В тумане моря голубом'
print s

В тумане моря голубом
```

Здесь мы не пытались изменить строку. Мы присвоили переменной совершенно другое значение. Говоря точнее, наша переменная теперь указывает на другой объект типа строка. А что случилось со старой строкой? Скоро за ней придёт страшный и ужасный Сборщик мусора, утащит и утилизирует.

Методы строк

Так что же, со строками совсем ничего нельзя сделать? Как известно, если нельзя, но очень хочется, то можно. Для начала, обычно можно сделать что-то тупо и в лоб. Следующий кусок кода удаляет из строки пробелы, тупо и в лоб:

```
s = 'Скажи-ка дядя, ведь недаром'
s_out = ''

for x in s:
    if x <> ' ':
        s_out = s_out + x
s = s_out

print s
```

Возможно, вам показалось, что это ну уж очень тупо. Это потому, что задача наша была не совсем элементарна. Вспомним ту, что была чуть раньше – нам захотелось поменять первый символ строки. Задача проще – и решение проще.

```
s = 'Белеет парус одинокий'
s = 'х' + s[1:]
print s

хелеет парус одинокий
```

Повторюсь – мы не изменили строку. Мы создали новую строку. Похожую на старую. А старая строка, что с ней? Отправилась в страну вечной охоты.

По сути, мы меняем строку, но делаем вид, что её не меняем. Имя изменяемой строки должно присутствовать слева от оператора присваивания. Для того чтобы делать таким образом со строкой всё, что угодно, существуют методы. Вернёмся к задаче удаления пробелов из строки и решим её по новой технологии.

```
s = 'Скажи-ка дядя, ведь недаром'
s = s.replace(' ', '')
print s

Скажи-кадядя,ведьнедаром
```

Имя метода – `replace`. Метод, как несложно догадаться из его имени, заменяет. Заменяет он все вхождения первого параметра в строку на второй параметр. Поскольку в нашем случае второй параметр это пустая строка, то все вхождения первого параметра, по факту, просто удаляются.

Какие методы используются чаше всего? Это – копирование части строки в другую, удаление части строки и поиск одной строки в другой. Почему именно эти? Потому что так мне напоминает мой долгий опыт программирования. Другие методы тоже очень желательны, но, на крайний случай, без них можно обойтись. Если кого-то надо ещё выбрать, то это, несомненно, будет метод замены, но его мы уже знаем. Смотрим, как в Питоне обстоят дела с этими тремя необходимыми средствами.

Для выполнения копирования отлично работают срезы. Теперь найдём в строке подстроку, так эта операция официально называется:

```
s = s.find('дядя')
print 's = ", s

s =  8
```

Тут, действуя по инерции, мы имеем некоторый сюрприз – наша строка потеряна. Разумеется, надо было по-другому:

```
where = s.find('дядя')
print 'where = ', where

where =  8
```

Теперь всё хорошо и строка цела. И запомним, что метод возвращает позицию только *первого* вхождения подстроки. И ещё – если подстроки в строке нет вообще, то мы получим в ответ минус единицу.

Строки экранированные и неформатированные и кое-что ещё

Этот раздел вы легко можете пропустить. Это всё необязательно. Если вы это прочтёте, поймёте и запомните, будет лучше. А если нет – то нет. Я буду краток.

Неформатированные – для программирования слово обычное. *Экранированное* – слово странное, но что есть, то есть. Однако поговорим о стихах. Имеем грустное стихотворение:

В лесу родилась ёлочка,
В лесу она росла

Грустное – потому что кончилось всё плохо. Чем стихи отличаются от прозы? Не рифмами, рифмы могут и отсутствовать. Тем, что нельзя выдавать текст как попало, сплошным потоком. Если в стихах что-то написано на отдельной строке, то так оно и должно быть навсегда записано, на отдельной строке. Если у нас две строки про ёлочку, то так их и должно быть – две, и граница между этими двумя строками неизменна. Извините, если вы это всё и так понимали.

Теперь практическая сторона. Как в программе вывести эти две строки на двух строках? Вариант – использовать два оператора **print** не предлагать. Это, конечно, сработает – для двух строк. А если нам нужен весь Евгений Онегин на экране? Напоминаю – онегинская строфа это четырнадцать строк. Что, рисовать четырнадцать однотипных операторов? На всякий случай напомню. Вот так, совершенно очевидным способом, написать нельзя:

```
s = 'В лесу родилась ёлочка
    В лесу она росла'
print s
```

Даже и не пытайтесь. Вторая строка кода не воспринимается как часть символьной строки. Тут у нас слово *строка* употребляется параллельно в

двух значениях – общепринятом и программистском. Есть способ с тройными кавычками, с которым вы уже знакомы. Пишем так:

```
s = '''В лесу родилась ёлочка
       В лесу она росла'''
print s
```

И получаем вот так. Уже лучше, но не совсем то.

```
В лесу родилась ёлочка
       В лесу она росла
```

Можно догадаться, что программный код слева от первой части символьной строки игнорируется, а вот пробелы слева от второй части символьной строки органично входят в её состав. Требуется доработка напильником. Вот так будет лучше:

```
s = '''В лесу родилась ёлочка
В лесу она росла'''
print s

В лесу родилась ёлочка
В лесу она росла
```

Тем не менее, столько неочевидных усилий для всего лишь ровного вывода двух строк кажутся немного утомительными. Кроме того, всему этому мы обязаны тройным кавычкам. Если они где-то будут утеряны, то будет утеряно и всё разделение на строки нашей грустной песенки. И вот здесь-то и возникают из ниоткуда экранированные строки. Пишем так:

```
s = 'В лесу родилась ёлочка \nВ лесу она росла'
print s
```

И немедленно получаем именно то, что и хотели:

```
В лесу родилась ёлочка
В лесу она росла
```

Обратите внимание, что обе строки стихотворения записаны в одну строку программного кода. Что интереснее и важнее, символы `\n` в строке присутствуют, а в её выводе – нет. Вот это самое `\n` и есть экранированная последовательность. Конкретно эта последовательность производит

переход в начало следующей строки. Последовательностей много, но эта самая полезная, ей и ограничимся.

А если вам понадобится строка, содержащая обратный слеш, поставьте их два рядом. Как вариант, для решения этой же проблемы можно использовать неформатированные строки. Это может иметь смысл, если экранированных последовательностей в строке очень много. Проще показать на примере.

```
s ='111\n222\n333\n444'
print s
111
222
333
444

s =r'111\n222\n333\n444'
print s
'111\n222\n333\n444'
```

Теперь обычная задача для тренировки программиста и для проверки его квалификации – подсчитать количество слов в строке и выделить из строки слово по его номеру. Слово – это то, что не разделитель. Разделитель, как правило, это пробел. Разделителем может быть и другой символ или даже несколько, но задачу это не усложняет. То есть термин *слово*, в данном случае, вполне совпадает с его обыденным смыслом. Конечно, в Питоне можно решить задачу с нуля, своими собственными руками, но есть и уже готовые средства. Поскольку задача эта в реальной программистской жизни встречается часто, покажем на примере.

Питон подходит к решению задачи по-своему, по-питонски:

```
s = '1 23 456 7890'
ls = s.split()
print ls

['1', '23', '456', '7890']
```

Питон разобрал строку на слова и записал слова в список – а вы делайте со списком, что хотите.

Глава седьмая. Функции

Напоминание - что такое функция

Мы уже пользовались функциями. Например, функциями в обычном математическом смысле слова:

```
import math

x = math.sin(math.pi/2)
print ' Sin(pi/2) = ', x

Sin(pi/2) =  1.0
```

Волшебное слово `import` не имеет прямого отношения к функциям, так что пока не обращайте внимания. Но идея понятна – у функции есть имя – `sin`. За именем следуют скобки. В скобках параметр. Мы получаем в ответ значение синуса от этого параметра. Теперь немного другой случай, а скорее – совсем другой:

```
s = [1,2,3,4,5]
print s
del s[2]
print s

[1, 2, 3, 4, 5]
[1, 2, 4, 5]
```

Теперь мы ничего не получаем от функции, по той простой причине, что функция *не* применяется слева от оператора присваивания. Она применяется сама по себе. Она выполняет свою работу. Можно было бы сказать – функция выполняет свою функцию. Конкретно эта функция удаляет один или несколько элементов из списка.

Систематизируем. Есть функции, которые можно употреблять слева от оператора присваивания – они возвращает какое-то значении, чаще всего число или строку. Есть функции, которые можно употреблять независимо от оператора присваивания, – они просто делают что-то полезное.

А теперь сосредоточимся и пристально рассмотрим все случаи, когда функции приносят пользу.

Функции. Сделать программу понятнее

Функции делают программу проще и понятнее. Они делят её на части. Вспомним «*Евгения Онегина*»:

Все ярусы окинул взором,
Всё видел – лицами, убором
Ужасно недоволен он © Александр Сергеевич

Обратите внимание – нигде не сказано, что Онегин окинул *лица* взором. Он окинул взором *ярусы* театра. А лица – они там, в ярусах. Так проще. Самые незатейливые функции так и работают. Вот примитивный код:

```
a = 'Иванов'
b = 'Петров'
c = 'Кошкин-Собакин'

ad = '01.01.1970'
bd = '02.02.1980'
cd = '03.03.1990'
```

Зачем это надо – неважно. В реальной программе, если что-то похожее появится, то оно будет намного длиннее. Вот на этом примере и введём понятие функции.

```
def names():
    a = 'Иванов'
    b = 'Петров'
    c = 'Кошкин-Собакин'

def birthdays():
    ad = '01.01.1970'
    bd = '02.02.1980'
    cd = '03.03.1990'

names()
birthdays()
```

Программный код стал заметно длиннее, но, на мой взгляд, заметно понятнее. При следующем просмотре программы можно обращать внимание только на две последние строки. А теперь разберём, что мы такое написали.

Сначала имеем заголовок функции

```
def names():
```

Он состоит из зарезервированного слова **def**, произвольного имени функции и пары скобок с двоеточием. После нажатия на клавишу <Enter>, как и всегда в таких случаях, имеем в следующей строке автоматически вставленные пробелы. Всё то, что написано на этом уровне, обычно называется *тело* функции. Вызов функции состоит из её имени и обязательно пары круглых скобок. При вызове функции выполняется код, записанный в её теле. Проверьте.

Функции. Когда приходится повторять

Часто бывает, что один и тот же программный код повторяется в программе несколько раз. Покажем на простом примере. Эта функция, кстати, очень полезна в личном общении с людьми.

```
def odin():
    print 'Один я умный в белом пальто стою красивый'
```

Как эту функцию можно применить?

```
print 'Отговорила роща золотая'
odin()
print

print 'Я помню чудное мгновенье'
odin()
print

print 'Скажи-ка дядя ведь не даром'
odin()
print
```

```
Отговорила роща золотая
Один я умный в белом пальто стою красивый

Я помню чудное мгновенье
Один я умный в белом пальто стою красивый

Скажи-ка дядя ведь не даром
Один я умный в белом пальто стою красивый
```

Снова обратите внимание на обязательность пустых скобок после имени функции. Если про них позабыть, то никто ничего об этом не напомнит, но функция ничего и не выполнит. Это в лучшем случае, результат может

оказаться и непредсказуемым. С другой стороны, согласитесь, что экономия кода значительна.

Функции. Когда у них есть параметры

Ранее приведенные примеры использования функций вполне обычны и нормальны. То, что их объединяет, – это то, что это функции без параметров. Другими словами, на вход этих функций ничего не поступает. Ещё более другими словами, эти функции работают всегда одинаково, просто потому, что не знают, что им надо работать как-то по-другому – ведь на вход им ничего не поступает. Напишем функцию, которая работает по разному. Чтобы работать по-разному, этой функции необходимы входные параметры. Откуда вообще возникает потребность в функции? Ответ – она возникает постепенно и сама собой.

Сначала у нас есть функция, простая, из тех, о которых я говорил в предыдущем разделе. Далее определение функции, её вызов и результат работы:

```
def fillEmptySpace():
    print 'Привет'
    print 'Привет'
    print 'Привет'

fillEmptySpace()

Привет
Привет
Привет
```

Теперь проявите фантазию, вернитесь в реальность и представьте, как оно будет в реальной программистской жизни. Вы написали функцию. Она работает. Вы ею пользуетесь. Если программа, которую вы пишите, носит исключительно учебный характер, то этим всё и заканчивается. Если это реальная программа, то, рано или поздно, ваша функция понадобится кому-нибудь ещё. Зачем кому-нибудь ещё программировать то, что уже запрограммировано? Лучше взять вашу готовую функцию. Но вот этому персонажу, назовём его *пользователь*, нужно не три слова *Привет*, а два. Или четыре. Поверьте, пользователю всегда нужно что-то другое, что-то такое, чего ещё у вас нет. Сначала вы пишите для пользователя специальную функцию, которая выводит *Привет* два раза, потом четыре. Потом вы задумываетесь, правильно ли это. Если мой рассказ кажется вам

скучным и неправдоподобным, то вы правы лишь отчасти. Скучным – да, неправдоподобным – нет.

Поэтому пришло время написать универсальную функцию, которая будет выводит слово *Привет* в заданном числе экземпляров. Опять определение функции, вызов и результат. Далее обсуждение.

```
def fillEmptySpace( num ):
    for i in xrange(0,num):
        print 'Привет'

fillEmptySpace(2)

Привет
Привет
```

В чём отличие от того, что было раньше? В описании функции в круглых скобках появилось имя параметра. Появилось оно исключительно для того, чтобы быть использованным внутри функции, иначе никакого смысла в параметрах не было бы. При вызове функции, опять-таки в круглых скобках, обязательно указать параметр. Параметр не обязан быть константой, он может быть и переменной, и даже выражением:

```
n = 2
fillEmptySpace(n*2)
```

В результате получим четыре привета. Обратите внимание, что в заголовке функции никак не указано, что `num` является переменной целого типа, однако внутри функции это предполагается. С одной стороны, такая свобода не может не радовать, а с другой – может служить неисчерпаемым источником забавных ошибок.

Двигаемся дальше – если у функции может быть один параметр, то почему не быть и двум? Пусть второй отвечает за текст выводимого сообщения.

```
def fillEmptySpace( text, num ):
    for i in xrange(0,num):
        print text

fillEmptySpace( 'Ку-ку', 2)

Ку-ку
Ку-ку
```

Очевидно, если параметр не один, надо очень аккуратно следить, чтобы порядок их строго соблюдался. Проведите опыт – поменяёте при вызове параметры местами. Расскажите, что случилось, и объясните, почему.

Теперь что-нибудь сложнее. Тренироваться будем на стихах. Для начала напишем функцию, которая заменяет в строке одно слово на другое. Да, я знаю. что такая функция – точнее, метод – у нас уже есть, писать её не надо, можно пользоваться уже готовой. Но надо же с чего-то начинать? И, главное, мы усложним задачу. Вместо второго параметра, то, на что надо заменить, у нас будет список слов, из которого мы случайным образом выберем одно.

```
import random

def ourReplace(     s,
                    what,
                    forWhat):

    where = s.find(what)
    if where >= 0:
        word = random.choice(forWhat)
        s = s.replace( what, word)
        print s

s = 'Я помню чудное мгновенье'
L = ['творенье', 'варенье', 'день рожденья',  'самоговаренье' ]
ourReplace( s, 'мгновенье', L)
```

Программа работает, я проверял. Для чего условный оператор? Чтобы иметь хоть какую-то гарантию, что функция не рухнет, если слова в строке нет. А кстати, она рухнет? Я сначала по привычке написал проверку и только потом подумал. Зато функция совершенно точно рухнет, если список слов для замены окажется пустым. Добавьте проверку. Функция, которая результат своей работы может только вывести на экран, обладает слишком узкими и специализированными возможностями. Лучше было бы вернуть изменённую строку для дальнейшего использования. Этим мы займёмся в следующем разделе. Точнее, через один раздел, потому что следующий будет посвящён важным теоретическим вопросам.

Функции. О параметрах подробнее

Этот раздел вы можете пропустить, если ваша цель научиться программировать как можно быстрее. Мне верить можно. Если я говорю,

что вы без чего-то обойдетесь – значит обойдётесь. Однако если вы хотите узнать, как работает Питон за пределами самых простых программ, – прочитайте. И если вы хотите знать, как надо писать программы на любом языке программирования, – тоже прочитайте. Конечно, в этом разделе вы не узнаете все секреты ремесла, но хоть что-то узнаете.

Введём важные понятия. Они, эти понятия, есть во всех традиционных языках программирования, а вот в Питон придётся их тащить контрабандой. Есть *формальные параметры* и есть *фактические параметры*. В случае функции `ourReplace` формальные параметры это `s, what, forWhat`. Они неизменны, до тех пор, пока мы не изменим код функции. А фактические параметры – это `s, 'мгновенье', L`. Они, вообще говоря, всегда разные, то есть отличаются – от одного вызова функции до другого.

В Питоне нельзя до вызова функции автоматически проверить соответствие типов фактических параметров формальным и соответствие их по смыслу. Есть такое экстремистское мнение, что в этом кроется огромное преимущество Питона. О чём речь?

Разбираемся по порядку. Первое – соответствуют ли фактические параметры формальным? А зачем? Хотелось бы показать на максимально простом примере, но все они уже бесконечное количество раз использованы, и для Питона, и для других языков. Поскольку деваться мне некуда, беру самый простой пример из самых возможных простых. Ничего проще сложения нет, напишем функцию, которая складывает. Зачем? Просто для демонстрации возможностей. Поскольку мы ещё не научились писать функции, возвращающие значения, наша функция будет просто тупо выводить результат сложения на экран.

```
def ourPlus( x,y):
    z = x + y
    print 'z = ', z
```

Просто и очевидно. А теперь вызываем функцию, целых четыре раза:

```
ourPlus( 1,2)
ourPlus( 1.23, 3.456)
ourPlus( 'Oh ', 'babe!')
ourPlus( [1,2,3,4,5], [77,88,99])
```

```
z =  3
z =  4.686
z =  Oh babe!
z =  [1, 2, 3, 4, 5, 77, 88, 99]
```

Что видим? Мы видим, что на вход нашей функции можно подать фактические параметры самых разных типов и мы – сюрприз! – получим вполне осмысленный результат. Это, безусловно, хорошо, и это считается одним из преимуществ языка Питон. Но вся эта красота напоминает древнюю, неописуемой красоты вазу, вокруг которой надо ходить бережно и на цыпочках – а то ваза вдребезги. Напишем вот такой совершенно невинный вызов:

```
ourPlus( [1,2,3], 101)
```

Казалось бы, чего проще. Ведь абсолютно ясно, что мы должны получить в результате – [1,2,3,101]. Получим мы, однако, нечто совсем другое и неожиданное:

```
TypeError: can only concatenate list (not "int") to list
```

Нельзя добавить целое число к списку, даже если список полностью состоит из целых чисел. К списку можно добавить только другой список, вот так, например:

```
ourPlus( [1,2,3], [101])

z =  [1, 2, 3, 101]
```

Тоже неплохо, но иллюзии тают и летят по ветру. В традиционных языках программирования с этого начинают, в Питоне до этого приходится доходит самому.

И опыт, сын ошибок трудных... © А. С. Пушкин

Вернусь с того, с чего начал. Питон не проверяет соответствия типов фактических параметров формальным. Значит, должны проверить мы. Такие способы в Питоне предусмотрены. Определим переменные разных типов:

```
x = 2
y = 3.1
```

```
s = 'abcd'
L = [1,2,3,4,5]
```

А теперь попытаемся проверить тип первой из них.

```
if type(x) == int:
    print 'yes'
else:
    print 'no'

yes
```

Проверка сработала. А вот так надо проверять остальные три типа:

```
if type(y) == float:
if type(s) == str:
if type(L) == list:
```

Обогащённые новыми знаниями, добавляем необходимые проверки в нашу функцию.

```
def ourReplace(     s,
                    what,
                    forWhat):
    if (type(s) == str) and (type(what) == str) and \
     (type(forWhat) == list):
        where = s.find(what)
        if where >= 0:
            word = random.choice(forWhat)
            s = s.replace( what, word)
            print s
```

Обратите внимание на обратный слеш и вспомните, что он обозначает. Для тех, кто не знал, да ещё и забыл, напоминаю – означает он продолжение оператора на следующей строке. Наша функция теперь проверяет входные данные на соответствие их, данных, операциям, над ними проводившимся. Хорошо это или плохо? Я считаю, что это хорошо. Наверное, это потому, что я программировал на многих языках. Фанатичные сторонники Питона считают, что проверки – это плохо. Это снижает универсальность кода. Ведь может быть, как в предыдущем примере про сложение – на вход функции дадут что-то совсем другое, неожиданное, и функция неожиданно для всех выдаст правильный результат. Вот из-за обсуждения таких философских вопросов я и не настаивал на чтении вами этого раздела.

Теперь о вещах важных и бесспорных – для меня бесспорных. Проверять или не проверять параметры на типы – личное дело каждого. А вот проверять входные данные на адекватность и корректность – необходимость, это уже от особенностей языка не зависит. В нашем случае желательны проверки на вхождение заменяемой подстроки в исходную строку и на то, что список слов для заменяя не является пустым. Реализуйте это сами.

Настоящие функции

Настоящие функции – те, которые возвращают значение. Говоря по-другому – те, которые можно использовать справа от оператора присваивания. Вот синус, это настоящая функция.

```
import math

x = 0.5
f = math.sin(x)
print 'sin(', x, ') = ', f

sin( 0.5 ) =  0.479425538604
```

Имя функция употребляется справа от оператора присваивания. Говоря по-другому, вместо имени функции там могло бы быть число. Ещё более говоря по-другому, имя функции и есть число. В таком случае говорят, что функция *возвращает* значение. Наш синус получает на вход значение 0.5 и возвращает значение sin(0.5)= 0.479425538604.

Опять-таки, в конкретном нашем случае, в каком-то другом оно могло бы быть и строкой и иметь логический тип. Функция, обычно, имеет параметр или несколько параметров. Бывают функции и без параметров, редко, но бывают. Например, у функции, возвращающей текущее время, параметров обычно не бывает.

Пора написать свою собственную функцию, чтобы она получала параметр на вход и её можно было бы использовать с правой стороны оператора присваивания. Начнём с самого простого, что только может быть. Пусть наша функция всегда возвращает значение *пять*. Примитивно, но достаточно для демонстрации идеи.

```
def onlyFive():
    result = 5
```

```
    return result

f = onlyFive()
print 'onlyFive = ', f

onlyFive =  5
```

В этом маленьком примере можно увидеть всё, что требуется от настоящей функции. Главное в функции это оператор `return`, который и возвращает её значение. Хорошим правилом является возвращать какую-то переменную со стандартным именем, обычно `result`. Мы с самого начала её появления в программном коде догадываемся, что именно она и будет возвращаемым результатом функции.

Для демонстрации чуть более сложного применения понятия функции обычно используется вычисление факториала. Причина в том, что в большинстве языков программирования функция вычисления факториала начисто отсутствует. Синус есть почти везде, а факториала почти нигде нет. Не будем отступать от традиции. Факториал мы уже вычисляли, но я сразу предупредил, что к этой задаче мы ещё вернёмся, на другом уровне.

Напоминаю, что такое факториал. Эта функция обозначается восклицательным знаком и определяется так: $N!=1\times 2\times 3\times ...\times N$, то есть $4!=1\times 2\times 3\times 4=24$.

для занудных
По определению считается, что $0!=1$. А почему, собственно? Так я ведь уже сказал – *по определению*.
конец Для

```
def Fuct( N ):
    result = 1
    for i in xrange( 2, N+1):
        result = result * i
    return result

f = Fuct(4)
print f

24
```

Мне не нравится N+1 в цикле – а что делать? Если мы хотим, чтобы цикл выполнялся до N, должны писать N+1. Ещё один опыт с функциями.

Вычислим синус. Вычислим не путём вызова одноимённой функции, а по-настоящему, суммируя ряд. Что это значит. Есть вот такая формула:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

Формула для вычисления синуса подозрительно похожа на формулу для вычисления числа π, и в этом есть глубокий математический смысл, об этом в другой раз. *Предел суммы* этого *ряда* в точности равен синусу. Что такое сумма ряда? Когда складывают несколько его членов – два, двести или двести тысяч. А предел – это когда складывают бесконечно много членов. Для нас эти математические тонкости непринципиальны. Прикиньте – в уме, разумеется – чему примерно равен хотя бы десятый член этого ряда. Так что просуммируем первые двадцать членов и будем считать это синусом. Чем ещё интересен этот пример, так это тем, что в формуле присутствует факториал. Для него мы только что запрограммировали функцию, так что теперь наша новая функция будет обращаться к нашей старой.

Как вообще программируют такого рода вещи, то есть связанные с рядами? Очень желательно записать выражение для общего члена ряда, то есть формулу, по которой, зная номер ряда N, можно сразу определить член ряда с номером N. После этого программирование становится элементарной задачей. Для начала, поскольку первый член ряда выглядит как-то не как все, перепишем его по общему образцу. $\sin x = \frac{x^1}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$ Так понятнее и легче догадаться об общей формуле: $a_N = \frac{x^{2N-1}}{(2N-1)!}$. Теперь можно и программировать.

```
def ourSin( x ):
    result = 0
    for N in xrange(1,20):
        result = result + (x**(2*N-1))/Fuct(2*N-1)
    return result

x = 0.5
f = ourSin(x)
print 'sin(', x, ') = ', f
```

Результат вам понравился? Да или нет? А вы запускали программу, вот честно, ведь нет? Я очень честный, и если, не проверяя, поместил в книгу программу, а в ней ошибка, то я в этом признаюсь и даже рассказываю, как я эту ошибку искал. Всегда полезнее учиться на чужих ошибках. А что ошибка здесь есть, я догадался, увидев ответ. Ответ получился таким, сравните его с полученным от уже готового, питонского синуса:

```
sin( 0.5 ) =  0.521095305494
```

Вы, конечно, уже нашли ошибку, причём сразу. А я нет. Сначала я подумал, что ряд очень медленно сходится и надо подбавить в сумму членов, к примеру до ста. При попытке подбавить Питон разумно ответил, что 100! – это слишком и вычислению не подлежит. После чего я немедленно заметил, что ряд для синуса *знакопеременный*, то есть его члены меняют знак с плюса на минус и обратно. Исправил одну строку и получил новый результат.

```
result = result + (-1)**(N+1)*(x**(2*N-1))/Fuct(2*N-1)

sin( 0.5 ) =  0.479425538604
```

Гораздо лучше. Задумайтесь – а нужно ли вычислять двадцать членов ряда? Разумеется, для учебной программы это неважно, но внутри того синуса, который в Питоне, ведь тоже находится что-то подобное. Вопрос о том, сколько же членов суммировать, на самом деле чрезвычайно сложен. Он рассматривается в одном из разделов высшей математики, вычислительной математике. Вы уже заметили, что цикл будет выполняться не двадцать, а только девятнадцать раз. В свете всего сказанного выше – да какая разница? Теперь о важном. Исходный код обеих функций, и факториала, и синуса, должен находиться в одном файле, что-то вроде *theBestSinus.py*. Но это только пока. В реальных программах коды часто применяемых функций обычно выносятся в отдельный файл. В Питоне – и в других языках – это называется *модуль*. Мы этим скоро займёмся.

И ещё один, не побоюсь этого слова, дурацкий вопрос. Если мы используем функцию в качестве переменной, то есть справа от оператора присваивания или в операторе печати, то что мы получим? Правильно, то что она возвращает в операторе **`return`**. А что, если его нет вообще? Творцы Питона об этом уже подумали, вот ответ:

```
def nothing():
    print 'do nothing'

xxx = nothing()
print 'xxx = ', xxx
```

```
do nothing
xxx =  None
```

Это самое `None` вовсе не означает, что случилась какая-то ошибка. Это специальное значение, которое функция возвращает, когда она ничего не возвращает. Какой в ней смысл и как эту переменную можно использовать, обдумайте сами.

Ещё о функциях

Функции могут возвращать не только числовые значения. Помните вот эту, очень глубокомысленную?

```
def odin():
    print 'Один я умный в белом пальто стою красивый'

print 'Отговорила роща золотая'
odin()
```

Нет, к содержанию вопросов нет, вопросы к форме. Такое её применение выглядит несколько неуклюже. Заменим старую функцию на новую, возвращающую значение.

```
def odin( s ):
    result = s + '\nОдин я умный в белом пальто стою красивый'
    return result

print odin( 'Четвёртые сутки пылают станицы')
```

```
Четвёртые сутки пылают станицы
Один я умный в белом пальто стою красивый
```

Оцените экранированную последовательность в начале добавляемой строки. Вы ведь помните, что такое экранированная последовательность? Теперь пример чуть проще, или чуть сложнее. Функция возвращает логическое значение, то есть *да* или *нет*. Отвечать таким образом функция будет на вопрос *Является ли число простым?*

```
def prime( n ) :
    result = True
    for i in xrange( 2, int(n**0.5)+1):
        if n % i == 0:
            result = False
            break
    return result

print prime(18)
print prime(19)

False
True
```

Для чего плюс один? А для надёжности. Обдумайте. Однако функции могут возвращать и более сложные типы данных, например списки. Решим простую, безо всякой математики, задачу. Есть список, в нём всякое разное. Пусть в нём останутся только целые числа. То есть заготовка функции, её вызов и её результат должны выглядеть так:

```
def onlyNums( L ):
# что-то там

a = [1,2,3, 'собачка Муму', True, 3.1415, 7]
a = onlyNums(a)
print a

[1, 2, 3, 7]
```

Так вот, эту простейшую задачу мы запрограммируем тремя способами – одним неправильным и двумя правильными. Это не потому, что я не знаю, чего бы мне ещё написать. Это потому, что в эту западню рано или поздно влетает каждый программист. Только одни очень быстро учатся, а другие не учатся никогда. Я бы предпочёл, чтобы вы научились прямо сейчас и немедленно.

Первый способ быстрый и изящный:

```
def onlyNums( L ):
    for x in L:
        if type(x) <> int:
            L.remove(x)
    result = L;
    return result
```

Для чего вроде бы ненужная переменная `result`? А для того, что я имею мнение – в конце функции мы должны иметь **`return`** `result` и это обязательно. Результат, однако, огорчает:

```
[1, 2, 3, True, 7]
```

Сначала возникают всякие хитрые гипотезы – а вдруг логический тип внутри себя самый обычный целый? В некоторых других языках программирования это обычное дело. Увы, нет. Попробуйте. Удалите из списка логическую переменную, замените чем-то другим – или не заменяйте, ответ всё равно будет неверным. Здесь мы имеем вечную проблему удаления из списка, и Питон в этом отношении ничуть не лучше других языков. Это надо понять, раз и навсегда. Есть список: [1,2,3]. Мы хотим удалить из списка все чётные элементы – то есть все целые элементы, являющиеся чётными. Мы пишем цикл, от единицы до трёх. Это если у нас простой, грубый, примитивный и традиционный язык программирования. В Питоне мы пишем красиво и изящно, но в результате та же фигня. Мы перебираем элементы: первый, второй, третий. Второй элемент – чётный. Мы его удаляем. Элементов остаётся два, потому что нынешний третий стал вторым. Но цикл-то наш уже выполнен до двух и сейчас будет выполняться для трёх. А третьего элемента уже нет и цикл, в лучшем случае, просто не выполнится. В худшем случае, получим сообщение об ошибке.

Это именно та ошибка, которую все совершают, причём во всех языках программирования. Если у вас есть список, или как это в вашем языке называется, то удаление элементов производится при прохождении списка от конца к началу! В этом случае после удаления второго элемента первый элемент так и останется первым. Запомните. Многие запоминают и немедленно забывают. Что это означает для нашего случая:

```
def onlyNums( L ):
    for i in xrange( len(L)-1, -1, -1):
        if type(L[i]) <> int:
            del L[i]
    result = L;
    return result

a = [1,2,3, 'собачка Муму', True, 3.1415, 7]
a = onlyNums(a)
print a
```

```
[1, 2, 3, 7]
```

Не боюсь показаться назойливым – запомните, пожалуйста. А теперь кончим строить из себя самых умных и решим задачу тупо и в лоб.

```
def onlyNums( L ):
    result = []
    for x in L:
        if type(x) == int:
            result.append(x)
    return result

a = [1,2,3, 'собачка Муму', True, 3.1415, 7, ['Why?', 'Where?', 'Why not?']]
a = onlyNums(a)
print a
```

```
[1, 2, 3, 7]
```

Для разнообразия я решил напомнить, что элементом списка может быть и список. Кроме того, если я об этом уже говорил – то повторюсь. Каждый оператор Питона должен быть записан на одной строке. Несколько операторов на одной строке – можно. Но если обязательно надо записать длинный оператор и в одну строку он не влезает – его можно искусственно склеить символом обратный слеш – \. Но иногда строки склеиваются в один оператор сами собой – а именно тогда, когда мы разделяем оператор внутри скобок. Сейчас у нас левая квадратная скобка в верхней строке, а правая квадратная скобка – в нижней. Всё вместе автоматически считается одним оператором.

И ещё о функциях. Всякое не очень обязательное

В этом разделе будет всякое разное, относящееся к функциям, что знать не обязательно. Точнее, это я так считаю, что знать это всё не обязательно, у других программистов может быть своё мнение.

Первое. Начну с того, что я считаю полезным и сам применяю. Это *параметры по умолчанию*. Напишем нехитрую процедуру, которая вычисляет вес цилиндра. Напоминаю, вес чего-либо это произведение его объёма на удельный вес. Объём цилиндра $V = \pi R^2 h$, где R радиус основания, h – высота цилиндра.

```
def weightCyl ( R, h, gamma):
    result = 3.14158 * R**2 * h * gamma
```

```
    result = result / 1000
    return result

print 'aluminum', weightCyl( 5, 100, 2.70)
print 'iron', weightCyl( 5, 100, 7.85)
print 'gold', weightCyl( 5, 100, 13.22)
```

```
aluminum 21.205665
iron 61.6535075
gold 103.829219
```

Что мы тут имеем? Размеры цилиндра задаются в сантиметрах, а удельный вес – в граммах на кубический сантиметр. Получившийся вес пересчитывается в килограммы. Наш цилиндр во всех трёх случаях имеет диаметр десять сантиметров и высоту один метр. А вот материал разный – алюминий, железо, золото. Если эту функцию будут вызывать часто, вполне может оказаться, что алюминий и тем более золото редко интересуют того, кто вызывает. Ему обычно нужно только железо, и его утомляет необходимость раз за разом указывать его, железа, плотность. У Питона есть решение проблемы.

```
def weightCyl ( R, h, gamma = 7.85):
    result = 3.14158 * R**2 * h * gamma
    result = result / 1000
    return result

print 'iron', weightCyl( 5, 100, 7.85)
print weightCyl( 5, 100)
```

```
iron 61.6535075
61.6535075
```

В заголовке функции мы добавили знак равенства и значение для последнего параметра. Это означает, что если мы его не напишем, то он по умолчанию будет считаться равным 7.85, то есть удельному весу железа. При этом вы имеете полное право указать его явно, и тогда он будет принят и учтён для расчета. Вдруг ваш цилиндр выточен из ореха грецкого? Кстати, понятия не имею, как он выглядит, но сосчитать можно:

```
print 'walnut', weightCyl( 5, 100, 0.64)
```

```
walnut 5.026528
```

Однако это не всё о параметрах по умолчанию. Насколько я разбираюсь в лабораторных испытаниях, там скорее всего взвешивают стандартные

цилиндры стандартного диаметра и стандартной высоты. Другими словами, радиус и высоту задавать не обязательно, а вот удельный вес – таки да. Возникает вполне естественное желание написать где-то так:

```
def weightCyl ( R = 5, h = 100, gamma):
    result = 3.14158 * R**2 * h * gamma
    result = result / 1000
    return result
```

За это мы немедленно получим по рукам. Так нельзя! И если мы совсем немного задумаемся, то немедленно поймём – почему. Каким, собственно, образом можно догадаться, какой именно параметр пропущен? Да, в нашем конкретном случае недостаёт в списке двух параметров и это наводит на определённые размышления. Но если не хватает только одного, то это может быть и первый, и второй, и последний. Поэтому, для устранения неоднозначностей, разрешается пропускать сколько угодно параметров, но при условии, что все они будут последними. Тщательно обдумайте.

Меняем параметры местами – выносим удельный вес на первое место:

```
def weightCyl ( gamma, R = 5, h = 100):
    result = 3.14158 * R**2 * h * gamma
    result = result / 1000
    return result

print 'tungsten', weightCyl(19.25)
print 'oak', weightCyl(0.7)

tungsten 151.1885375
oak 5.497765
```

В Питоне есть ещё возможность задавать параметры по именам, но поскольку мне это кажется совершенно бесполезным, то и говорить об этом я ничего не буду.

Второе. Без этого нельзя. Это называется *рекурсия*. Это нужно очень и очень редко, но в таких случаях без рекурсии уже никак не обойтись. Но хотя в работе это мало кому понадобится, ни один учебник программирования обойтись без рекурсии не может, и от каждого программиста ждут хотя бы теоретического знания, что это такое.

Рекурсия – это когда функция вызывает сама себя. Это называется *прямая рекурсия*. Есть ещё и *косвенная рекурсия*. Это когда функция A вызывает функцию B, а уже та вызывает функцию A. Сразу возникает действительно, кроме шуток, важный вопрос – а как же такая функция завершится? Конечно, если функция будет вызывать себя с теми же параметрами, с которыми её вызвали снаружи, то цепочка вызовов окажется бесконечной. Отсюда простой вывод – при каждом вызове хотя бы один из параметров должен меняться. Более того, обязательно должно быть такое значение параметра, при котором цепочка вызовов прерывается и возвращается конкретное значение.

Есть в обучении программированию старая традиция – обучать рекурсии на примере факториала – опять, да. Так и поступим. Напоминаю – опять, да, факториал обозначается восклицательным знаком и определяется так; $N!=1\times 2\times 3\times ...\times (N-1)\times N$, то есть $5!=1\times 2\times 3\times 4\times 5=120$. Если приглядеться, то прямо-таки бросается в глаза, что $5!=4!\times 5$. Поскольку $1!=1$, то напрашивается такая организация функции:

```
def Fuct( N ):
    if N > 1:
        result =  N * Fuct(N-1)
    else:
        result = 1
    return result
```

Вызов рекурсивной функции ничем не отличается от вызова обычной – снаружи все функции одинаковы. Проверьте. Коме того, напоминаю, что чисто по определению $0!=1$. Проверьте и это.

Понятно, что эта задача легко решается и без привлечения рекурсии. Случаев, когда рекурсия действительно необходима, мало, а большинство учебных задач носят искусственный характер. На то они и учебные.

Две задачи на рекурсию для самостоятельного решения. Первая – частая и практически встречающаяся. Вторая – редкая, но иногда очень практическая, в методах оптимизации.

Первая. Есть дерево каталогов, или директорий, или папок – как это называется в вашей операционной системе? Обойдите всё дерево и выведите имена всех файлов во всех каталогах. Более практично, однако, подсчитать общий размер всех файлов. На самом деле, тут есть нюанс –

даже пустой каталог занимает сколько-то байт – но этим можно пренебречь.

Вторая, нетрадиционная, задача. Есть целое число, не очень большое, иначе задача становится вычислительно сложной. Выдайте все варианты разбиения этого числа на слагаемые. То есть для числа 3 надо выдать:

```
3,0,0
0,3,0
0,0,3
2,1,0
2,0,1
```

и так далее.

Скучное – глобальные и локальные переменные

Продолжаем разговор. Это *не* является необходимым, но знать это и хорошо понимать это совершенно обязательно. Вот такой парадокс. Очень простая функция, то что называется *квадратный трёхчлен*.

На вступительном экзамене в академию Василий Иваныч получил задание: из квадратного трехчлена выделить полный квадрат. И вот он плачет, а саблю точит!
© анекдот, математически верный

Выделить трёхчлен можно, это вполне разумная постановка задачи. Но у нас всё проще, мы только хотим посчитать значение соответствующей функции: $f(x) = ax^2 + bx + c$. То есть на вход четыре параметра, на выходе значение трёхчлена. Очень просто.

```
def Trinomial( a, b, c, x):
    result = a*x**2 + b*x + c
    return result

f = Trinomial( 1,2,3, 1.0)
print 'f = ', f

6.0
```

Кстати, я только что узнал, что *trinomial* это *трёхчлен* на английском. Что до работы функции, то всё просто и ожидаемо. Поменяем код:

```
def Trinomial():
    result = a*x**2 + b*x + c
    return result

a = 1;
b = 2;
c = 3;
x = 1.0

f = Trinomial()
print 'f = ', f

6.0
```

Код изменился, а результат нет. Попутно мы впервые встретились с очень важным понятием – *глобальными переменными*. Если быть точным, так это называется в традиционных языках программирования, а в Питоне этот термин не очень употребителен. Можно сказать и по-другому. `a, b, c, x` – *внешние* переменные. Внешними они являются по отношению к нашей функции, потому что значения этим переменным присвоены *вне* функции. Когда внутри функции встречается имя переменной `A`, то немедленно требуется узнать, чему же она равна. Внутри функции никакой информации мы об этом не найдём, поэтому ищем снаружи – и находим. А там, снаружи функции, все эти четыре имени являются совершенно обычными переменными.

Продолжаем опыты.

```
def Trinomial():
    a = 3  #!!!
    b = 4  #!!!
    result = a*x**2 + b*x + c
    return result

a = 1;
b = 2;
c = 3;
x = 1.0

f = Trinomial()
print 'f = ', f

10.0
```

Я специально, для наглядности, скопировал весь код, хотя добавились только две строки, их я отметил комментариями. Если вы потрудитесь

самостоятельно подсчитать результат, то увидите, что действуют те значения переменных A и B, которые заданы внутри функции. Тех переменных, которые снаружи, изнутри не видно. Это так красиво и называется – *область видимости*, а переменные, которые внутри, называются *локальные переменные*. Локальные они, разумеется, только по отношению к внешнему миру, для самой функции это самые обычные переменные.

Задаю хитрый вопрос – а что, если написать вот так:

```
def Trinomial():
    a = a + 1  #!!!
    result = a*x**2 + b*x + c
    return result
```

Возникает соблазн предположить, что будет взято внешнее значение переменной A, равное единице, к нему будут добавлена ещё единица и в сумме получим два. Ответ неправильный. Будет вот это:

```
UnboundLocalError: local variable 'a' referenced before assignment
```

Нам как бы намекают, что переменная A не определена, другими словами, не имеет никакого значения. Питон следует такой хитрой логике – как только имя переменной A появилось слева от оператора присваивания, она, эта переменная, была создана и является локальной для этой функции. Следовательно, та переменная A, которая справа от оператора присваивания, это та же самая переменная, что и слева. Так чему же она равна? А ничему! Её значение – пока ещё – не определено. Как только вы это поймёте, всё это для вас покажется простым и естественным. Однако я очень упорный и занудный. Внесём совсем небольшое изменение в код:

```
def Trinomial():
    c = a + 1
    result = a*x**2 + b*x + c
    return result

5.0
```

Всё хорошо, никаких претензий к коду. Хотя нам хочется программировать и программировать, приходится немного подумать. Переменная C появилась слева от оператора присваивания, значит создана новая переменная внутри нашей функции. Для присваивания ей значения

используется переменная A, которой внутри функции нет. Значит её ищут снаружи – и находят. Таким вот образом оно и работает.

Как я уже говорил, если вы хотите всё усвоить быстро и легко, это раздел можно пропустить. Дальше будет ещё несколько скучных опытов.

Зададим себе простоя вопрос – можно ли изменить изнутри функции значения внешних переменных? Если вы уже знаете, каким будет ответ, то очень хорошо.

```
def Trinomial():
    a = 1000000
    b = 1100000
    result = a*x**2 + b*x + c
    return result

a = 1;
b = 2;
c = 3;
x = 1.0

print 'before ', a,b,c,x

f = Trinomial()
print 'f = ', f

before  1 2 3 1.0
f =  2100003.0
after  1 2 3 1.0
```

Значения переменных успешно меняются внутри функции, но снаружи остаются прежними. Оно и понятно. При попытке изменить значение внутри функции ничего не меняется, а возникает новая переменная – все изменения которой действуют только внутри этой функции.

Вы думаете, вам всё уже понятно? Как бы не так.

```
def Change():
    a = [4,5,6]
    b.append(99)

a = [1,2,3]
b = [101,102,103]

print 'before ', a, b
Change()
print 'after ', a, b
```

```
before  [1, 2, 3] [101, 102, 103]
after  [1, 2, 3] [101, 102, 103, 99]
```

Первый список не изменился. Второй – да. В случае первого списка можно рассуждать по аналогии с числовыми переменными. Имя списка появилось слева от оператора присваивания, значит это новая переменная, не имеющая никакого отношения к той, что во внешнем мире. Дальше надо напрячь весь интеллект головного мозга. Список по имени `B` изменился, это факт. Но переменная по имени `B` не изменилась! Если объяснять всерьёз и глубоко, то надо говорить о том, что всё, что есть в Питоне, является объектами. и долго рассказывать, что такое эти объекты. Если по-простому, то имя переменной *не* стоит слева от оператора присваивания. Мы вносим изменения, обратившись к методу `b.append`. Методы есть у каждого объекта. Объект остаётся прежним, но содержимое его может измениться.

Оглядевшись по сторонам, я заметил мусорную корзину, она же урна. Даже две мусорные корзины. Одну физическую, в углу, другую виртуальную, на экране. С виртуальной всё ясно и неинтересно, она является объектом во всех смыслах, для программиста это очевидно. Реальная корзина устроена много проще. Она чёрная и стоит в конкретном северо-западном углу. Назовём корзину *северо-западная*. В этом углу есть место только для одной корзины, так что северо-западная корзина может быть только одна. Я могу принести другую корзину, зелёную, и поставить её в северо-западный угол, заменив ею чёрную. Для двух там места нет. А вот набросать в урну я могу что угодно, в разумных пределах. Потом придёт уборщица и всё вытряхнет. Содержимое меняется, мусорная корзина остаётся. Вот и со списками то же самое. Обдумайте.

Четвёртое. Поговорим о похожем, только в отношении параметров. Сразу скажу, что там всё очень похоже на обычные переменные.

```
def simpleProc( n, s, L ):
    n = 1
    s = '1'
    L = [1]

nOut = 12345
sOut = '12345'
LOut = [1,2,3,4,5]
```

```
print 'before', nOut, sOut, LOut
simpleProc( nOut, sOut, LOut)
print 'after ', nOut, sOut, LOut

before 12345 12345 [1, 2, 3, 4, 5]
after  12345 12345 [1, 2, 3, 4, 5]
```

Попытки просто и без затей изменить значения параметров не работают. А вот обращение к методам списка и к функциям списков работает очень хорошо:

```
def simpleProc( n, s, L ):
    del L[2]
    L.append(6)

before 12345 12345 [1, 2, 3, 4, 5]
after  12345 12345 [1, 2, 4, 5, 6]
```

Какие выводы? Если вы только начинаете программировать, то для вас всё должно быть простым и понятным. Если вы уже знакомы с другими языками программирования, то очень многое на этом этапе может показаться странным и даже диким. Сейчас я всех помирю и даже дам очень полезные советы. Советы эти относятся не только к Питону, они относятся к программированию на любом языке, традиционном или не очень. Возможно где-то есть и *совсем* нетрадиционные языки, но я до них ещё не добрался. Некоторую надежду, впрочем, подаёт классический LISP.

Советы. Никогда. *Вы слышите меня, бандерлоги?* © Киплинг и Путин. Никогда не пытайтесь изменять значения внешних переменных, находясь внутри функции, даже если язык это позволяет. Если не позволяет, тем более не пытайтесь. Если возможны какие-то фокусы, как в Питоне, с вызовами методов списков, не пытайтесь тоже. И будете программировать долго и счастливо. Разумеется, если будете выполнять и другие советы, которые я вам дам.

Можно ли менять значения параметров функции? В традиционных языках программирования это делается сплошь и рядом. Причина – хорошо, если функция вычисляет синус и возвращает только одно число. Тоже хорошо, если функция вычисляет точку в N-мерном пространстве и возвращает список координат. Хуже, если функция вычисляет, определяет, находит что-то совершенно разнородное. Возникает желание вернуть всё это в виде нескольких параметров. Даже не задумываясь над тем, что в Питоне это

очень непросто – запомним, так делать не надо. Если так делать надо, значит что-то в программе не так.

Пятое. О вложенных функциях. Помните вычисление синуса? В процессе происходило обращение к функции вычисления факториала. Пример не совсем удачный, потому что функция для факториала имеет самостоятельную ценность, но зато обе функции уже написаны. Покажем на них. Те две функции, синус и факториал, находились в тексте на одном уровне. Начиналось описание первой функции, заканчивалось, за ней следовала вторая функция. Вложенная функции, как легко догадаться, вложена в другую функцию. В нашем случае вложенной функцией будет факториал.

```
def ourSin( x ):
    def Fuct( N ):
        result = 1
        for i in xrange( 2, N+1):
            result = result * i
        return result

    result = 0
    for N in xrange(1,20):
        result = result + (-1)**(N+1)*(x**(2*N-1))/Fuct(2*N-1)
    return result
```

Все модификации свелись к перемещению кода функции сверху вниз, в тело синуса и сдвигу вправо – в Питоне, напоминаю, это не эстетика, а суровая необходимость. Обращение к синусу ничем не изменилось, а вот к факториалу обратиться извне больше нельзя. Разумеется, вложенных функций может быть сколько угодно. Размещаться они могут на одном уровне, а могут и внутри друг друга. Дальше должно следовать занудное объяснение про области видимости. Пишем вот такое чисто для демонстрации концепции кода:

```
def first():
    def second():
        x = 1
        f = x + y + z
        print 'f = ', f
    x = 11
    y = 12
    second()

x = 101
y = 102
```

```
z = 103

first()
```

Что будет выведено? Правильно, 116. Переменная `x` во внутренней функции перекрывает переменную с тем же именем во внешней функции и вообще снаружи. Переменной `y` во внутренней функции нет, поэтому она берётся из внешней функции, но не из основного тела программы. А переменная `z` – её взять больше неоткуда, только из самого внешнего мира. Вы всё поняли? Очень хорошо, забудьте. Не надо обращаться из функции к внешним переменным. А для чего я всё это так аккуратно объяснял? А потому, что считается – нельзя стать программистом, не освоив всю эту дребедень.

Теперь вопрос, относящийся к технологиям программирования. Зачем вообще нужны вложенные функции? Первое – чтобы не загружать излишне мозг программиста. Зачем программисту думать о двух функциях, если можно думать только об одной? Если факториал находится внутри синуса, то думать можно и нужно только о синусе. Для конкретно факториала, как я уже сказал, это не совсем так, факториал полезен и сам по себе. Второе – это добавляет коду компактности и переносимости. Если вы вздумаете скопировать в другую программу функцию синуса, есть шанс, что вы потеряете по дороге функцию факториала. Если факториал внутри синуса, то потерять его уже не получится.

Но, в общем и целом, использование вложенных функций я одобряю только в самых крайних ситуациях.

Функции с функциями

В предыдущем разделе речь шла о функциях, внутри которых находятся другие функции. Тему обсудили и тема закрыта. Но взаимоотношения функций могут строиться и по-другому. Есть вариант, когда функция вызывает сама себя. Это называется рекурсия и это мы уже обсуждали. Но ещё остаётся случай, когда функция получает в качестве параметра другую функцию.

Решим совершенно реальную и практическую задачу. Она встречается достаточно часто. Есть функция – в математическом смысле слова. К примеру $f(x) = x^2 - 6x + 11$. Вопрос – где у неё минимум, то есть при

каком значении x функция принимает самое маленькое значение? Область математики, которая этим занимается, называется *нелинейное программирование*. Слово *программирование* здесь имеет примерно то же отношение к программированию на компьютере, что и математическая функция к питоновской, то есть почти никакое. О том, как искать минимум, написаны сотни книг и десятки тысяч статей, потому что это действительно нужно и действительно важно.

Для конкретно нашей функции минимум можно найти чисто математическим способом, без привлечения компьютера. Всего-то надо взять производную, приравнять её к нулю и решить уравнение. Получим ответ, что минимум достигается при $x = 3$ и равен двум. В реальной жизни всё так легко не решается, просто потому, что или функция не вполне аналитическая, или данные поступают в реальном времени от некоторого прибора – и функции-то никакой и нет. Вот исходя из этого, и решим нашу задачу.

Но, конечно, найти минимум для одной и только одной функции совершенно неинтересно. Мы напишем функцию, которая получает на вход какую-то другую функцию, к примеру наш квадратный трёхчлен или что угодно, и ищет её минимум. При написании любой серьёзной программы начинать следует с оформления её программного интерфейса – то есть кто кого вызывает и с какими параметрами.

```
def func( x):
    y = x**2 - 6*x + 11
    return y

def opt( f, a,b, eps):
    #???
    return x

a = -100.0;  b = +100.0
eps = 0.01
x = opt( func, a,b, eps)
print 'min x = ', x
```

На месте вопросительных знаков должен быть самый важный код поиска минимума. Теперь о том, чем поиск минимума математически отличается от поиска минимума практически. В математике хотя и не всегда, но часто минимум ищется для функции вообще, то есть везде, где она определена. На практике почти всегда минимум ищут на заданном конкретном

интервале, иначе не получается. Интервал задаётся строкой `a = -100.0; b = +100.0`. Обратите внимание, что в одной строке целых два оператора, разделённые точкой с запятой. И заметьте, что числа заданы в плавающем формате, с точкой. Иначе с ними обращались бы как с целыми и в результате последующего деления получались бы целые результаты, что нас совсем не устраивает.

Поиск минимума для функции одной переменной называется *одномерная оптимизация*. Мы используем метод, называемый *дихотомия*, по другому – деление надвое. Идея метода очень проста, впрочем как и всех других методов одномерной оптимизации. Это тот самый случай, когда описание метода словами длиннее его программного кода. В начале у нас есть интервал $[a,b]$ содержащий искомый минимум. Выбираем на интервале две точки x_1 и x_2. Выбрать одну точку можно многими способами, выбрать две точки это настоящее искусство – именно это определяет, каким именно методом мы пользуемся. То, как мы выбрали начальные точки, оказывает серьёзнейшее влияние на эффективность работы программы. Поскольку наша программа в промышленных целях применяться не будет, всё, что мы хотим, – чтобы эти две точки не совпадали и чтобы выполнялось условие $x_1 < x_2$.

Дальше мы смотрим, какая из точек больше – именно больше, хотя, напоминаю, мы ищем минимум. Пусть $f(x_1) \geq f(x_2)$. Возможное равенство значений здесь роли не играет. Поскольку функция *вогнутая* (математический термин), имеем $f(a) \geq f(x_1) \geq f(x_2)$. Совсем немного подумав, приходим к выводу, что на участке $[a,x_1]$ минимума быть не может, ну никак не может. Поэтому число a выбывает из соревнования за звание минимума и мы заменяем его на x_1. Если бы соотношение между x_1 и x_2 было бы обратным, из гонки вылетело число b и заменилось на x_2. Поздравляю, вы только что познакомились с одним из важнейших алгоритмов нелинейного программирования.

информация для математически одарённых
У нас функция одной переменной. Обычно приходится искать минимум для функции произвольного числа переменных. Ну как произвольного? Никакой, даже современный компьютер – доступный простому нефтянику – не потянет поиск для десяти переменных. Пять-шесть это реально. Тем не менее, поиск всегда многомерен. Однако, внутри большинства способов

оптимизации в конце концов обнаруживается одномерная оптимизация. Как сказал великий русско-еврейский комик Аркадий Райкин, о которым вы, возможно, и не слышали – *Внутри средневекового рыцаря наши опилки*. В математике оно часто так. А уж в программировании всегда.
И ещё, если функция линейная, то всё будет совсем по-другому. И занимается этим отдельная ветвь прикладной математики, под названием линейное программирование.
конец Информации

Приступаем к реализации. Наша будущая функция называется `opt`, это от импортного слова *optimization*, то есть *оптимизация*. На вход её передаются четыре параметра, первый из которых сам по себе функция, а два последующих – плавающие числа. Обратите внимание – это только мы знаем, что они по сути своей плавающие. По их внешнему виду в вызове функции об этом догадаться нельзя. Но вы-то, конечно, догадались по их именам, что это границы интервала, внутри которого мы будем искать минимум. Четвёртый параметр пока имеет немного смутную формулировку – это *точность*. А что такое точность, мы решим по ходу программирования. Пока ясно одно – чем меньше это значение, тем ближе мы должны быть к результату.

С чего мы начнём? Начнём мы с программирования вложенной функции – той самой, от программировании которых я совсем недавно предлагал отказаться. Эта вложенная функция будет выполнять необязательный, но очень полезный сервис – она будет показывать, как идёт наш процесс оптимизации и насколько мы приблизились к цели. Это не искусственное требование, любая реальная программа оптимизации должна выводить информацию такого рода. Иначе нетерпеливый пользователь может начать нервничать и бить копытом.

Вложенная функция будет вложена в основную функцию, в ту, которая и занимается оптимизацией и приобретает вот такой вид:

```
def opt( f, a,b, eps):
    def ShowProcess():
        stroka = 'SP.  ' + 'a = ' + str(a) + '  b = ' + str(b)
        print stroka
    #  что-то загадочное
    return x
```

Теперь я познакомлю вас с интересным понятием программирования – *псевдокод*. Это означает, что мы пишем программу как бы по-русски, но имея в голове, что чуть позже всё это будет переведено с русского языка на язык программирования. В нашем случае псевдокод выглядит так:

```
задать начальные приближения x1,x2 внутри интервала a,b
рассчитать значения функции для них

пока a-b > eps
   если функция(x1) >= функция(x2)
      a = x1
   иначе
   если функция(x1) <= функция(x2)
      a = x2
   пока конец

   задать новые начальные приближения x1,x2 внутри нового интервала a,b
   рассчитать новые значения функции для них
конец пока
```

Самый важный вопрос – а как именно выбрать начальные приближения? Приближения только называются начальными, на каждом следующем шаге оптимизации мы должны ещё раз выбрать приближения по той же самой схеме. Общий принцип выбора – если не знаешь, как выбрать значения на интервале, дели поровну. Если надо выбрать одну точку – дели пополам. Если две точки, дели на три, и так далее. То есть для нашего случая, когда интервал $[-100,+100]$, первые приближения выбираются как $-33.33,+33.33$. Теперь результат:

```
def opt( f, a,b, eps):
    def ShowProcess():
       print 'x1 = ', x1, '  x2 = ' , x2
       stroka = 'SP.  ' + 'a = ' + str(a) + '  b = ' + str(b)
       print stroka
       print f(a), f(x1), f(x2), f(b)
       print

    x1 = a + (b-a)/3
    x2 = a + ((b-a)/3)*2

    fa = f(a);    fb = f(b)
    fx1 = f(x1); fx2 = f(x2)

    ShowProcess()

    while ( abs(a-b) > eps):
        if fx1 >= fx2:
            a=x1;
```

```
        else:  # if fx1 <= fx2:
            b=x2;

        x1 = a + (b-a)/3
        x2 = a + ((b-a)/3)*2

        fa = f(a);     fb = f(b)
        fx1 = f(x1);   fx2 = f(x2)

        ShowProcess()

    x = a + (b-a)/2
    return x
```

Теперь задача для самостоятельного решения. Узнайте, какие ещё бывают методы одномерной оптимизации, выберите по вкусу и реализуйте. Хотя бы метод золотого сечения.

Функция, у которой много параметров

Много, это не когда миллион. Много это когда заранее не известно сколько. Можно их, конечно, собрать сначала в список и передать как один параметр, но можно и так, без списка. Вы знаете, что такое среднее арифметическое? Правильно, это когда всё сложить и поделить. Так вот это мы программировать не будем. Мы будем программировать *среднее геометрическое* $g(x_1, x_2 ... x_n) = \sqrt[n]{x_1 \times x_2 \times ... \times x_n}$. Предполагается, что все числа должны быть положительными, то есть строго больше нуля. А почему? Объясните. Вот программный код, заметьте в нём что-то новое:

```
def G( *xs):
    result = 1
    for x in xs:
        result = result * x
    result = result ** (1.0/len(xs))
    return result

print 'G = ', G( 1,2,3 )

G =  1.81712059283
```

Первое, что бросается в глаза, бросается именно в первой строке. А именно – звёздочка перед именем единственного параметра. Она означает, что внутри функции с этим параметром можно обращаться как со списком. Не совсем, как со списком. Точнее, совсем не как со списком. Этот как бы список можно только читать, но менять его нельзя. Впрочем, менять нам

его и не надо, чтения достаточно. Если вам цикл кажется неочевидным, можно его заменить на другой, подлиннее и традиционнее:

```
for i in xrange( 0, len(xs)):
    result = result * xs[i]
```

Обратите внимание, что начальное значения для произведения равно единице. И ещё больше обратите внимание на число 1.0 – если написать просто 1, ответ будет неверным. Особенность Питона. Если хотите плавающий, в смысле дробный, ответ – используйте в вычислениях плавающие значения.

И, разумеется, параметры не обязаны быть числами.

И опять. Квадратное уравнение

Приобретя новые знания, всегда полезно применить их к старой задаче.

Автор с пером в руке перечитал книгу, написанную свыше тридцати лет назад. Вмешаться в произведение такой давности не легче, чем вторично вступить в один и тот же ручей. Тем не менее можно пройти по его обмелевшему руслу, слушая скрежет гальки под ногами и без опаски заглядывая в омуты, откуда ушла вода.
© Леонид Леонов, предисловие к роману *Вор*

Задача уже изучена по существу. Можно думать о методах и форме. Оформим решение квадратного уравнения в виде функции. Первый и главный вопрос при написании любой функции – не что у неё будет внутри. Главный вопрос – какой у неё будет интерфейс, что будет на входе и что будет на выходе. Для функции, используемой в реальном приложении, это во многом диктуется особенностями кода, из которого эта функция будет вызываться. Наша функция пишется в чисто учебных целях, поэтому всё намного проще.

На входе – просто три коэффициента. С выходом сложнее. Функция должна возвращать значение – ни, или, по крайней мере, я так считаю. У нас уже есть программа, помещающая найденные корни в список. Очень хорошо, его и будем возвращать. Можно бы этим и ограничиться. Но! Если в списке два элемента – корней два, если один элемент – то корень один. А если список пустой, то что? Это у нас корней нет или это у нас тождество?

Поэтому нельзя отказываться от текстового описания результата. Предлагаю добавить его в список последним элементом. Охотно допускаю, что в реальной программе это оказалось бы неудобным, но у нас пока не реальная программа. Имеем вот такую функцию, в первом приближении:

```
def QuaEq(A,B,C):
    roots = []
    result = 'так, на всякий случай'
    if A <> 0:
        dis = B**2 - 4*A*C;

        if dis > 0:
            x1 = (-B+dis**0.5)/(2*A);
            x2 = (-B-dis**0.5)/(2*A);
            result = 'два корня'
        elif dis == 0:
            x1 = (-B)/(2*A); x2 = x1
            result = 'один корень'
        else:
            x1 = 0; x2 = 0
        roots.append(x1)
        roots.append(x2)
    elif B <> 0:
        x1 = -C/B
        result = 'один корень'
        roots.append(x1)
    else:
        if C == 0:
            result  = 'тождество'
        else:
            result = 'корней нет'

    roots.append(result)
    return roots
```

Имя функции представляет собой сокращение от слов *quadratic equation* – квадратное уравнение. Обратите ещё раз внимание, что в списке могут храниться элементы произвольных типов, что для нас очень удобно. Теперь о том, как эту функцию вызвать и как воспользоваться её результатами.

```
roots = QuaEq(3,10,3)

print roots[len(roots)-1]
for i in xrange(0,len(roots)-1):
    print roots[i]

два корня
```

```
-0.333333333333
-3.0
```

Совпадение имени `roots` внутри функции и имени `roots` снаружи, разумеется, совершенно случайно. Теперь о важном. Для *этих* параметров вызова всё работает хорошо. А вот для других вылезает старая проблема с целыми типами.

```
roots = QuaEq(0,5,3)
один корень
-1
```

Пора с этим что-то сделать. Почему именно сейчас? Потому что наше решение уравнения оформлено в виде функции. Весь смысл функции в том и заключается, что её можно и нужно использовать многократно. А значит, наш скорбный труд не пропадёт.

Предлагаю решить проблему самым незатейливым способом. Поскольку мы не хотим задавать коэффициентам плавающий тип *снаружи* функции, зададим *внутри*. Немного изменим самое начало программного кода. Вместо

```
def QuaEq(A,B,C):
    roots = []
```

напишем

```
def QuaEq(A,B,C):
    A = float(A);  B = float(B);  C = float(C)
    roots = []
```

Что мы сделали? Мы насильственно указали, что в переменных `A,B,C` хранятся плавающие значения. Естественно, операция деления после этого выдала требуемый нам плавающий – и точный – результат. Естественно, тип переменной от этого не изменился – потому что у переменной типа нет. Не могу удержаться от очередной очень уместной цитаты:

Потому что хочу в уборную,
А уборных в России нет
© Сергей Есенин «*Страна негодяев*»

Потому что с типами было бы проще, но типов в Питоне нет. И почему я так подробно на этом останавливаюсь уже в четвёртой по счёту главе?

Потому что это очень важно для понимания Питона. У переменных типа нет. Переменная – это просто ящик, в котором что-то лежит. А вот у этого *что-то* тип очень даже есть. Обдумайте.

Попутно мы решили ещё одну проблему, сами того не заметив. Что было бы, пока мы не добавили эту строку, при таком вызове:

```
roots = QuaEq( '3', '10', '3')
```

Правильно, сообщение об ошибке во время исполнения программы. А теперь всё отлично работает – наша добавленная строка кода преобразует строковые значения в плавающие. Преобразует потому, что их *можно* преобразовать.

Дальше хуже. Типов нет. На вход функции можно задать что угодно, лишь бы параметров было три. Например, вот такое:

```
roots = QuaEq( 'А почему у тебя такие большие уши?',
               'А почему у тебя такие большие глаза',
               'А потому что какаю')
```

Так не пройдёт – вылетает на последней строке, что характерно. Конечно, вы совершенно точно помните, какого типа параметры нужно задать на вход вашей функции и никогда не ошибётесь. Но надо предполагать с самого начала, что пользоваться нашей функцией будем не только мы, но и какие-то другие неприятные нам люди. И эти люди совершенно обязательно зададут на вход нашей функции совершенно неуместные и нелепые параметры.

Однако напоминаю, тип переменной может быть проверен.

```
if (type(A) <> int) and (type(A) <> float):
    print 'Всё пропало'
```

Здесь мы только выводим паническое сообщение. Добавьте проверку типов двух других входных параметров. Оформите всю проверку в виде функции, возвращающей булевский результат. Вызовите *эту* функцию в начале *той* функции и примените условный оператор. Пока всё.

Глава восьмая, короткая. Модули. Коротко

Коротко не потому, что неважно. Модули – это очень важно. Но в Питоне идея модулей реализована осень просто. Разумеется, как и всегда, есть хитрые навороты, но сама основа настолько проста, что проще просто быть не может.

Постановка задачи

Прежде чем решить проблему, надо её найти, наступить на грабли и осознать. Для начала напишем невероятно простую, но вполне практически применимую функцию. Она будет вычислять площадь прямоугольного треугольника по катетам. На всякий случай напоминаю – стороны треугольника традиционно обозначаются как $a.b.c$, а углы как α,β,δ. Что важно, против угла α находится сторона a, против угла β – сторона b, а против стороны c – почему-то угол δ. В прямоугольном треугольнике гипотенуза обозначается как c, а катеты a,b.

Формула вот: $s=\frac{ab}{2}$. Я обещал, что будет очень просто. А теперь функция:

```
def S_1(a,b):
    return (a*b)/2
```

Почему функция называется `S_1` – потому, что будет `S_2`. А почему без фантазии и однообразно?

– Дорогая, ну придумай что-нибудь, ты же у меня такая фантазёрка!
© Старый, но смешной анекдот

Формулы для вычисления этой же площади есть и другие, уже с применением тригонометрии. Вот две. Первая вычисляет площадь по катету и углу. Вторая – по гипотенузе и углу. Угол, понятное дело, не прямой.

$$s=\frac{a^2 tg\beta}{2}$$

$$s = \frac{c^2 \sin\alpha \cos\alpha}{2}$$

Функции для вычисления чуть сложнее, но не намного.

```
import math

def S_2(a,ugol):
    return ((a**2)* math.tan(ugol))/2

def S_3(c,ugol):
    return (c**2*math.sin(ugol)*math.cos(ugol))/2
```

Обратите внимание на необходимость импорта математического модуля. Углы надо задавать в радианах. Если угол у нас в градусах, надо при вызове перевести его в радианы. Вот эти три вызова трёх функций вычисляют площадь одного и того же треугольника, ну или почти того же, потому что углы я округлил до целых градусов. Это Пифагоров треугольник, со сторонами (3.4,5).

```
s = S_1(3,4)
s = S_2(3,math.radians(53))
s = S_3(5,math.radians(53))
```

Хотя функции очень простые, но очень полезные и, скорее всего, могут понадобиться ещё и ещё раз. Можно, разумеется, прикопать исходный текст, а потом, когда очень понадобится, извлекать его из рукава и вставлять в нужное место, но мне это кажется каким-то неправильным. Ведь когда в том коде, что вверху, мы вычисляем синус, мы не вставляем чей-то грязный программный код в наш белоснежный. Мы просто пишем `import math` и после этого получаем доступ к синусам и прочему богатству. Хотелось бы и нам так же.

Маленькое замечание. Лично моё мнение – назначение модулей заключается почти исключительно в том, чтобы, написав что-то хорошее и полезное, иметь возможность использовать это ещё и ещё раз. У других авторов могут быть другие мнения.

Решение задачи

Так как сделать из наших трёх функций полноценный модуль для постоянного использования? Да почти никак! Не в смысле, что это почти

невозможно. В смысле, что это не требует почти никаких усилий. Мы берём вот этот код

```
import math

def S_1(a,b):
    return (a*b)/2

def S_2(a,ugol):
    return ((a**2)* math.tan(ugol))/2

def S_3(c,ugol):
    return (c**2*math.sin(ugol)*math.cos(ugol))/2
```

и сохраняем его в файл под именем `triMod.py`. `tri` – это не в том смысле, что у нас только три функции, это сокращение от слова *triangle*. Всё. Модуль готов. Поверьте, в других языках программирования создание модуля требует заметно бóльших усилий. Теперь о том, как нам его использовать. Это чуть сложнее. Вот три вызова трёх функций:

```
import triMod
import math

s = triMod.S_1(3,4)
print s
s = triMod.S_2(3,math.radians(53))
print s
s = triMod.S_3(5,math.radians(53))
print s
```

Здесь уже есть на что обратить внимание. Появилась строка **import** `triMod`. Перед именами вызываемых функций появилось уточнение, к какому модулю они принадлежат. Это ожидаемо и понятно. Теперь о важном. Как видите, всё получилось быстро и очень просто. Но, как всегда, есть нюансы.

Первый, безобидный. Наш модуль ссылается, в свою очередь, на модуль `math`. Это понятно, ведь нам нужен синус и прочие тангенсы. Головная программа тоже ссылается на тот же модуль `math`. Так вот, она импортирует его не потому, что он используется в импортируемом ею модуле `triMod`, а потому, что она, головная программа, использует его сама – в процессе перевода градусов в радианы. Говоря по-другому, если модуль `A` импортирует модуль `B`, а модуль `B`, в свою очередь, импортирует модуль `C`, то модуль `A` вовсе не обязан импортировать модуль `C`. Конкретно

в нашем случае это значит, что если бы нам не нужно было вызывать преобразование в радианы, то строку `import math` в головной программе можно было бы удалить. Это хорошая новость, потому что упрощает процесс программирования.

Теперь плохая. У меня на диске есть каталог/директория/как вам угодно. Имя у него что-то вроде `D:\Мои гениальные книги \ Гениальные книги по программированию \ конкретно Питон \ sources`. Ну, вы поняли, я человек очень скромный. Так вот, все исходные тексты у меня хранятся в этом каталоге. Среди них головная программа под именем `main.py` и наш модуль `triMod.py`.

```
# запоздалое разъяснение
```
Почему *головная программа*? Головная программа – это та штука, которая вызывает всех, а её никто не вызывает, кроме пользователя.
```
*Мужчины любят женщин*
*Женщины любят детей*
*Дети любят хомячков*
*И только хомячки никого не любят*
# конец Запоздалого
```

И всё отлично работает. На самом деле, всё гораздо сложнее и работает всё отлично только потому, что оба текста программ размещены в одном каталоге. А если нет? В Питоне реализован довольно сложный алгоритм поиска подключаемого модуля. Для учебных задач, вроде нашей, это неактуально. В реальной жизни, скорее всего, используемые вами – и всеми – модули будут лежать в каком-то одном каталоге. Программы, использующие эти модули, будут наверняка храниться в совсем других каталогах. Но это, как принято выражаться, – *далеко выходит за рамки обсуждаемых в нашей книге тем*.

Что ещё важно знать

Разрозненные замечания о модулях. Здесь о том, без чего можно и обойтись, но лучше всё же знать.

Первое замечание. У нас программа импортирует написанный нами модуль, который импортирует модуль `math`, который мы не писали, и исходный текст, которого даже не видали. Вполне возможно, что мы его и

не могли видеть вообще, потому что он написан на совсем другом языке программирования. Многие математические модули написаны на C или C++, так они быстрее работают. Разумеется, написанный нами модуль с тем же успехом мог бы ссылаться и на ещё один опять же написанный нами модуль и так далее. Это тривиально, но об этом обязательно надо сказать.

Второе замечание. Теперь о ситуации, которая может показаться вам искусственной и даже более того, не побоюсь этого слова, высосанной – из пальца. Это не так, проблема очень реальная, частая и встречается во всех языках, где возможны модули. И во всех языках с этим возникают проблемы. В Питоне с этим очень даже неплохо, но лучше разобраться с этим сразу. Ситуация эта называется *циклическая ссылка*. Что это такое ?

Есть программа, не важно, в каком файле, и есть два модуля, в файлах `A.py` и `B.py`. Первый модуль:

```
import B

def doSomething_A():
    print 'Ok'

def do():
    B.doSomething_B()
```

Второй модуль:

```
import A

def doSomething_B():
    print '''I'm here'''
    A.doSomething_A()
```

Вот сама программа, очень короткая:

```
import A
A.do()
```

Что происходит? Или, точнее, что, как мы ожидаем, должно происходить? Программа вызывает свою собственную функцию `A.do()` . Та вызывает из модуля `B` функцию `B.doSomething()`. Та что-то выводит и, в свою очередь, вызывает импортированную из `A` функцию `A.doSomething()`. Законно ли это? А сначала попытайтесь ответить, что же выведет программа. Правильный ответ:

```
I'm here
Ok
```

Если вы именно этого и ожидали, то это очень хороший результат, я не шучу. Так делать можно. Мне не нравится только то, что в Питоне делать это очень легко, даже слишком легко. В других языках, чтобы два модуля обращались друг к другу, требуется приложить определённые усилия и соблюдать достаточно строгие правила. Это заставляет предварительно задуматься. А задуматься надо, потому что последствия неаккуратного вызова могут быть очень нехорошими и причину найти будет очень трудно. В Питоне всё легко и просто и думать не надо.

Третье замечание. Кроме волшебного слова **import**, есть ещё слово **from**. Разница в том, что **from** извлекает из модуля не абсолютно всё, а только то, что мы конкретно закажем. Минус в том, что так будет однозначно длиннее. Плюс в том, что перед именами функций можно не указывать через точку имена модулей. Я считаю, что на данном этапе нашего развития возможность эта для нас лишняя.

Четвёртое замечание. Можно импортировать не только функции. Вот пример импорта проинициализированной переменной. Программируем очень короткий модуль по имени `mathConsts` в файле `mathConsts.py`. Напоминаю, имя модуля совпадает с именем файла.

```
pi = 3.14158
```

А вот его использование:

```
import mathConsts

def SofCircle(R):
    return mathConsts.pi*R**2

print 'S = ', SofCircle(2)
```

Пятое замечание. Теперь я научу вас плохому, но вы должны это немедленно забыть. Программа может менять значения переменных в импортированных модулях, вот так:

```
mathConsts.pi = 3
print 'наше пи самое лучшее пи в мире и равно ', mathConsts.pi
наше пи самое лучшее пи в мире и равно 3
```

Не делайте так, пожалуйста.

показ эрудиции
В 1897 году американский штат Индиана принял закон о том, что $\pi = 3.2$. То есть теперь-то они утверждают, что закон был принят только палатой представителей, а после отклонён сенатом, но мы-то знаем. Голосом Задорнова – *ну тупыыые...*
конец Показа

Шестое замечание. А если очень постараться, то можно написать модуль и на другом языке программирования, то есть не на Питоне, и подключить его к программе на Питоне, но об этом даже не в следующей главе, а в следующей книге.

Наше любимое квадратное уравнение

Украинское радио.
- Так, - говорит DJ, - а от прийшов лист від Петрика з села Залупівки. Петрик просить передати пісню про комбайн. Добре, Петрику, слухай пісню про комбайн.
Второй выход.
- О, знову лист від Петрика з села Залупівки, Петрик знову просить передати пісню про комбайн. Добре, Петрику, слухай пісню про комбайн.
Третий.
- А ось Петрик з села Залупівки просить передати композицію Twenty first century shizoid man з першого альбому групи "Кінг Крімзон"... Петрику, не вы%:;ся, слухай пісню про комбайн...
© анекдот, конечно

Вот и с нами то же самое – всё равно вернёмся к любимому квадратному уравнению. То есть оформим нашу функцию его решения в виде отдельных модулей. Она состоит из отдельных случаев, а, как в другом анекдоте, *случаи бывают разные*. Разнесём разные случаи по разным модулям. Кроме того, у нас есть и текстовые сообщения, неплохо и их отправить в отдельный модуль. Очень может быть, вы зададитесь вечным вопросом из третьего анекдота – *к чем эти нелепые телодвижения*? На самом деле, то что мы сейчас программируем – самая обычная большая программа. Только

маленькая. Да, в нашем случае её можно бы и не делить на модули. Но если вырастить кабанчика в десять раз толще, то придётся. И разделка туши будет производиться именно по тем же линиям, что и у нас.

Как учат нас теоретики программирования – прочитайте их, пожалуйста, наконец – есть два способа проектирования и программирования большого проекта. Один называется *сверху вниз*, другой, что как-то даже и неожиданно, *снизу вверх*. Первый случай выглядит так. У нас есть программа. Программа импортирует модуль `a` и вызывает функции из него. Модуль `a`, в свою очередь, обращается к модулю `b`. С кого начинать программирование? Согласно концепции *сверху вниз*, сначала мы программируем программу, затем модуль `a` и только потом модуль `b`. В случае концепции *снизу вверх*, всё строго наоборот.

Вам кажется, что всё это странно и глупо, и вообще ни о чём? Для маленькой, игрушечной, учебной программы – да, всё это ни о чём. Для программы реальной, той, которая за деньги, вопрос становится важнейшим. К тому времени, как специалиста допускают к разработке программ за деньги, он осознаёт одну очень важную вещь. Программировать – легко, проектировать – сложно. Правильно спроектированную программу можно отдать программировать программисту третьего сорта. Он справится, если за ним присматривать и периодически подвергать публичной порке. Такова суровая жизнь.

Главное в работе руководителя – подборка исполнителей и проверка исполнения © Черчилль? Бисмарк? Хемингуэй? Лейба Троцкий? Точно знаю, что не Пушкин

Это не шутки, это серьёзно. Во времена, когда я только начинал программировать, выбор однозначно делался в пользу подхода *сверху вниз*. Но когда появилось ООП – Объектно Ориентированное Программирование – то эти два подхода стали, по меньшей мере, равноценными. До ООП мы ещё не добрались, поэтому выберем консервативный вариант *сверху вниз*. В нашем случае, это значит, что начнём мы с головной программы. Но даже оголтелые фанаты этого подхода согласны, что некоторые модули нижнего уровня лучше запрограммировать в самом начале. В нашем случае это модуль со строками – текстовыми описаниями результата.

Хотя мы начинаем программировать с головной программы, уже на этом этапе вы должны определить интерфейсы всех трёх подчинённых модулей. Поскольку в модуле с константами комментировать будет явно почти нечего, начнём с него:

```
# -*- coding: cp1251 -*-
sqTwo = 'Два корня'
sqOne = 'Один корень'
sqNone = 'Корней нет'
sqIdent = 'Тождество'
sqStrange = 'Что-то странное'
sqRoots = 'А теперь корни'
```

Напоминаю только, что загадочная первая строка отвечает за правильный вывод русских букв на экран. В программном коде они будут видны и так, а при выполнении программы – кто его знает. Впрочем, Питон растёт и развивается, и вполне возможно, в новой версии всё будет немного не так, в лучшую сторону, конечно.

Теперь головная программа, она получается не такой уж и простой. Дело в том, что в стандартной программе всегда присутствуют две стандартные операции – ввод и вывод. Если программа очень серьёзная и очень большая, то их обязательно оформляют в виде отдельных модулей. И при этом опять-таки обязательно надо позаботится об обмене данными. В нашем случае функция ввода должна вернуть коэффициенты уравнения, а функция вывода получить корни уравнения и текстовое сообщение. Но раз наша программа чисто учебная, ограничимся оформлением ввода и вывода в виде локальных, вложенных функций. Причём функция ввода будет просто присваивать значения коэффициентам.

```
import sqSq
import sqLineary
import sqIdent
import sqConst

def Input():
    global A,B,C
    A = 3;  B = 10;  C = 3

def Output():
    print 'A = ',A, '  B = ', B, '  C = ',C
    if roots <> []:
        print roots[len(roots)-1]
    if len(roots) >= 2:
        print sqConst.sqRoots
```

```
    for i in xrange(0,len(roots)-1):
        print roots[i]

A = 0;  B = 0;  C = 0
Input()

if A <> 0:
    roots = sqSq.Rez(A,B,C)
elif B <> 0:
    roots = sqLineary.Rez(B,C)
elif (C <> 0) or (C == 0):
    roots = sqIdent.Rez(C)
else:
    roots = [sqConst.sqStrange]

Output()
```

Даже только сама головная программа получилась немаленькая. Что у нас тут нового? Вот это:

```
def Input():
    global A,B,C
    A = 3;  B = 10;  C = 3
```

Без слова **global** проинициализированные переменные A,B,C так бы и остались известными только внутри функции, а внешний мир ничего о них бы не узнал. Передача этих величин в качестве параметров функции не помогла бы – внутрь они передались бы, но наружу не вернулись бы. Строка с **global** указывает, что эти переменные существуют во внешнем мире независимо от нашей функции, даже если до тела функции они ни разу не упоминались.

Можно сказать, что в определённом смысле переменные эти создаются внутри функции, но живут снаружи. Обратите внимание на тонкий момент – если сразу после описания функции, то есть того программного кода, что написан чуть выше, мы добавим что-то вроде **print** A,B,C то программа завершится аварийно. Причина в том, что это только объявление функции, которое само по себе не создаёт никаких переменных. Ни глобальных, ни локальных. Переменные создаются при вызове функции. Да, немного запутанно, однако это плата за другие преимущества Питона.

Предполагается, что в списке roots находятся корни – в количестве двух, одного или ни одного. За корнями последним элементом следует текстовое

описание результата. Теперь три модуля для трёх вариантов. Обратите внимание, что первые два вполне можно использовать и отдельно, к примеру для решения линейного уравнения. Третий, сам по себе, вряд ли кому-то понадобится. Сначала модуль для полноценного квадратного уравнения. Важно следующее – если головная программа может храниться в файле с любым именем, это никого не волнует, то внешний модуль должен иметь в точности то имя, которое указано в инструкции импорта.

```
import sqConst

def Rez(A,B,C):
    dis = B**2 - 4*A*C
    if dis > 0:
        x1 = (-B+dis**0.5)/(2*A)
        x2 = (-B-dis**0.5)/(2*A)
        roots = [x1,x2,sqConst.sqTwo]
    elif dis == 0:
        x = (-B)/(2*A)
        roots = [x,sqConst.sqOne]
    else:
        roots = [sqConst.sqNone]
    return roots
```

Этот модуль, в свою очередь, ссылается на модуль с текстовыми константами. Что главное в этом модуле? Правильно, не забыть последней строкой написать `return roots`. Вы ведь не забыли? А я сначала забыл и очень удивлялся результатам выполнения. Теперь случай линейного уравнения:

```
import sqConst

def Rez(B,C):
    B = float(B);  C = float(C)
    x = -C/B
    roots = [x, sqConst.sqOne]
    return roots
```

Всё очень просто, вот только понадобилась строка с явным приведением типов. Это опять-таки тот случай, когда недостатки Питона являются продолжением его достоинств. Нам не надо, как в традиционных языках, объявлять переменные и явно определять их тип. Мы можем в любой момент переобуться на лету и сменить тип переменной. Взамен за это Питон сам решает, каким образом ему обращаться с нашими переменными. Далее третий модуль, с двумя первыми нулевыми коэффициентами:

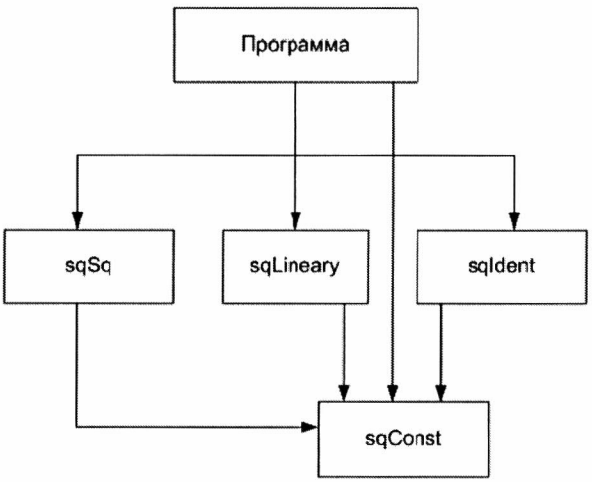
```
import sqConst
```

```
def Rez(C):
    if C == 0:
        roots = [sqConst.sqId]
    else:
        roots = [sqConst.sqNone]
    return roots
```

Интересно, что он ничуть не короче второго. Всё, что осталось, – проверить работоспособность программы в целом. Напоминаю, что пока у нас все пять файлов с исходными текстами должны лежать в одном каталоге. При запуске с прописанными насмерть коэффициентами (3,10,3) на выходе мы должны получить

```
A =  3   B =  10   C =  3
Два корня
А теперь корни
-0.333333333333
-3.0
```

Проверьте результат. А теперь нарисуем схему, кто в нашей программе к какому модулю обращается.



У нас очень простая организация программы и каждый модуль содержит только одну функцию В таких случаях обычно можно обойтись без схемы и держать всё в голове. По мере усложнения программы схема становится абсолютно необходимой.

Глава девятая. Файлы

Что такое файл, вообще

Материя – это объективная реальность, данная нам в ощущениях
© Вроде бы, В. И. Ленин, хотя он, вроде, немного другое говорил

Цитату поняли, про материю? Материю можно пощупать. Если щупать можно – материя. Нельзя щупать – однозначно не материя. Все понятия программирования, с которыми мы имели место до сих пор, пощупать, было нельзя. Максимум – на них можно было посмотреть на экране монитора. Файл пощупать можно, во всех отношениях. Чисто руками пощупать, конечно, получится не очень. Но файл, как область чего-то намагниченного на жёстком диске, или чего-то там с изменённым фазовым состоянием на CD или DVD прибором пощупать можно вполне. Файл – он есть.

С точки зрения программиста – файл остаётся после завершения программы, его можно аккуратно перенести на другой компьютер и даже прочитать другой программой. Что интересно, файл можно прочитать даже если другой компьютер работает под другой операционной системой – в нашем случае, не под Windows. Более того, программа, читающая файл, не обязана быть написана на том же языке, что и программа, его записавшая. К сожалению, это уже не всегда. Нет, при желании прочитать файл можно всегда – где угодно и чем угодно, но часто приходится применять несколько противоестественные способы. Не вполне поняли? Скоро поймёте.

Ещё очень важный момент – для того, чтобы прочитать файл, надо знать, как его записали. Если вам это кажется очевидным, поздравляю и усиливаю формулировку – чтобы знать, что прочитать из файла, надо знать, что в него записали. Странно? Скоро поймёте.

И ещё одна невероятно сложная концепция – есть файл как понятие, определенное в языке программирования. Есть файл как физическая сущность – на диске или на флешке. И то и другое может сочетаться самыми причудливыми способами.

Теперь конкретно. Во всех языках программирования и во всех операционных системах есть *текстовые файлы*. Иногда они могут

называться как-то не так, но они обязательно есть. В традиционных языках текстовые файлы обычно задвинуты куда-то на обочину. Если вы хотите получить файл, вы получаете просто файл. Чтобы получить текстовый файл надо предпринять некоторые дополнительные усилия. В Питоне текстовые файлы лежат в основе иерархии и если вы хотите просто файл – то получите именно текстовый файл. Давайте познакомимся.

Но! Если вы – вдруг! – решите пропустить раздел о текстовых файлах и перейти сразу к бинарным, ничего страшного не случится. Темы эти эти между собой, по сути, совершенно не связаны.

Шаг первый. Текстовые файлы. Теория

Я это уже писал, и не раз, в своих книгах. На всякий случай – *мои книги* это те, которые я написал, и не важно, под каким именем они изданы. Повторюсь, но в этот раз на более высоком и продвинутом уровне. Может быть, даже и очень для кого-то сложном.

Возможно, и я надеюсь, вам известно понятие *байт*. Если неизвестно, неважно, переходите к следующему абзацу. Байт – это восемь битов. Бит – единица измерения информации. Бит – двоичная цифра – или да, или нет. Байт может содержать целое число в диапазоне 0...255. Подумайте, почему. Если думать не хотите, посмотрите в приложении про биты и байты. Главное, что сейчас надо запомнить – не обязательно понять – то, что любой файл состоит из байтов. Вообще-то, в программировании из байтов состоит абсолютно всё, но к файлам это относится особо. О некоторых байтах говорят, что они *символы*. Текстовые файлы, – такие файлы которые содержат только символы. Ну или почти только. Теперь подробнее.

Символ, в первом приближении – буква. Или цифра. Или знак препинания. Или что ещё можно написать, чтобы это было видно. Впрочем, текстовые файлы могут содержать и кое-что ещё. Например, коды перевода строки и возврата каретки. Эти термины достались нам из древних времён пишущих машинок. Перевод строки означает, что мы перешли на следующую строку. Звучит глупо, не правда ли? Мы ведь просто нажали <Enter> и перешли на следующую строку. На самом деле, после того, как мы нажали клавишу, в текст был вставлен код перевода строки – и мы действительно перешли на следующую строку. Более того, одновременно в текст был добавлен ещё

один как бы символ – код возврата каретки. Он возвращает нас к началу строки, в которой мы находимся. Без него мы бы так и зависли в конце строки, куда перешли. Эти два символа – перевод строки и возврат каретки – всегда ходят парой, в Windows, по крайней мере.

И опять, я об этом уже говорил, файл Microsoft Word выглядит как текстовый, никаких сомнений, но на самом деле текстовым не является. Принципиальная особенность текстового файла – один символ на экране означает один символ или, что то же самое, один байт в файле. Если сразу это не очень понятно, обдумайте, поймёте потом.

Как убедиться, что файл текстовый? Очень просто, попытаться прочитать его как текстовый. Например, открыть Блокнотом из Windows. Если вы любознательный и вам хочется подробностей, то кое-что о файлах и байтах вы прочтёте в приложениях.

Вы песен хотите? Их есть у меня!
© одесское народное, затем к/ф Интервенция

Шаг второй. Текстовые файлы. Запись

В Питоне, если вы создаёте файл и не указали явно, какой именно файл вы хотите, то файл будет текстовым. Более того, вы уже к этому времени должны привыкнуть, что если вам хочется сделать что-то не совсем тривиальное, надо будет импортировать какой-то дополнительный модуль. Для текстовых файлов ничего импортировать не надо.

Вот самый простой пример, с комментариями. Главное, что надо понять с самого начала, – файл можно записать и файл можно прочитать. Способ записи и способ чтения очень похожи, но всё-таки разные. И если вы записали файл, то заранее думайте о том, что читать его может и кто-то другой, не обязательно вы. И читать, очень возможно, вам придётся файл, записанный кем-то другим. К счастью, для текстовых файлов это не вызывает почти никаких проблем. Вот пример:

```
f = open( 'd:/justfile.txt', 'w')

f.write( 'Вдруг из маминой из спальни');
f.write( 'Кривоногий и хромой');
f.write( 'Выбегает...');
```

```
f.close()
```

Самая первая строка сверху – это *открытие* файла. В обыденной жизни открыть можно только то, что уже существует. Если мы открываем банку тушёнки, то она у нас есть. Если мы хотим заготовить банку малинового варенья, то это называется как-то по другому. В программировании файл всегда открывают – и не важно, есть он или нет. Последняя строка – это, наоборот, *закрытие* файла. Любой открытый файл должен быть закрыт. А что, если нет? Если вы с этим файлом не собираетесь в этой же программе работать дальше, то ничего особенно плохого. А если собираетесь – то обязательно закройте.

Теперь, если мы заглянем на диск D:\, то обнаружим там новый файл по имени justfile.txt. С первым параметром функции **`open`** разобрались. Второй параметр чуть менее очевиден. Сначала обратите внимание, что он тоже в кавычках, то есть строка. У многих возникает желание записать его просто как есть, без кавычек. Так делать не надо. Именно это `'w'` означает, что файл создаётся заново. То есть если файла ещё нет, он будет создан. Если файл уже есть, он будут уничтожен и создан заново. В любом случае вся информация в файле, если она там была, исчезнет.

Теперь посмотрим внутрь файла, например Блокнотом. Увидим мы там не совсем ожидаемое:

```
Вдруг из маминой из спальниКривоногий и хромойВыбегает...
```

Всё смешалось в доме Облонских © Лев Николаевич и Аня
Это не бага, это фича
It's not bug, it's feature
© программистское, из моей молодости

Всё слиплось, короче. Это не ошибка, это особенность. Об этой особенности уже говорилось там, где говорилось о строках. Хотя мы три раза записали по одной строке, мы ни разу не указали, что это отдельные строки. Перечитайте то, что было сказано раньше, или просто запомните – для записи в текстовый файл отдельных текстовых строк пишите так:

```
f = open( 'd:/justfile.txt', 'w')

f.write( 'Вдруг из маминой из спальни\n');
f.write( 'Кривоногий и хромой\n');
```

```
f.write( 'Выбегает...\n');

f.close()
```

Я привёл весь код, а не только изменённые строки. Это только для начала, пока вы не заучили наизусть, что всякий файл должен быть сначала открыт, а потом закрыт. Теперь результат соответствует ожиданиям:

```
Вдруг из маминой из спальни
Кривоногий и хромой
Выбегает...
```

А что если мы вспомнили стих дальше и решили дописать продолжение в файл? Буква `'w'`, как вы наверняка догадались, – сокращение от слова *write*. Если взять слово *append* – которое, в частности, означает *прибавлять*, и сократить это слово, то получим букву `'a'`. Дальше открываем файл так:

```
f = open( 'd:/justfile.txt', 'a')
f.write( 'И качает головой\n');
f.close()
```

Если вы теперь прочитаете файл, то обнаружите в нём все четыре строки. Однако до сих пор мы читали наш файл только средствами Windows. А если у нас возникнет желание прочесть его самим, программно? Или, страшно сказать, если у нас возникнет желание прочитать файл, записанный в том же Блокноте?

Шаг третий. Текстовые файлы. Чтение

Что интересно, если записать текстовый файл можно, по сути, только одним способом, то для чтения есть выбор. Приступаем.

Сначала о плохом. Когда мы пишем строки в файл, мы заранее знаем, сколько строк мы туда запишем. Звучит банально. Но когда мы читаем строки из файла, мы не знаем, сколько их там – особенно, если файл писали не мы. Первый способ чтения выглядит *где-то* так:

```
f = open( 'd:/justfile.txt', 'r')

#  здесь читаем

f.close()
```

Что изменилось? Вместо `'w'` имеем теперь `'r'`. Это значит, что файл не создаётся с нуля, а читается. Или, специально для альтернативно одарённых, файл уже существует и, очень может быть, создали его не мы. Файл – ещё раз напоминаю, речь сейчас только о текстовом файле – можно прочитать разными способами.

Сначала мы прочитаем файл просто и без затей.

```
f = open( 'd:/justfile.txt', 'r')
s = f.read()
print s
f.close()

Вдруг из маминой из спальни
Кривоногий и хромой
Выбегает...
```

Что мы сделали? Мы прочитали весь файл в *одну* строку сразу. Когда мы эту строку вывели на экран, мы получили три строки. Почему? Потому, что символ перевода строки это такой же обычный символ, как буква или цифра и был сначала прочитан в строку, а потом выведен на экран. При его выводе на экран он сделал то единственное, на что способен, – перевел строку. Если для вас это пока не очень понятно – забудьте и примите как есть.

Хорошо ли это – такое чтение файла? Да, это хорошо, но если нас только интересует содержание файла. Чаще бывает, когда нам интереснее каждая строка файла в отдельности. Мы хотим её изучить, проанализировать и разобрать на символы. Кроме того, вполне возможен случай, когда текстовый файл слишком большой. Вот прямо сейчас передо мной лежит реальный текстовый файл объёмом 27 мегабайт, в котором 576957 строк. Это протокол некоторого процесса, завершившегося аварийно. Файл надо разобрать, изучить и проанализировать. Чтение такого файла в один приём может потребовать слишком много памяти или времени. И, главное, если мы прочитали файл сразу, а нас интересуют отдельные его строки, то разборка его на эти строки ложится на нас.

Сейчас мы будем читать файл построчно. Это значит, что за один раз мы читаем только одну строку. Если для вас это очевидно и вам обидно за такие мои разъяснения, то я за вас рад. Для того чтобы построчное чтение

имело хоть какую-то цель, сделаем самое простое, что только может быть – выведем для каждой строки её длину.

Вот первый, немного корявый, вариант:

```
f = open( 'd:/justfile.txt', 'r')
s = ' '
while s <> '':
    s = f.read()
    print s
f.close()
```

Почему он мне кажется корявым? Сейчас объясню. Но сначала заунывное уточнение. Вы ведь наверняка обратили внимание, что в операторе `s = ' '` у нас строка из одного пробела, а в операторе `while s <> '':` у нас пустая строка? Пустая строка – строка, не содержащая символов, в которой вообще ничего нет. Это не случайно. Наш цикл **while** работает до тех пор, *пока* – пока по-английски и есть while – пока выполняется условие, что прочитанная из файла строка не является пустой. Прочитанная строка будет пустой в том и только в том случае, когда файл банально кончился. Да, это не очень красиво, но это работает. А что мне здесь совсем не нравится, так это то, что признак конца файла – пустая строка – тоже выводится на экран.

Результат тоже не вполне желателен:

```
Вдруг из маминой из спальни

Кривоногий и хромой

Выбегает...
```

То есть между нашими строками появились и ещё какие-то пустые строки, которые мы туда не писали. Причина этого странного явления в том, что каждая прочитанная нами строка уже содержит код перевода строки. Оператор **print** тоже трудолюбиво переходит на новую строку. Результат налицо. Теперь выведем число символов в строке для первого и второго варианта чтения.

```
s = f.read()
print 'length(s) = ', len(s)
length(s) =  60
```

```
while s <> '':
    s = f.readline()
    print 'length(s) = ', len(s)
length(s) =  28
length(s) =  20
length(s) =  12
length(s) =  0
```

Что мы видим? Первое, мы убедились, что сначала весь файл действительно читался сразу, а теперь отдельно по строкам. Второе, мы опять-таки убедились, что в конце мы прочитали строку нулевой длины. Третье – символов в последней строке прочитано 12, а если считать глазами, то увидим только 11 символов. Это и есть тот самый невидимый код перевода строки.

Теперь избавимся от последней пустой строки. Можно так:

```
s = f.readline()

while s <> '':
   print s, '  length(s) =', len(s)
   s = f.readline()
```

Проверьте и убедитесь, что пустая строка с экрана пропала. Конечно, мы по-прежнему её читаем, но уже не обрабатываем. А вот это официальный вариант, который настоятельно рекомендуется во всех руководствах по Питону:

```
while True:
    s = f.readline()
    if s == '':
        break
    print s
```

Оператор **break**, напоминаю, производит немедленный выход из цикла. Выберите тот вариант, который вам приятнее. Теперь о том, как избавиться от назойливого перевода строки. Нет, в файле он необходим, но если мы прочитали файл построчно, то этот код уже как-то и ни к чему. А вот так:

```
s = f.readline()
s = s.rstrip()
```

Итак, с высоты птичьего полёта, есть два способа чтения текстового файла. Или мы читаем его одним оператором, но потом нам придётся повозиться, чтобы разобрать его по строкам. Или мы повозимся в начале, но прочитаем файл построчно. Есть и третий способ.

```
f = open( 'd:/justfile.txt', 'r')
L = f.readlines()
print L
```

В результате мы получим список, элементами которого являются отдельные строки файла. Присмотритесь к коду и заметьте, что вместо `readline()` появилось `readlines()`. Разница в одной букве, а результат совсем другой. Строки эти содержат код перевода строки. Хорошо это или плохо, зависит от задачи, которую вы решаете. Разумеется, если файл *очень* большой, такой метод может потребовать много памяти – и времени.

До сих пор мы писали в файл только строки, причём непосредственно. С тем же успехом можно писать и строковые переменные.

```
s1 = 'Once upon a midnight dreary, '
s2 = 'while I pondered, weak and weary,'

f.write(s1)
f.write(s2)
```

Зададим – себе – вопрос посложнее. Можно ли записать в файл числа или что-то более хитрое, например списки? И нужно ли их туда писать?

Любит ли слонопотам поросят? И как *он их любит?*
© Винни-пух в переводе Заходера

В текстовый файл можно *в принципе* записать всё. Только это всё мы должны при записи своими собственными руками преобразовать в строки. А при чтении опять-таки этими же руками преобразовать из строк в то, чем это было до записи. Мне кажется, это нам не нужно. В текстовый файл нужно писать текст. А если хочется чего-то другого, впереди у нас ещё много интересного о файлах.

Шаг четвёртый. Запись объектов в файл. Только для Питона

Это очень удобный и простой способ записи данных в файл. Даже более чем удобный и простой – идеальный. Но есть один нюанс. Теперь цитата, мне кажется, к месту. Для тех, кто в школе с русским языком не дружил – а будущие программисты обычно не дружат – поясняю. Мужик по фамилии Даль – автор *Толкового словаря живого великорусского словаря*. Типично русская фамилия не случайна. *Этимологический словарь русского языка* составил Макс Фасмер, а *Правила русского правописания* сочинил Розенталь. Тенденция, однако. Вот, собственно, цитата.

В 1837 году, встретившись в Уральске с Жуковским, Даль представил ему образец двоякого способа выражения: общепринятого книжного и народного. Фраза на книжном языке имела такой вид: «Казак седлал лошадь как можно поспешнее, взял товарища своего, у которого не было верховой лошади, к себе на круп и следовал за неприятелем, имея его всегда в виду, чтобы при благоприятных обстоятельствах на него напасть». На народном же — «Казак седлал уторопь, посадил бесконного товарища на забедры и следил неприятеля в назерку, чтобы при спопутности на него ударить». В ответ на характеристику В. И. Далем народного способа изложения как более короткого и выразительного В. А. Жуковский заметил, что вторым способом можно говорить только с казаками и притом о близких им предметах. © Из предисловия к шестому изданию словаря Даля

Смысл в том, что если вы пишете файлы этим способом, то его прочитает только такой же казак, как и вы – в смысле такой же питонист. Текстовый файл прочтут почти все, бинарный, о котором в следующем разделе, вообще абсолютно все, а вот этот – увы нет.

Создадим список, да позаковыристее:

```
L = [ 1,2, 3.141582, 'Blue oyster', True,
       [99,77, [], 'Your mother dont like me']]
print L
[1, 2, 3.141582, 'Blue oyster', True, [99, 77, [], 'Your mother dont like
me']]
```

Теперь наша задача – записать это безобразие в файл. Ну и прочитать после, само собой. Как ни странно, это совсем не сложно, хотя и придётся подключить дополнительный модуль.

```
import pickle

f = open( 'd:/something.strange', 'w')
pickle.dump(L,f)
f.close()
```

Результатом является появившийся там, где и надо, файл следующего содержания

```
(lp0
I1
aI2
aF3.141582
aS'Blue oyster'
p1
aI01
a(lp2
I99
aI77
a(lp3
aS'Your mother dont like me'
p4
aa.
```

Разумеется, я, если постараюсь, смогу объяснить вам что это, и почему оно выглядит и пахнет именно так. Но мне кажется, это лишнее. Главное понять, что никто и никогда прочитать это не сможет. Кроме нас, программистов на Питоне. А мы будем читать это следующим образом:

```
f = open( 'd:/something.strange', 'r')
L = pickle.load( f)
print 'out L = \n', L
f.close()

out L =
[1, 2, 3.141582, 'Blue oyster', True, [99, 77, [], 'Your mother dont like
me']]
```

Весь наш хитроумный список чудесным образом идеально прочтён. На что обратить внимание. Строка `import pickle`. Без неё никак. Мы обязательно должны подключить этот внешний модуль. Открытие и закрытие файла ничуть не изменилось. А вот для чтения и записи данных в файл

используются функции модуля `pickle`. И, в конечном итоге, файл всё равно получается текстовым.

Главное и ещё раз повторённое – вы можете легко и просто записывать абсолютно любые данные таким способом, но прочитать их сможете только в Питоне. Для внешнего мира эти ваши файлы – бессмысленный набор символов.

И ещё одно – процесс записи непонятно каких данных в текстовый файл по научному называется словом *сериализация*. Запомните, очень может пригодиться – не столько процесс, сколько красивое слово.

Глава десятая. Файлы бинарные

Бинарные файлы настолько важны, что появилась неотложная нужда выделить их в отдельную главу. Если вы прочитали всё предыдущее – всё предыдущее можно забыть. Шучу, конечно. Забывать ничего не надо, любые знания бесценны. Тем не менее, все эти файлы – это не совсем настоящие файлы. Настоящие файлы – те, в которые пишутся бинарные данные, как они есть. Скорее всего, такая формулировка вам непонятна – но я объясню. Если вы пропустили, по моему совету, предыдущие разделы и перешли сразу сюда, то вам будет даже проще.

Повторю ещё раз – бинарный файл абсолютно универсален, его можно совершенно однозначно прочесть в любой операционной системе и применяя любой язык программирования.

есть нюансы, конечно
В разных языках и на разных платформах разница, конечно, со временем вылезает. К примеру, есть минимум два формата записи плавающих чисел. Но к тому моменту, когда перед вами возникнут такие проблемы, вы уже будете знать, как с ними бороться.
конец Нюансов

Хотя я считаю, что бинарные данные – это главный, основной и базовый формат записи и чтения файлов, Питон считает по-другому. С точки зрения Питона, это формат вспомогательный и реализован соответственно, то есть не очень. И ещё – если вы будете читать книги по Питону и даже его официальную и полуофициальную документацию, то везде встретите утверждение, что этот метод записи предназначен для обмена данных с языками C и C++. Это, конечно, так – но не только. Это даёт возможность обмениваться данными с кем угодно и с чем угодно.

Запись и чтение бинарного файла

Решим простую и бессмысленную практически задачу, подходящую для учебных целей. Есть радиус круга – целое число. Считаем площадь круга – число, неизбежно, дробное. Записываем и то, и другое в бинарный файл. Затем читаем. Сначала запись, подробно и с комментариями:

```
R = 10
S = 3.1415 * R**2
```

```
print R, S

import struct

f = open( 'd:/square.bin', 'wb')

data = struct.pack('<i4',R)
f.write(data)

data = struct.pack('<f4',S)
f.write(data)

f.close()
```

Сначала обязательное **import** `struct` – подключаем внешний модуль. Имя файла, разумеется, совершенно произвольно. То, что расширение имени именно `*.bin`, никакой роли не играет. А вот параметр `'wb'` очень важен. Именно он указывает на то, что файл пишется на запись, что вам уже понятно, и что файл бинарный. Теперь о главном, прошу максимально сосредоточиться. Если вы поймёте следующие две строки, то вы поймёте всё о бинарных файлах, я не шучу, это серьёзно.

```
data = struct.pack('<i4',R)
f.write(data)
```

В первой строке имеем вызов функции из модуля `struct`, это понятно. Функция `pack`, что очевидно из её названия, *пакует*. У неё два параметра и возвращаемое значение. Начнём с самого простого – второй параметр по имени `R` – это то, что надо упаковать. Теперь немного сложнее. Возвращаемое значение `data` – это то, во что мы упаковали значение параметра `R`. Какой тип у переменной `data` и что в точности у неё внутри, нам совершенно не интересно. Она есть, и это всё. Пока и почти.

Теперь о самом важном, о первом параметре функции `'<i4'`. Выглядит, мягко говоря, невразумительно. Эта загогулина не единое целое, не какой-то иероглиф, а три совершенно отдельных независимых символа. Первый символ, который знак меньше, просто должен там быть. Запомните и всегда его пишите. Для любознательных будет разъяснение дальше. Символ `i` указывает, что мы хотим упаковать именно целое число. Цифра 4 означает, что мы хотим упаковать его в четыре байта. Почему именно в четыре, а что это значит и как можно ещё, вы можете прочитать в приложении. Там же всё рассказано и о байтах, на всякий случай. А теперь о самом главном – вот это самое число четыре и есть самое главное во всей

этой операции. Если мы позже захотим успешно прочитать наш файл, то должны сейчас запомнить это число наизусть. Это не какая-то особенность Питона. Это общий принцип работы с бинарными файлами, а 99% файлов в мире бинарные. Хотя, скорее всего, я неправ. 99.99%.

Следующая строка программного кода выглядит совершенно невинно. Она записывает упакованное целое число в файл. С точки зрения Питона, тут думать вообще не о чем. С точки зрения внешнего мира, надо подумать о том, что файл, который реально и физически останется на диске после выполнения нашей программы, стал больше на четыре байта. Следующие две строки выглядят очень похоже, отличие только в одной букве.

```
data = struct.pack('<f4',S)
f.write(data)
```

Это понятно, мы снова записываем число, только не целое, а плавающее. Буква `f` – сокращение от *float*. Цифра четыре снова подсказывает, что мы хотим записать плавающее число в четырехбайтовом формате, бывают и другие. В результате размер нашего файла увеличился на четыре байта и достиг восьми. Проверьте, пожалуйста.

Теперь прочитаем файл. Чтение очень похоже на запись – только всё наоборот.

```
f = open( 'd:/square.bin', 'rb')

data = f.read(4)
R = struct.unpack('<i4',data)
print 'R = ', R

data = f.read(4)
S = struct.unpack('<f4',data)
print 'S = ', S

f.close()
```

Сначала очень новый оператор: `data = f.read(4)`. Он означает, что мы читаем файл, но читаем из него не строку и не объект. Мы читаем просто четыре байта в переменную `data`. Что у этой переменной внутри, нас, опять-таки, не интересует. И ещё раз о самом главном – мы должны точно знать, что должны прочитать из файла именно *четыре* байта. Если раньше мы паковали целую переменную, то теперь мы её распаковываем:

```
R = struct.unpack('<i4',data)
```

Выглядит очень похоже на упаковку. И ещё раз о главном, извините, если уже надоел, а вы всё давно уже поняли – мы должны очень чётко помнить, что прочитанные нами строкой раньше четыре байта это именно целая переменная, а не плавающая, к примеру. Теперь интересная особенность чтения, особенность эта, мягко говоря, с первого раза немного пугает. Выведем то, что только что прочитали.

```
print 'R = ', R
R =  (10,)
```

Прочитанное число читается не просто так, а обязательно становится элементом кортежа. Кортеж из одного элемента, поэтому в хвосте нелепая запятая. Отказаться от этого – от кортежа – нельзя. Я не знаю, зачем и кому это нужно. Разумеется, ситуацию можно исправить, но только потом:

```
S = struct.unpack('<f4',data)
R_ = R[0]
print 'R_ = ', R_
R_ =  10
```

Об этой странности я больше напоминать не буду, а вы не забывайте. Теперь обещанные подробности о формате записи и что означает знак *меньше* и возможный на его месте знак *больше*.

ненужные подробности для любознательных
Вообще говоря, первый и не очень понятный параметр можно упростить, вот таким образом:

```
data = struct.pack('i',R)
R = struct.unpack('i', data)
```

Указывать тип совершенно необходимо, а вот длина в четыре байта берётся просто по умолчанию. Это хорошая, подходящая величина. С первым символом сложнее. По умолчанию на его место подставляется знак больше. Что это означает? Означает это, в каком порядке записываются четыре байта составляющие целое число.

Что означают знаки меньше и больше? Вы можете узнать подробнее о байтах и о внутреннем устройстве переменных в приложениях, здесь о

том, кто из них лучше. Целое или плавающее число занимает в нашем случае четыре байта. Есть младший байт. Есть старший. Знак меньше указывает, что сначала пишется младший байт, знак больше – что наоборот. В других, традиционных языках, если не извращаться специально, порядок записи – от младшего к старшему.

Записанный в Питоне по умолчанию, то есть с символом больше, файл отлично прочитается, при условии, если чтение будет абсолютно симметричным, то есть тоже с пропущенными символами по умолчанию. Есть только одна маленькая проблема – прочитается он нашей программой на Питоне и кем-то, возможно, ещё. Большинство сторонних программ наш файл не поймут.
Вывод – не умничайте и делайте в точности, как я сказал.
конец Ненужных подробностей

Продолжаем разговор. Сначала решим ещё одну учебную задачу, а после изучим, как пишутся в бинарный файл данные других типов. А потом решим ещё одну учебную задачу, но уже почти настоящую.

Учебная задача. Записать – и прочитать – случайное число случайных целых чисел. Зачем? Да какая разница! Несмотря на несколько искусственный характер, задача имеет и очень даже обычные практические особенности. Редко приходится читать файл, заранее зная, что в нём содержится ровно двадцать три целых числа. Гораздо чаще мы сначала читаем из файла число чисел в нём, в файле, записанных, а потом в цикле сами эти числа. Понятно. Что всё это организовано много и много сложнее, но идея та же, что и в нашей учебной задаче.

Но мы дополнительно приблизим нашу учебную задачу к реальной. Какая первая проблема встаёт перед нами при чтении любого нами записанного файла, структуру которого мы отлично знаем? Первая проблема – убедиться, что это действительно наш файл, а не подсунутая врагами дешёвая китайская подделка. Чтобы отличить наш файл от потустороннего, используется *сигнатура*. Это не какое-то сверхсложное понятие. Это всего-навсего последовательность символов, записанная в начале файла. Есть символы – значит наш файл. Нет – значит нет.

интересный факт

Если вы заглянете в любой исполняемый файл, тот, который с расширением *.exe, то обнаружите в начале некие буквы MZ. Это сокращение от имени и фамилии Миша Цукерман. Вот честное слово. И как теперь с этим жить?
конец Факта

Как записать и прочитать строку

Однако для реализации этой опции нам надо научиться записывать в файл третий, после целых и плавающих, тип данных – строки. С глубоким сожалением, должен заметить, что технология заметно отличается, и не в лучшую сторону. Вот запись:

```
name = 'circle'
data = struct.pack('6s',name)
f.write(data)
```

Мы просто решили напомнить читателю файла, что записали в него что-то, относящееся к кругу. Если вы заглянете в созданный файл, то увидите, что он увеличился ровно на шесть байт – что неудивительно – и что в этих шести байтах содержатся шесть букв слова circle. Это можно увидеть, посмотрев файл чем угодно вроде Total Commander или FAR. Со строками, однако, есть небольшая проблема. В отличие от целых и плавающих, их очень редко можно просто так взять и прочитать. Фактически мы записали не строку, а шесть символов, из которых она состоит. Наша книга не интересуется другими языками, только Питон, поэтому, как и всегда, всё, относящееся к другим языкам программирования, идёт в приложениях. Читайте приложение о том, как читаются бинарные файлы в других языках.

А пока сами, в Питоне, прочитаем то, что записали:

```
data = f.read(6)
name = struct.unpack('<6s',data)
print 'name = ', name
name =  ('circle',)
```

Всё банально и ожидаемо, главное помнить, что мы записали именно шесть байт. Но вот программа, написанная на другом языке, может этого и не понять. Впрочем, огорчаться не надо, что касается записи и чтения строк, эти самые не питонские программы обычно друг друга тоже не понимают.

Запись и чтение работают, но теперь нам приходится запоминать не только то, что мы пишем в файл строку, но и то, что её длина именно шесть символов. От запоминания того, что это именно строка, мы избавиться не сможем, но вот с шестью символами проще. Надо выполнить небольшую замену.

```
# было
data = struct.pack('6s',name)
f.write(data)

# стало
data = struct.pack('7p',name)
f.write(data)
```

Чтение тоже немного изменилось.

```
data = f.read(7)
name = struct.unpack('<7p',data)
```

подробности для любознательных
Буква *p* здесь является сокращением от слова *Pascal* – имя языка программирования. Именно там использовался такой формат записи строк. В прошедшем времени – потому, что это не тот Object Pascal, который в среде программирования Дельфи, а тот, который ещё Turbo Pascal. Там строки имели ограничение в 255 символов. Удивительно или неудивительно, но в Питоне для строк, записанных в этом режиме, действует то же самое ограничение.
конец Для

Обратите внимание, вместо шестёрки в программном коде я написал семёрку. Это вовсе не опечатка, так надо. И, точно так же, если до того у нас раньше шла буква, означающая формат, а только потом его длина в байтах, то теперь всё наоборот. Так надо, запомните.

Что будет с нашим новым файлом после такой записи? Файл увеличился на один байт, его длина теперь равна семи и в первом байте записана длина строки. В этой длине строки записано однако не семь, а шесть – потому что это и есть длина строки. Это надо твёрдо усвоить – длина строки при таком способе отличается на единицу, в меньшую сторону, от её размера в файле.

При записи в файл мы указываем, что пишем семь байт, или семь символов. как вам это больше понравится. Байт с длиной строки тоже учитывается. Читаем из файла опять-таки семь байт, потому что семь и записали, это понятно. А вот то, что при распаковке мы указываем тоже семь, мне кажется несколько нелогичным, но что имеем, то имеем. Это надо просто запомнить.

К чему, однако, все эти сложности? В чём награда? Теперь нам не надо помнить длину строки, нам надо только помнить что это именно строка. Её длина хранится в файле. Точно так же любая другая программа на совсем другом языке программирования, читающая наш файл, должна только знать, что её ожидает далее именно строка. И, ещё раз, с записью строк в файл ситуация просто безобразная во *всех* языках программирования. Ничего не поделать.

Теперь мы прочитаем это, но предполагая, что мы знаем только, что это строк,. но не знаем, сколько именно в этой строке символов. Программа будет короткой, но очень насыщенной новыми концепциями и понятиями.

```
data = f.read(1)
Ls = struct.unpack('<b1', data)
form = '<' + str(Ls[0]) + 's'
data = f.read(int(Ls[0]))
name = struct.unpack( form, data)
```

Комментариев для такого короткого текста потребуется много. Первая строка – читаем один байт. Вторая строка – распаковываем её в переменную. Переменная, само собой, оформляется в кортеж. Третья строка важнее и интереснее. Строку формата мы не задаём для последующей распаковки непосредственно, а формируем её, что называется, на лету. Для этого из кортежа мы извлекаем первый элемент – с нулевым индексом – и переводим его в строку. Четвёртой строкой мы читаем из файла число байт, которое, что важно, заранее не известно. Число байт мы принудительно устанавливаем целым числом, поскольку всем известно, что в Питоне типов нет.

Однако опять-таки всем известно, что в Питоне преобразования между строками и числами сами собой не случаются и их надо делать насильственно, своими собственными руками. Пятая строка, к счастью, совершенно обычна.

Учебная задача

Напоминаю структуру нашего файла:

Сигнатура. Пусть это будет строка из восьми символов 'Random '
Количество случайных чисел. Четырёхбайтовое целое
Сами случайные числа. Тоже четырёхбайтовые целые

Почему сигнатура имеет длину восемь байт, если из них заняты только шесть? Чтобы удобнее было смотреть на файл в шестнадцатеричном виде. *Красную шапочку* помните? *Почему у тебя такие большие глаза*? *Уши? Зубы?* Вот и в программировании то же самое. Многое программируется именно так, а не по-другому, только для того, чтобы потом было проще тестировать, отлаживать и исправлять ошибки. Это признак хорошего программиста, плохие программисты об этом и не думают. И ещё – проверку сигнатуры оформим как функцию, возвращающую булевское значение. Просто чтобы напомнить, как это делается.

А теперь я открою вам тайну. Ну или сакральное знание, как вам больше нравится. Я не шучу, многие программисты доходят до этого годами, во всех смыслах *доходят*. Программа должна быть максимально симметричной. Если мы задумали написать функцию, которая читает и проверяет сигнатуру, мы должны немедленно подумать и о функции, которая пишет сигнатуру. И мы её напишем тоже. А почему только пишет, но не проверяет? А потому, что мы абсолютно уверены, что на этапе записи сигнатура в полном порядке. Вот эти две функции:

```
goodSign = 'Random  '

def writeSign(f):
    data = struct.pack('<8s', goodSign)
    f.write(data)

def checkSign(f):
    data = f.read(8)
    sign = struct.unpack( '<8s',data)
    if sign[0] == goodSign:
        return True
    else:
        return False
```

Теперь ещё одно важное знание. Если какое-то значение, не совсем тривиальное, вроде нуля или единицы, используется в программе два раза

или больше, его не рекомендуется писать в программе как оно есть. В нашем случае этим значением является сигнатура `'Random '`. Мы должны завести специальную переменную, в котором это значение хранится.

расширение кругозора
В других языках есть такое понятие, как *константа*. Это выглядит почти как переменная, с одним только отличием. Константу нельзя изменить. В Питоне констант, в точном смысле этого слова, нет. Если вам непонятно, зачем нужна переменная, которую изменить нельзя, значит вы полностью прониклись духом Питона.
Кстати, в некоторых очень традиционных языках программирования есть ещё и константы, значения которых можно изменять.
конец Расширения

Программа для записи и на что в ней обратить внимание.

```
f = open( 'd:/rand.bin', 'wb')
writeSign(f)
howMany = random.randint(1,20)
print howMany

L = []
for i in xrange(0,howMany):
    r = random.randint(1,100)
    L.append(r)
print L

data = struct.pack( '<i4', howMany)
f.write(data)

for i in xrange(0,howMany):
    data = struct.pack(  '<i4', L[i])
    f.write(data)

f.close()
```

Возможно, у вас возник вопрос, почему в программе два цикла – один для генерации случайных чисел, другой для их записи. Почему нельзя было объединить всё в один цикл? Ведь программа стала бы короче? Это хороший, правильный вопрос. Этого делать не надо по очень простой причине – потому что это разные задачи и разные процессы. Можно даже выразиться сильнее – если что-то в программе можно поделить на части, обязательно делите.

И ещё. В коде присутствует отладочный вывод на экран. Он даёт такие результаты:

```
4
[18, 15, 95, 51]
```

Быстро сосчитайте в уме, каким будет размер файла. (Правильный ответ – 28). Займёмся чтением.

```
f = open( 'd:/rand.bin', 'rb')

Ok = checkSign(f)
L = []

if Ok:
    data = f.read(4)
    howMany = struct.unpack( '<i4', data)
    for i in xrange(0,howMany[0]):
        data = f.read(4)
        r = struct.unpack( '<i4', data)
        L.append(r[0])
else:
    print 'Bad. very bad.'

f.close()
```

Здесь прямо-таки напрашивается ряд интересных комментариев. Обратите внимание, что присвоение переменной списка пустого значения происходит *перед* входом в условный оператор. Причина понятна – даже если сигнатура окажется неправильной, список должен имеет хоть какое-то, но значение. Второе замечание менее оптимистично – вам надоело вечно писать нулевой индекс после чтения из файла в переменную? Переменная эта неизбежно оказывается кортежем. Ответа не требуется.

Обобщаем и систематизируем

Теперь суммируем, а точнее дополняем. Есть ещё некоторые, не затронутые нами типы данных. Целые, плавающие и строки – это практически всё, но есть и ещё кое-что. Снова предупреждаю – если вам интересен Питон и только Питон и сдача ЕГЭ с его применением – то этот раздел не для вас.

Начать с того, что целые переменные можно записывать в файл с разным размером – в байтах. Мы писали с размером четыре. Хорошо это или плохо? Есть такая философская наука – диалектика. Она учит нас, в частности, что истина конкретна. В нашем случае это означает, что если размер файла для нас важен, то, *возможно*, четыре много. А если у нас ну очень большие целые числа, то, *возможно*, это мало. Поясняю.

Целые числа можно записывать в файл с самой разной длиной. Записанных чисел может быть очень много, например миллионы. Это совершенно обычно. Размер целого числа при записи может меняться от двух до восьми байт. Чем меньше размер, тем больше чисел в файл поместится. Однако в наименьшее двухбайтовое число мы можем поместить только значение в диапазоне -32768..32767. Этого, как правило, оказывается недостаточно. В восьмибайтовое целое помещается почти всё, Но размер файла растёт в четыре раза. Выбирайте, что для вас важнее. При современном развитии технических средств размер файла не очень важен. Однако часто очень важна скорость передачи данных по каналам связи и приходится выбирать эти самые несчастные два байта, если диапазон данных помещается в них. А если к нам по каналу пришли только два байта, зачем писать в файл больше? Итак, как же пишутся эти целые в других форматах? Проще показать на примерах.

```
import struct

f = open( 'd:/something strange.bin', 'wb')

something = 128

# целое - два байта
data = struct.pack('<h2',something)
f.write(data)

# целое - четыре байта
data = struct.pack('<i4',something)
f.write(data)

# целое - восемь байт
data = struct.pack('<q8',something)
f.write(data)

f.close()
```

Как это всё прочитать, я указывать не буду, не сомневаюсь, вы справитесь отлично. Почти то же самое и с плавающими числами, только выбор там меньше.

```
somethingFloat = 123.45

#  плавающее - четыре байта
data = struct.pack( '<f4', somethingFloat)
f.write(data)

#  плавающее - восемь байт
data = struct.pack( '<d8', somethingFloat)
f.write(data)
```

Здесь по сути выбора нет – выбирайте четыре байта. Плавающее из восьми байт это я даже и не знаю, где может понадобиться. Четыре байта – это точность в 6-7 десятичных чисел. Восемь байт, понятно, в два раза больше. Если вы не запускаете ракеты на Марс, вам оно нужно?

Теперь, просто для полноты картины, запись булевских переменных.

```
theSenseOfLife = True

# булевское
data = struct.pack( '?1', theSenseOfLife)
f.write(data)
```

В первом приближении это всё, что надо знать о бинарных файлах. Есть ещё другие форматы и другие способы записи, например возможность одновременной записи большого количества однотипных данных, но без них можно обойтись. Пока можно.

Это хорошая, добрая глава. Мы будем рисовать. Сначала мне казалось, что глава эта будет почти никак не связана с другими. Оказалось, показалось. Прежде чем что-то нарисовать, очень неплохо будет освоить ещё один несложный метод работы с функциями. Конкретно – передача параметров по именам. Чуть раньше я его упоминал, но объявил, что нам он не нужен. Был неправ.

Подготовительные упражнения. Параметры по именам

А сначала вспомним о параметрах по умолчанию. Функция вычисления веса цилиндра по его радиусу, высоте и удельному весу, только теперь по умолчанию будут задаваться все параметры:

```
def weightCyl ( R = 5, h = 100, gamma = 7.85):
    result = 3.14158 * R**2 * h * gamma
    result = result / 1000
    return result

print 'iron', weightCyl( 5,100, 7.85)
print 'iron', weightCyl()

iron 61.6535075
iron 61.6535075
```

Это значит, что функцию можно вызывать без параметров вообще, результат будет тем же, что и с параметрами по умолчанию. Всё знакомо и всё понятно.

А теперь вызов параметров по именам. Заголовок функции станет совсем обычным, даже без параметров по умолчанию. С вызовом функции всё должно быть очевидно.

```
def weightCyl ( R, h, gamma):
    result = 3.14158 * R**2 * h * gamma
    result = result / 1000
    return result

print 'copper', weightCyl( R=5,h=100, gamma = 8.96)
copper 70.371392
```

Перед каждым значением параметра указано имя параметра. Зачем это нужно и какие даёт преимущества? Обычно говорят – так гораздо нагляднее и с первого взгляда всё понятно, причём всем. Второе преимущество, параметры теперь можно писать в произвольном порядке:

```
print 'copper', weightCyl( gamma = 8.96, R=5, h=100)
```

Я открою секрет, само по себе это практически бесполезно, но очень полезно в сочетании с параметрами по умолчанию. При использовании параметров по умолчанию можно пропускать и не писать *только* последние параметры, в любом количестве. Нельзя написать первый и третий, а второй пропустить. А теперь можно, вот таким образом. Только мы должны использовать первую версию функции, ту, что сверху.

```
print 'copper', weightCyl( R=7, gamma = 8.96)
```

Для чего это нужно и где это используется, вы увидите почти немедленно, как только мы займёмся непосредственно рисованием.

Начинаем рисовать

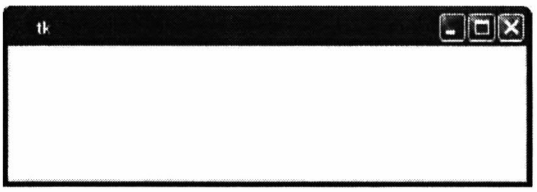
Чтобы сосчитать синус стандартной функцией, надо подключить модуль `math`. Чтобы нарисовать хоть что-то, надо подключить модуль `Tkinter`. На самом деле этот модуль делает гораздо больше. Он отвечает за работу с GUI – Graphic User Interface – Графический Интерфейс Пользователя. То есть за рисование кнопок, меню, флажков, полос прокрутки и многого-многого другого. И не только за рисование, но и за то, чтобы они правильно реагировали на события внешнего мира. Если клиент тыкает мышкой в кнопку, и что-то должно произойти, то за это тоже отвечает модуль `Tkinter`. Но пока кнопок не будет, мы будем просто рисовать – *палка, палка, огуречик, получился человечек.*

Кроме импорта модуля, необходимо добавить ещё несколько не вполне понятных с первого взгляда строк кода:

```
from Tkinter import *

c = Canvas( width = 400, height = 100, bg = 'white')
c.pack( expand = YES, fill = BOTH )
#
#  а здесь мы будем рисовать
#
```

```
mainloop()
```

Эту программу уже можно запустить на исполнение, вот результат.



Это совершенно стандартная программа для Windows. Её окно можно перетаскивать, его можно максимизировать на весь экран, можно скрыть. Закрыть программу можно, нажав на крестик в правом верхнем углу. Другого способа закрыть программу пока нет. Единственная индивидуальная черта программы – иконка в левом верхнем углу, напоминающая о модуле `Tkinter`.

Комментарий к программному коду. Со строкой импорта понятно. В следующей строке видим слово `Canvas`, по-английски это означает *холст*, во всех значениях этого слово. Конкретно здесь это тот холст, на котором рисуют художники, а сейчас будем рисовать мы. Это слово в этом же смысле встречается и во многих других языках программирования. Слева от оператора присваивания незаметная переменная `c`. Мы создаём объект типа *холст* и присваиваем ссылку на этот новый объект переменной.

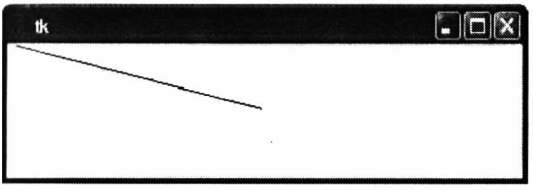
Объект создаётся не просто так, а с передачей функции, его создающей, нескольких параметров. Про объекты задумываться пока не надо, а вот о параметрах поговорим. Форма задания параметров однозначно указывает, что это параметры по именам. А если это параметры по именам, то их, скорее всего, гораздо больше, чем мы видим. Просто остальные получили значения по умолчанию. Значения первых двух очевидны, width = ширина, height = высота. Это размеры формы. Именно такие размеры у неё для того, чтобы не занимать много места на странице. Для рисования этого мало, немедленно увеличьте хотя бы до 400×400. Третий параметр не так очевиден, пока вы не догадаетесь, что bg = background = фон. Будучи совсем занудным, *white* это *белый*. То есть цвет фона нашего холста, на котором мы будем дальше рисовать, – белый.

Следующая строка, которая `c.pack`, является специфической для именно Питона. Её надо написать и всё тут.

А вот та строка, что после комментариев, то есть `mainloop()`, является общей для всех программ под Windows. Это так называемый *основной цикл*, возможны и другие имена. Смысл этого цикла в том, что он крутится *вечно*. Пока работает программа, работает и этот цикл. Задача его в том, чтобы реагировать на все внешние события – от движения мыши на один пиксел до… даже не знаю, до чего. Другое дело, что обычно – в других системах программирования – эта строка остаётся скрытой и никто её не видит. И даже никто не пишет. Но здесь вам не тут, здесь придётся писать явно.

Однако начинаем рисовать. Геометрия начинается с точки, но точку на экране почти не видно. Поэтому начнем с линии. Линия в геометрии задаётся двумя точками – началом и концом. В Питоне точно так же. Точка задаётся двумя координатами, по X и по Y. С координатой X всё хорошо – она дёт слева направо, как и положено. А вот координата Y в программировании традиционно перевёрнута – вверху ноль, а по мере опускания вниз координата растёт. Вспомнив всё это, наконец рисуем линию. Все наши будущие художества должны находиться на месте комментариев в заготовке программы.

```
c.create_line( 0,0, 200,50)
```



`c` – это объект холста. У объекта есть методы, обычно. Здесь вы видите метод с говорящим именем `create_line`, то есть *создать линию*. Далее координаты конечных точек.

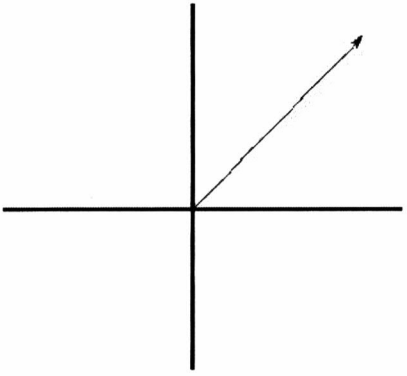
Линии со смыслом

Нарисуем что-то более осмысленное, конкретно оси прямоугольных координат. Сразу напрашивается что-то вроде такого:

```
c.create_line( 0,200, 400,200)
c.create_line( 200,0, 200,400)
```

Это будет работать, но это не очень хорошо. Почему? Центр координат у нас сейчас находится в точке (200,200). Если нам захочется его сдвинуть в другую точку, придётся менять все константы. Гораздо лучше, если для этого будет достаточно изменить только два этих числа – собственно начало координат. И чтобы два раза не вставать, вынесем в переменные и размер нашего поля вывода, который является квадратом со стороной 400. Ну, и чтобы попутно узнать что-то новое, нарисуем линии пожирнее и потолще. Для сравнения толщин линий проведём диагональ. И добавим ещё кое-что простое и полезное.

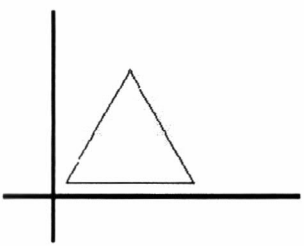
```
x0 = 200
y0 = 200
s  = 400

c.create_line( 0,y0, s,y0, width = 3)
c.create_line( x0,0, x0,s, width = 3)

c.create_line( x0,y0, x0+s/3,y0-s/3, arrow = LAST)
```



Ещё раз – всё это рисование вставляется на место комментариев в заготовке программы. Здесь мы видим два вида параметров – обычные и с именами. Сначала идут обычные, с именами – потом. Толщина линии задаётся как параметр с именем. А параметр `arrow` отвечает за стрелочку на конце линии. Если хотите стрелочку в начале, напишите `arrow = FIRST`. Если хотите обе сразу, укажите оба параметра.

На самом деле, возможности метода для рисования линий несколько шире. Разных параметров, или, по другой терминологии, *опций*, у него больше двадцати. Кроме того, точек, по которым рисуется линия, может быть больше двух. Демонстрирую. Сначала напишем функцию, рисующую равносторонний треугольник, в качестве параметров – сторона и координаты левого нижнего угла. Из геометрии. Высота равностороннего треугольника по его стороне вычисляется так: $h = \frac{a\sqrt{3}}{2}$. Для чего нам высота? В геометрии построение выполняется циркулем и линейкой. У нас ни циркуля, ни линейки нет, зато у нас есть прямоугольная система координат. Это даже лучше.

```
def Triangle( a, x0,y0):
    h = (a*3**0.5)/2.0
    c.create_line( x0,y0,        x0+a,y0)
    c.create_line( x0+a,y0,      x0+a/2,y0-h)
    c.create_line( x0+a/2,y0-h, x0,y0)

Triangle( 100, x0+10,y0-10)
```



Всё традиционно, только в очередной раз напоминаю, что `x0,y0` в заголовке функции и `x0,y0` в вызове функции это очень разные имена. Если присмотреться к вызовам метода, то прямо-таки становится очевидным, что вторая точка первой линии совпадает с первой точкой второй линии и так далее. А тогда зачем писать лишний код? Правильно, можно вот так:

```
def Triangle( a, x0,y0):
    h = (a*3**0.5)/2.0
    c.create_line( x0,y0, x0+a,y0, x0+a/2,y0-h, x0,y0)
```

Мы просто перечисляем точки, через которые должна пройти линия, только уже не прямая, а ломаная. Точек не три, а четыре, потому что

необходимо обеспечить замкнутость линии. Проверьте, результат должен быть тот же. Немного усложним код, или упростим, кому как покажется:

```
def Triangle( a, x0,y0):
    h = (a*3**0.5)/2.0
    L = [ x0,y0, x0+a,y0, x0+a/2,y0-h, x0,y0 ]
    c.create_line( L )
```

Результат рисования, предсказуемо, тот же. Мы сначала затолкали точки в список, а потом передали его для отображения. На этом мы не остановимся, ещё один вариант.

```
L = [ (x0,y0), (x0+a,y0), (x0+a/2,y0-h), (x0,y0) ]
```

В чём его преимущество? В том, что координаты сгруппированы попарно, каждая пара координат – точка. Так нагляднее. Теперь можно написать, казалось бы, длиннее, а значит, опять-таки, казалось бы, хуже. Но это только кажется.

```
def Triangle( a, x0,y0):
    h = (a*3**0.5)/2.0
    L = []
    L.append( (x0,y0) )
    L.append( (x0+a,y0) )
    L.append( (x0+a/2,y0-h) )
    L.append( (x0,y0) )
    c.create_line( L )
```

Вторые скобки необходимы, потому что метод `append` в состоянии принять на вход только один параметр. Согласно идеологии Питона, два числа, взятые в скобки, немедленно превращаются в один кортеж, который можно добавить в список за один приём. Иначе два числа пришлось бы добавлять за два раза. Это ничего, что я так подробно всё объясняю? Преимущество от такого подхода в том, что если точки можно добавлять по одной руками, то можно добавлять их и в цикле, а это даёт новые возможности. Вот пример – рисование правильного многоугольника. Это такой многоугольник, у которого равны все стороны и равны все углы. Самый любимый народом экземпляр, безусловно, квадрат.

бесполезная информация
Некоторые правильные многоугольники можно построить циркулем и линейкой. Гаусс доказал, что правильный многоугольник можно построить, если число его сторон равно простому числу Ферма. На данный

момент известны следующие простые числа Ферма – 3, 5, 7, 257, 65537. Подходит и любое число сторон, которое является произведением этих чисел и любой степени двойки, например $3 \times 7 \times 257 \times 2^{10}$.

Один слишком навязчивый аспирант довёл своего научного руководителя до того, что тот сказал ему: «Идите и разработайте построение правильного многоугольника с 65537 сторонами». Аспирант удалился, чтобы вернуться через 20 лет с соответствующим построением (которое хранится в архивах в Геттингене).

© Литлвуд, Математическая смесь.

Это потому, что у них компьютера не было.

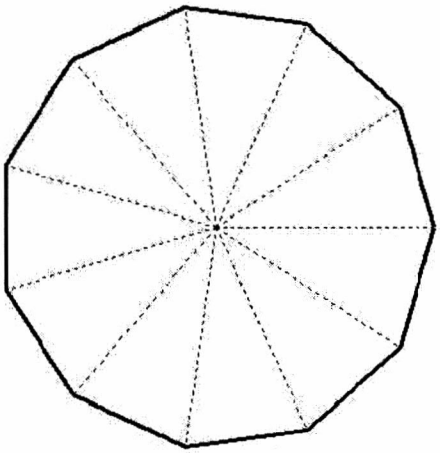
конец Бесполезной информации

На вход функции будут поступать очевидные параметры – количество вершин, длина стороны и позиция, от которой начинать отрисовку. И ещё один неочевидный, для контроля происходящего, параметр. А именно, рисовать ли радиусы, или как это у многоугольника называется, то есть – линии, соединяющие центр многоугольника с его вершинами.

```
def DrawRegPolygon( x0,y0,
                    numOfV, R,
                    drawRs = False):

        alfa=(math.pi*2)/numOfV

        if drawRs:
           for i in xrange(1,numOfV+1):
               x = round(R*math.cos(alfa*(i-1)));
               y = round(R*math.sin(alfa*(i-1)));

               c.create_line( x0,y0, x0+x,y0-y, dash = (1,1))

        L = [(x0+R),y0]
        for i in xrange(1,numOfV+1):
            x = round(R*math.cos(alfa*i));
            y = round(R*math.sin(alfa*i));
            L.append( (x0+x,y0-y) )
        c.create_line( L, width = 3 )
```

Формулы простейшие, самое начало тригонометрии. Вершины многоугольника помещаем в список и рисуем радостно всё сразу. Для начала правильный одиннадцатиугольник (очень сложное с точки правописания слово) и с радиусами, пожалуйста.

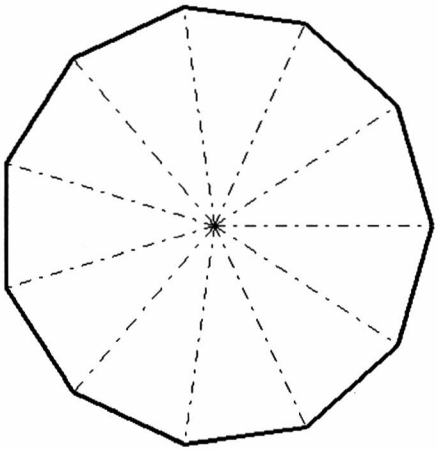
```
DrawRegPolygon( x0,y0, 11, 170, True);
```



Рисование многоугольника начинается с крайней правой точки, той, что лежит на положительной части оси абсцисс – если вы понимаете, о чём я говорю. Вершины мы рисуем в порядке, противоположном движению часовой стрелки. Counter-clockwise – это если вы хотите показаться избыточно умным.

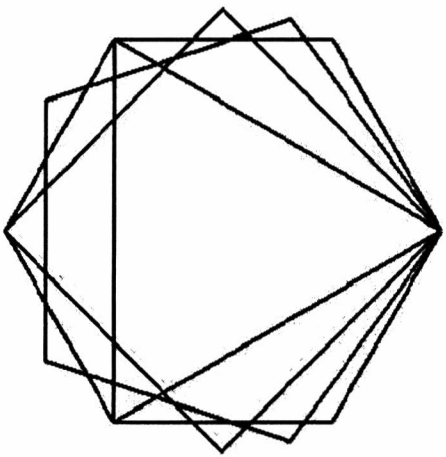
Должен привлечь ваше внимание к вот такому выражению: `dash = (1,1)`. Сначала оно меня в плохом смысле удивило, а потом очень даже понравилось. Понятно, что это параметр, переданный по имени. Непонятно, что в скобках. А в скобках описание линии, которую мы рисуем. Написанное в скобках (1,1) означает очень простую вещь – когда мы рисуем линию, то мы одну точку рисуем, а одну пропускаем. Непонятно? Если так: (5,1) – то рисуем тире. Что интереснее, описание линии может быть сколь угодно более заковыристым. Вот код, а вот результат:

```
c.create_line( x0,y0, x0+x,y0-y, dash = (1,1, 5,1))
```



Возвращаемся к геометрии. Радиусов больше не надо, просто нарисуем всего и много:

```
DrawRegPolygon( x0,y0, 3, 170);
DrawRegPolygon( x0,y0, 4, 170);
DrawRegPolygon( x0,y0, 5, 170);
DrawRegPolygon( x0,y0, 6, 170);
```



На этой милой картинке мы видим правильные многоугольники, от трёх до шести углов. Возможно, вам показалось, что всё это немного

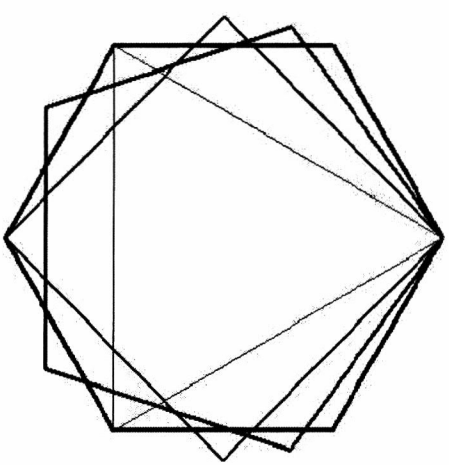
неразборчиво. Можно поправить. Добавим в заголовок функции параметр со значением по умолчанию.

```
def DrawRegPolygon( x0,y0,
                    numOfV, R,
                    w = 1,  # width
                    drawRs = False):
```

Реализацию осуществите сами. Подумайте, почему я дал ему имя `w`, а не `width`. Вот вызов:

```
DrawRegPolygon( x0,y0, 3, 170, 1);
DrawRegPolygon( x0,y0, 4, 170, 2);
DrawRegPolygon( x0,y0, 5, 170, 3);
DrawRegPolygon( x0,y0, 6, 170, 4);
```

Вот результат.



Гораздо лучше, правда?

Ежели один человек построил, другой завсегда разобрать может
© к/ф «Формула любви»

Если кто-то линии рисует, значит, кто-то их и стирает? Специального метода для стирания линий, увы, нет. Но уже нарисованную линию можно заново нарисовать белым цветом, точнее цветом фона. Если точнее, линию можно нарисовать любым цветом. Книга не цветная, поэтому картинки не будет, а пример кода будет.

```
c.create_line( x0,y0, x0+s/3,y0-s/3, fill = 'green')
```

Обратите внимание, что цвет задаётся в кавычках. Так как же стереть линию и зачем это нужно? Очевидно, если мы что-то нарисовали чёрным, вот так или вот так

```
c.create_line( x0,y0, x0+s/3,y0-s/3)
c.create_line( x0,y0, x0+s/3,y0-s/3, fill = 'black')
```

то стирать будем так:

```
c.create_line( x0,y0, x0+s/3,y0-s/3, fill = 'white')
```

А нужно это бывает для анимации, то есть для реализации движения объекта на экране. Анимация – это не обязательно игры или мультфильмы, анимация применяется и в очень серьёзных и скучных программах.

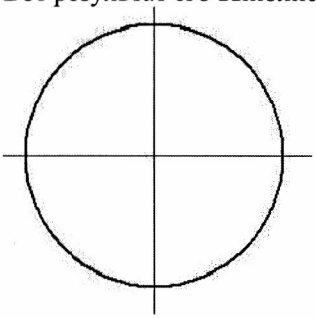
Круги и прочие эллипсы

С прямыми у нас уже всё очень хорошо, я надеюсь. Теперь кривые. Самая совершенная из кривых – круг. И его лучший друг эллипс, разумеется. Где-то далеко вверху мы уже нарисовали оси координат и треугольник на их фоне. Треугольник уберём и нарисуем круг, с центром в начале координат и радиусом сто. Это самое простое, что только может быть с нарисованным кругом. Вот программный код, базирующийся на ранее написанном:

```
from Tkinter import *

c = Canvas( width = 400, height = 400, bg = 'white')
c.pack(expand = YES, fill = BOTH)

x0 = 200
y0 = 200
s =  400
R  = 100

c.create_line( 0,y0, s,y0, width = 1)
```

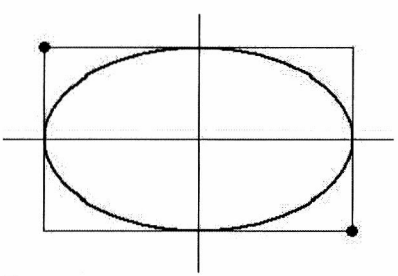
```
c.create_line( x0,0, x0,s, width = 1)

c.create_oval( x0-R,y0-R, x0+R,y0+R, width = 2)

mainloop()
```

Вот результат его выполнения:



Теперь комментарии. Круг рисует метод `create_oval`. У метода четыре параметра. Параметры эти задают прямоугольник, в который вписан эллипс. Первые два – координаты левого верхнего угла, следующие два – координаты левого нижнего угла. Если наш центр координат находится в точке (200,200) то для круга с радиусом 100 надо задать (200-100,200-100, 200+100,200+100) = (100,100, 300,300).

Непонятно? Вот иллюстрация:



Вот эти жирные точки мы и должны задать методу `create_oval`, чтобы нарисовать тот эллипс, который внутри прямоугольника. Примечание – прямоугольник при этом не нарисуется, он здесь только для наглядности. Обратите внимание, эллипс чуть раньше мы рисовали вот так:

```
c.create_oval( x0-R,y0-R, x0+R,y0+R, width = 2)
```

Последний параметр, как несложно догадаться, это толщина линии, которой нарисован эллипс. Жирные точки по углам – тоже эллипсы, только маленькие, и рисуются они так:

```
c.create_oval( x0-120-4,y0-70-4, x0-120+4,y0-70+4, fill = 'black')
c.create_oval( x0+120-4,y0+70-4, x0+120+4,y0+70+4, fill = 'black')
```

Величины 120 и 70 – радиусы нашего основного, большого эллипса. В правильной программе вместо конкретных чисел должны стоять переменные. 4 – радиус жирной точки. А последний параметр делает жирную точку действительно жирной – он закрашивает весь маленький эллипс чёрным цветом.

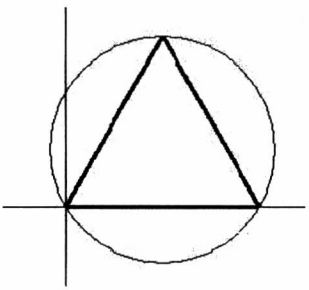
А теперь задача чуть более математическая. Нарисуем равносторонний треугольник и описанную вокруг него окружность. Равносторонний треугольник это такой, у которого равны все три стороны и все три угла. Кстати, из первого равенства следует второе и наоборот. Левый нижний угол треугольника пусть совпадает с началом координат.

Напомню, хотя вы наверняка знаете, что центр описанной вокруг равностороннего треугольника окружности лежит в точке $(\frac{a}{2},\frac{h}{3})$, а радиус её равен $R=\frac{a\sqrt{3}}{3}$, где a – сторона треугольника, а h, в свою очередь, высота треугольника, равная $h=\frac{a\sqrt{3}}{2}$.Всё для вас должно быть уже просто и очевидно.

```
from Tkinter import *

x0 = 200
y0 = 200
a = 150
h = (3**0.5)*a/2
R = (3**0.5)*a/3

c = Canvas( width = 400, height = 400, bg = 'white')
c.pack(expand = YES, fill = BOTH)

c.create_line( 0,y0, 400,y0)
c.create_line( x0,0, x0,400)

c.create_line(x0,y0, x0+a,y0, width=3)
```

```
c.create_line(x0,y0, x0+a/2,y0-h, width=3)
c.create_line(x0+a,y0, x0+a/2,y0-h, width=3)

c.create_oval( x0+a/2+R,y0-h/3+R, x0+a/2-R,y0-h/3-R)

mainloop()
```

Довольно симпатичный результат.

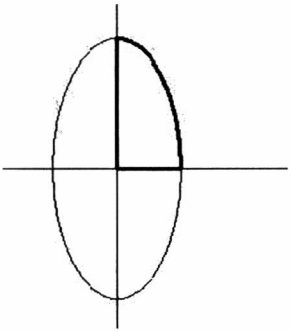


В геометрии принято обозначать вершины, углы и вообще всё, что только возможно, буквами. Буквами мы займёмся в следующем разделе, а пока завершим тему с кругами и эллипсами. Методов у холста не так и много, так что можно рассмотреть вообще все. Следующий метод называется `create_arc()`. Он тоже рисует эллипс, но не весь, а только часть от него. В геометрии это называется *дуга*.

Для начального постижения достаточно следующего кода, если после него посмотреть на картинку. На картинке, в обязательном порядке, оси координат и *цельный эллипс*, часть от которого мы рисуем.

```
x0 = 100
y0 = 200

c.create_line( 0,y0, 400,y0)
c.create_line( x0,0, x0,400)

c.create_arc( x0-50,y0-100, x0+50,y0+100, width = 3)
c.create_oval( x0-50,y0-100, x0+50,y0+100, width = 1)
```

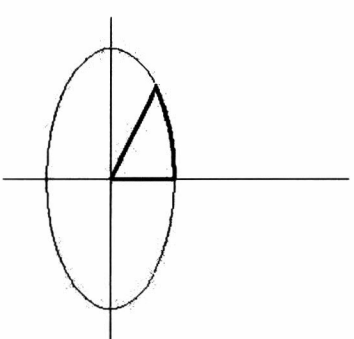
Я не пишу здесь совершенно обязательные строки про **`import`**, `pack`, `mainloop` и прочее – вы всё это уже прекрасно освоили. Вот результат:



То, что картинка как бы вытянута по вертикали, так это так и задумано. Тонкой линией нарисованы координаты и тот эллипс, от которого мы откусили кусок. Сама дуга толстая и жирная. Что интересно? Для рисования эллипса и дуги использованы одни и те же координаты, а результат принципиально разный. Будем разбираться. По умолчанию, дуга имеет размер ровно в 90°. В этом отношении Питон ведёт себя очень дружественно к программисту, в других языках используют вместо градусов радианы. Дуга идёт от горизонтали – то есть от линии, идущей между началом координат и крайней правой точкой нашего эллипса. Заданные 90° отсчитываются *против* часовой стрелки. Если всё это кажется вам пока не очень понятным, то я с вами согласен. Постигать рисование дуг будем на опытах.

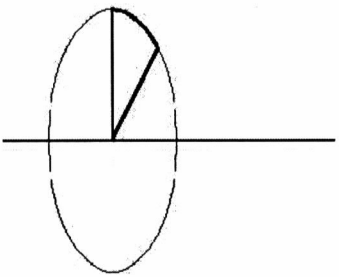
Для начала зададим дугу поменьше, в 45°.

```
c.create_arc( x0-50,y0-100, x0+50,y0+100, width = 3, extent = 45)
```



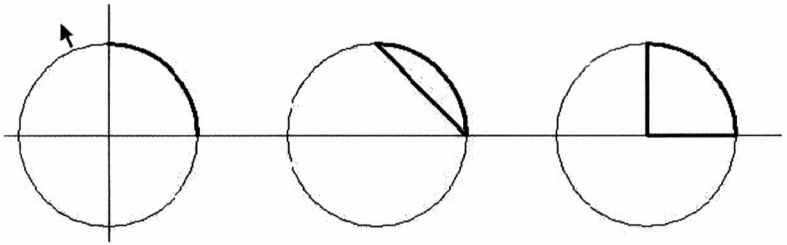
Итак, параметр `extent` отвечает за размер дуги в градусах. Теперь видно, почему оговорено рисование именно против часовой стрелки. Идём дальше. Сейчас мы нарисовали дугу между тремя и шестью часами, если вернуться к понятиям часового циферблата. А если надо между полуднем – двенадцатью – и тремя? Учим новое слово.

```
c.create_arc( x0-50,y0-100, x0+50,y0+100, width = 3,
              extent = 45, start = 45)
```



Разумеется, то, что два раза встретилось число 45 – это чисто случайно и совпадать они не обязаны. Третий параметр, представляющий интерес и уникальный именно для дуги, называется не очень оригинально – `style`. Чем рассказывать словами, проще показать.

```
x0 = 100
y0 = 200
r  =  70

c.create_arc( x0-r,y0-r, x0+r,y0+r, width = 3, style = ARC)
c.create_oval( x0-r,y0-r, x0+r,y0+r)
x0 = x0 + 3*r

c.create_arc( x0-r,y0-r, x0+r,y0+r, width = 3, style = CHORD)
c.create_oval( x0-r,y0-r, x0+r,y0+r)
x0 = x0 + 3*r

c.create_arc( x0-r,y0-r, x0+r,y0+r, width = 3, style = PIESLICE)
c.create_oval( x0-r,y0-r, x0+r,y0+r)
```



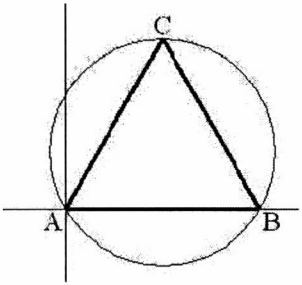
Извините за некоторую небрежность стиля программирования. Разумеется, лучше было бы оформить рисование дуги и соответствующего ей эллипса в виде функции. Зато так быстрее. Главное, всё сразу понятно.

Текст

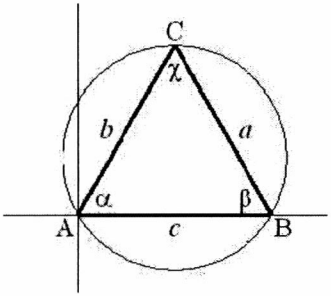
Для вывода текста используется метод с совершенно неожиданным названием `create_text`. Ничего принципиально сложного в нём нет. Мы задаём *куда* выводить, *что* выводить и *как* выводить. С технической точки зрения, возникают проблемы с точной подгонкой места вывода – как сделать так, чтобы текст не перекрывал линии.

```
c.create_text( x0-10,y0+10,    text='A', font=('Times',16))
c.create_text( x0+a+10,y0+10,  text='B', font=('Times',16))
c.create_text( x0+a/2,y0-h-10, text='C', font=('Times',16))
```

Последний параметр, шрифт, при выводе текста очень важен. При выводе линии мы могли не задавать её цвет и толщину. По умолчанию линия чёрная и толщина её равна единице, обычно это всех устраивает. Параметры по умолчанию для шрифта обычно не устраивают никого. Не потому, что они плохие, а потому, что требования к тексту бывают очень и очень разными. Далее – шрифт задаётся немного сложнее, чем цвет или толщина. Шрифт – это, строго выражаясь, кортеж. Кортеж, напоминаю, это нечто, заключённое в круглые скобки и через запятую. Для какого-нибудь другого языка программирования такое определение прозвучало бы наивно и даже глупо, а в Питоне – в самый раз. Задаём мы имя шрифта и размер. Поскольку параметры передаются не по именам, порядок их следования важен. А теперь, наконец, результат.



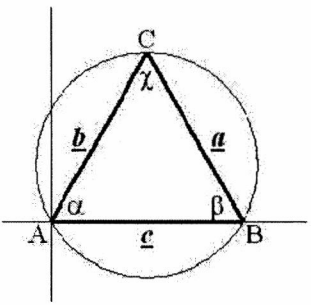
Однако на этом мы не остановимся. Обозначим ещё и стороны, а также углы. По традиции, стороны обозначаются маленькими буквами и курсивом. Сами буквы совпадают с буквами для углов напротив сторон. Чтобы всех запутать, углы при вершинах A,B,C подписываются греческими буквами α, β, γ. Пойду против традиции и сначала приведу результат, а только потом код.



Да, выглядит как-то не очень и вообще, напоминает символику какой-то масонской ложи, если вы вообще понимаете, о чём я говорю. Тем не менее, с точки зрения геометрии, всё правильно. Теперь с точки зрения программирования. Добавились следующие строки:

```
c.create_text( x0+h/6,y0-h/2, text='b', font=('Times',16, 'italic'))
c.create_text( x0+h,y0-h/2,   text='a', font=('Times',16, 'italic'))
c.create_text( x0+a/2,y0+10,  text='c', font=('Times',16, 'italic'))

c.create_text( x0+20,y0-14, text='\x61', font=('Symbol',16))
c.create_text( x0+a-20,y0-14, text='\x62', font=('Symbol',16))
c.create_text( x0+a/2,y0-h+16, text='\x63', font=('Symbol',16))
```

Что интересного мы здесь видим? Места, куда выводить текст, подбираются научным *методом тыка*. С этим ничего не поделать. Греческая буква γ выглядит несколько не так, но с этим тоже ничего не поделать – просто подберите другой шрифт, посимпатичнее. В описании шрифта добавился ещё один параметр – `'italic'`. Слово это, в переводе с их языка на человеческий, означает *курсив*. Параметр этот несколько нетрадиционен, в том смысле, что слов в строке может быть много, например `'italic bold underline'`. Результат будет таким –



То есть обозначения сторон мало того что остались написанными курсивом (*italic*), но ещё стали жирными (***bold***), а также подчёркнутыми (***underline***). Это, в общем, понятно.

С обозначениями углов интереснее. Углы – по традиции – обозначаются греческими буквами, которые в среде IDLE набрать, мягко говоря, затруднительно. Для отображения греческих букв мы используем шрифт Symbol. Почему именно этот? Универсальный совет – выбираете в главном меню <Windows Настройка\Панель управления\Шрифты> и ищете шрифт, в котором есть нужный вам символ. Дальше надо узнать, какой у этого символа код – у любого символа есть код. Как узнать код? Для этого есть специальные, то, что называется *сторонние* программы. Вы их без труда найдете. Я же, честно признаюсь, подобрал этот код опять-таки *методом тыка*, без особых усилий.

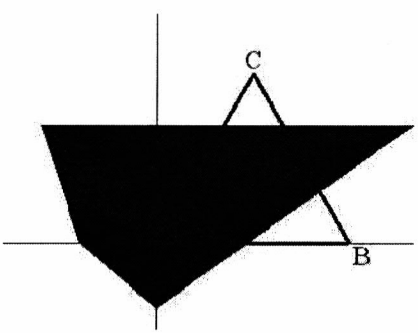
Что важнее, обратите внимание, на то, как задаётся этот код символа. А именно `text='\x61'`. Об этом мы уже говорили раньше, в главе о строках. `x` обозначает шестнадцатеричную систему счисления. Бэкслеш – обратный слеш – разделяет символы, заданные в этой самой шестнадцатеричной системе.

Прямоугольники и многоугольники. В том числе и без углов

Многоугольники мы уже вполне успешно рисовали, но теперь подойдём к делу более систематически и формально. Нарисуем четырехугольник. Совершенно не правильный и, не побоюсь этого слова, даже где-то корявый. Нарисуем его на фоне уже ранее нарисованных осей координат и равностороннего треугольника. Для чего? Чтоб лучше было видно – и наши достижения и наши неудачи. Начало и конец кода я пропущу – вы ведь уже всё это отлично знаете. Только оси, треугольник и наш уродливый многоугольник.

```
c.create_line( 0,y0, 400,y0)
c.create_line( x0,0, x0,400)

c.create_line(x0,y0, x0+a,y0, width=3)
c.create_line(x0,y0, x0+a/2,y0-h, width=3)
c.create_line(x0+a,y0, x0+a/2,y0-h, width=3)

c.create_polygon( 110,110, 400,110, 200,250, 140,200)
```

Вот что получилось, немного неожиданно, правда, ведь мы ожидали совсем не этого:

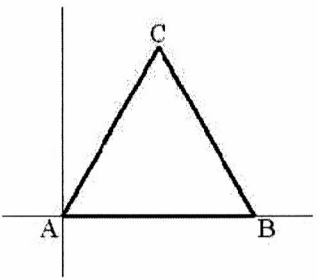


Не знаю, может вам хотелось увидеть именно *это*, но что касается меня, то точно нет. Хорошая сторона – нам достаточно указать только вершины многоугольника, замкнутость многоугольника будет обеспечена без нашей заботы. Теперь о главном – а почему он весь чёрный? Проблема в параметре `fill`, точнее, не в самом параметре, а в том, что мы его не

задали. Параметр этот отвечает на вопрос – а каким, собственно, цветом закрашивать внутренности многоугольника? Если параметр пропущен, то, как и обычно – чёрным. То есть отсутствующий параметр, это то же самое, что и `fill = 'black'`. Нас это не устраивает. Если мы напишем `fill = ''`, то внутренности останутся прозрачными, то есть – всё, что было внутри многоугольника, всё это и останется в неприкосновенности. Пробуем. Код:

```
c.create_polygon( 110,110, 400,110, 200,250, 140,200, fill = '')
```

Результат:



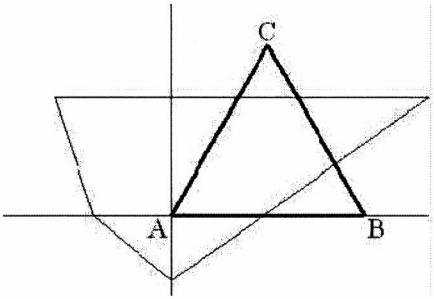
Но ведь это уже совсем никуда не годится! Просто бред какой-то! Что же случилось? Действительно, бред какой-то. Питон считает, что если ему не дано другого указания, то контур многоугольника надо рисовать *прозрачным* цветом. Прозрачный – это невидимый, если что.

Я не знаю, зачем и кому это нужно
© Песня, А. Вертинский, поётся грустным голосом
Шо маэмо, то маэмо
© Украинская универсальная отмазка

Придётся задать цвет контура явным образом.

```
c.create_polygon( 110,110, 400,110, 200,250, 140,200,
                  fill = '', outline = 'black')
```

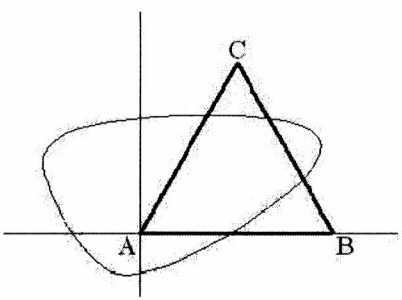
Имеем именно то, что и хотели иметь:



Немного страшно выглядит, конечно, но мы ведь именно этого и хотели? Последний штрих. Напишем вот так:

```
c.create_polygon( 110,110, 400,110, 200,250, 140,200, fill = '',
                  outline = 'black', smooth = 1)
```

Последний загадочный параметр отвечает за то, стоит ли нам нарисовать наш многоугольник со сглаженными, закруглёнными углами.



Маленький, но важный, комментарий. Параметр, отвечающий за гладкость, прямо-таки всем своим видом демонстрирует, что является числом. Кажется, что если задать значение побольше, то и многоугольник станет ещё глаже. Первое впечатление обманчиво. По смыслу, это булевская переменная – или да или нет. Единица – да, ноль – нет. Более наглядно, хотя и немного длиннее, можно записать так:

```
c.create_polygon( 110,110, 400,110, 200,250, 140,200, fill = '',
                  outline = 'black', smooth = True)
```

Разумеется, многое из того, что было сказано о рисовании линий, точно так же относится и к многоугольникам. Вот пример:

```
L = [(110,110), (400,110), (200,250), (140,200)]
c.create_polygon( L, fill = '', outline = 'black', width = 7)
```

Картинку приводить не буду, поверьте, а лучше проверьте – вы увидите тот же, незакруглённый, четырёхугольник, только нарисованный очень и очень жирной линией. Важнее то, что вершины переданы в виде списка. Такой способ мне кажется более наглядным и более доступным для внесения неизбежных изменений в программу. В реальные программы всегда приходится вносить изменения.

Картинки

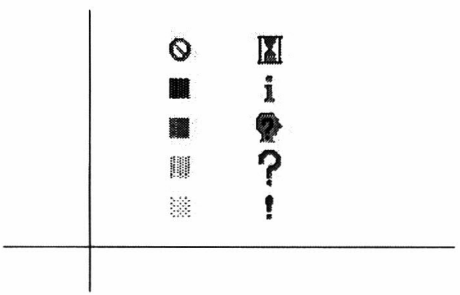
Картинки в Питоне можно рисовать двумя методами. Отличие между ними не сказать, чтобы чёткое и ясное, но мой долг описать оба способа – а вдруг пригодится.

Способ первый – метод холста `create_bitmap()`. Он выводит, как нетрудно догадаться, изображение в формате Windows bitmap, обычно хранимые в файлах с расширением *.bmp. Я, конечно, язык Питон очень уважаю, но не всё в нём просто и очевидно. И здесь мы имеем тот самый случай неочевидности. С помощью этого метода мы можем нарисовать одну из десяти заранее определённых картинок – или свою собственную, произвольную. Маленькое ограничение – картинки могут быть только монохромными, то есть двухцветными. По умолчанию они чёрно-белые, но могут быть и красно-зелёными, к примеру. Сначала выведем картинки дармовые, сделанные до нас, все десять, оси координат сохранены для наглядности.

```
from Tkinter import *

x0 = 100
y0 = 200
shift = 70

c = Canvas( width = 400, height = 400, bg = 'white')
c.pack(expand = YES, fill = BOTH)

c.create_line( 0,y0, 400,y0)
c.create_line( x0,0, x0,400)
```

```
c.create_bitmap( x0+shift, 50, bitmap = 'error');
c.create_bitmap( x0+shift, 80, bitmap = 'gray75');
c.create_bitmap( x0+shift,110, bitmap = 'gray50');
c.create_bitmap( x0+shift,140, bitmap = 'gray25');
c.create_bitmap( x0+shift,170, bitmap = 'gray12');

c.create_bitmap( x0+shift*2, 50, bitmap = 'hourglass');
c.create_bitmap( x0+shift*2, 80, bitmap = 'info');
c.create_bitmap( x0+shift*2,110, bitmap = 'questhead');
c.create_bitmap( x0+shift*2,140, bitmap = 'question');
c.create_bitmap( x0+shift*2,170, bitmap = 'warning');

mainloop()
```

Результат:



Не сказать, чтобы очень красиво. Можно раскрасить.

```
c.create_bitmap( x0+shift*2, 50, bitmap = 'hourglass',
                 foreground = 'green', background = 'red');
```

Получили зелёные часы на красном фоне – омерзительное зрелище. Для чего вообще нужны эти убогие картинки? Забегая вперёд, Питон умеет создавать и вполне настоящие программы с интерфейсом под Windows – с кнопками и прочим счастьем. Так вот, основное предназначение этих картинок в том, чтобы помещать их на кнопках и других графических элементах.

Но, как уже было сказано, позволено вывести и что-то своё, из файла. Хотя Питон и здесь идёт своим путём. В любом графическом редакторе можно создать файл формата BMP – но Питону он не подойдёт. Нужен обязательно файл формата X Bitmap. Что это такое? Как ни странно, это текстовый файл в шестнадцатеричном формате. Или, если так звучит лучше, шестнадцатеричный файл в текстовом формате.

Вам непонятно? Что вам? Даже программистам преклонного возраста непонятно. Сам формат – дремучий атавизм.

// анатомическое разъяснение
Атавизм, если кто не знал, да ещё и забыл, это такая часть организма, которая когда-то была очень полезна, а теперь никому ни нужна и растёт чисто по привычке. Например, у человека есть хвост – называется *копчик*. Раньше он был длинный и эффективный, можно было висеть и раскачиваться, зацепившись за ветку. Теперь он маленький и где-то под кожей, но он там есть. Проверьте.
// конец Анатомии

Так вот, формат X Bitmap пришёл к нам из древних UNIX-систем. UNIX – это такая операционная система. Формат действительно текстовый. Это значит, что файл с картинкой можно создать в любом текстовом редакторе, например *Блокнот* в Windows. Много раз уже где-то говорил и повторю ещё раз – если вы что-то набрали в Microsoft Word, это не обязательно текстовый файл. Скорее всего, это вообще не текстовый файл. Таковым он станет только если вы сохраните его как *Обычный текст (*.txt)*. По крайней мере, так это называется в моей версии Word.

Итак, наберите где угодно вот такой текст:

```
#define mu_width  32
#define mu_height  8
static unsigned char x_bits[] = {
   0x00, 0x00, 0x00, 0x00,
   0xff, 0xff, 0xff, 0xff,
   0xff, 0xff, 0xff, 0xff,
   0xff, 0xff, 0xff, 0xff,
   0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00,
};
```

Сохраните его как простой текст. Имя файла пусть будет mu.xbm. *.XBM – традиционное расширение для файлов типа X Bitmap. Что мы сделали? Первые две строки задают ширину и высоту, именно в таком порядке, наше картинки. Сразу видно, что её размер 32×8. В директиве #define при задании ширины и высоты традиционно используется имя файла. Если

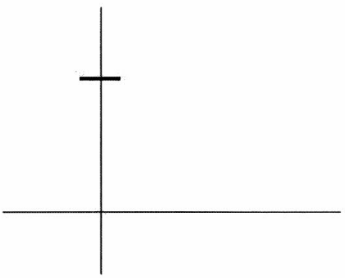
файл называется mu.xbm, директива – `#define mu_width 32`. Это необязательно, префикс может быть любым, за исключением `x`. На всякий случай, префикс – это то, что *перед* подчёркиванием.

Третья строка имеет чисто ритуальный характер. Последняя, где скобка и точка с запятой, тоже. Отступы, опять-таки, только для красоты.

Теперь о главном – о самом изображении. Во-первых, как его вывести на экран? Вот так:

```
c.create_bitmap( 100,100, bitmap = '@d:\mu.xbm')
```

Обратите внимание на *коммерческое эт* (официальное название) перед именем файла. По другому этот символ зовут *собака*. Во-вторых, а каким оно, созданное нами изображение, будет? Не очень впечатляющим, признаюсь.



Вот эту жирную чёрточку, которая поперёк, мы и нарисовали. С формальной точки зрения, это прямоугольник. Теперь будем разбираться, а как же мы его нарисовали. Файл состоит из строк вида `0x00, 0x00, 0x00, 0x00,`. То, что файл разделён на строки, никакой роли не играет, это исключительно для наглядности. Строка состоит из групп вида `0x00,` перечисленных через запятую. Каждая группа отвечает за восемь точек по горизонтали, значит, одна текстовая строка это восемь умножить на четыре, равняется 32 точки, что нам и требовалось.

Для дальнейшего надо что-то знать о системах счисления, конкретно о двоичной и шестнадцатеричной. Очень коротко и ясно всё об этом изложено в приложении. Каждая группа вида `0x00` является шестнадцатеричным числом. Первые два символа обязательны для формата записи и самостоятельного смысла не несут. Следующие два

важны. Теперь вспоминаем – $00_{16} = 00000000_2$. Вот эти восемь двоичных цифр, конкретно нулей, отвечают за восемь точек на экране и по горизонтали. Если у нас нули – ничего не рисуется. А что нарисует группа `0xff`? Элементарно. $FF_{16} = 11111111_2$. Имеем восемь нарисованных точек.

Для завершения и полного понимания – а что, если нам надо нарисовать что-то полосатое, через точку? Можно, например, так – $10101010_2 = AA_{16}$. То есть в текстовом файле должно стоять `0xaa`. Проверьте и обдумайте.

Всегда полезно, узнав что-то новое, объединить его с чем-то старым. И старое закрепляется и новое подтверждается.

– Откушать просим, доктор, чем бог послал.
– Откушать можно. Коли доктор сыт, так и больному легче.
© к/ф «Формула любви»

Данные для рисования картинки берутся из текстового файла. Файл мы создавали неведомо где, неведомо чем и неведомо как. Создадим его сами, своими белыми ручками, в той же программе чуть раньше и непосредственно перед выводом на экран.

```
f = open( 'd:/mu_1.xbm', 'w')

f.write( '#define mu_width   8\n');
f.write( '#define mu_height  8\n');
f.write( 'static unsigned char x_bits[] = {\n');
f.write( '   0x81,\n');
f.write( '   0x42,\n');
f.write( '   0x24,\n');
f.write( '   0x18,\n');
f.write( '   0x18,\n');
f.write( '   0x24,\n');
f.write( '   0x42,\n');
f.write( '   0x81);\n;');

f.close()

c.create_bitmap( 200,100, bitmap = '@d:\mu_1.xbm')
```

Это я для разнообразия нарисовал крестик. Если захотите изображение покрасить, то сказанное раньше о цветах полностью применимо и здесь:

```
c.create_bitmap( 200,100, bitmap = '@d:\mu_1.xbm', foreground = 'blue')
```

А можно ли просто и без затей загрузить изображение из файла какого-то более привычного формата? Можно. Но, как и во многих других случаях, Питон идёт здесь своим путём. Для начала, вы можете вывести изображение в формате PPM. Вы о таком слышали? Вот и я тоже. Или слышал и много лет как забыл. Далее следует столь же популярный формат PGM. К счастью, ещё допускается и вполне традиционный формат GIF. Для очень образованных, знающих, что GIF бывает и анимированный, уточняю – анимация не предусмотрена. В этом случае будет выведен только первый кадр из файла. Но есть и хорошая новость – изображение имеет право быть цветным. Никаких дополнительных заклинаний для этого произносить не требуется. Если изображение цветное – оно и отобразится как цветное.

Рисуется изображение в два шага:

```
im = PhotoImage( file = 'd:/t.gif')
c.create_image( 100,100, image = im)
```

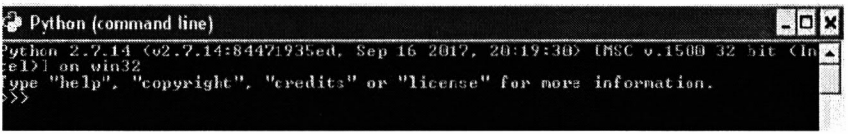
Ничего сложного. На данном этапе постижения Питона это надо просто запомнить. Никаких специальных лично для себя параметров метод не имеет.

Если хочется странного. Библиотека PIL

PIL расшифровывается как Python Image Library – ведь всё понятно и без перевода, правда? Библиотеку уже не поддерживают почти десять лет, но есть то, что в кинематографе называется *вбоквел*, а у некоторых программистов fork. Эта штука называется Pillow и, по слухам, имеет намного большие возможности. Нам выше головы хватит и начальной библиотеки PIL. Библиотека совершенно бесплатна. Где взять? Можно здесь – *http://www.pythonware.com*. Или просто наберите в любом поисковике PIL. Установка библиотеки не представляет никаких затруднений – при условии, что сам Питон установлен корректно. Вы справитесь.

Сейчас мы займёмся крайней левой иконкой. Выглядит она вот так – 

. Извините за чёрный фон, в оригинале он был зелёным. Если вызывать программу через Меню, то вызов будет выглядеть так – Программы \Python 2.7 \Python (command line). Опять-таки, во всех книгах это называется *консоль*. После запуска выглядит консоль, скажем прямо, страшно:



Даже я такого ужаса много десятилетий не видел. Теперь пишем текст в консоли:

```
>>>2+3
5

>>>2-3
-1

>>>2*3
6
```

То есть пока что мы имеем банальный калькулятор. Как вы поняли, умножение обозначается звёздочкой. Продолжаем опыты.

```
>>>2/3
0

>>>2 % 3
2

>>>3 % 2
1
```

Несложно догадаться, что косая черта производит деление нацело, а процент, наоборот, возвращает остаток от деления. Но всё-таки, что делать, если нам нужно настоящее деление, с дробями?

```
>>>2.0/ 3.0
0.66666

>>>2/3.0
0.66666
```

Надо явным образом указать, что у нас не целые числа, а дробные, то есть после целой части поставить точку. Можно у обоих чисел, можно у одного.

Как мы видим, консоль можно использовать в качестве калькулятора. На самом деле, в консоли Питона можно писать самые настоящие программы. Вот пример маленькой, но программы:

```
>>>a = 2
>>>b = 3
>>>c = a + 2**b
>>>print c
10
```

А вот совсем настоящая программа:

```
x = -1
if x > 0:
…    print 'positive'
else:
…    print 'negative'
negative
```

Можно делать почти всё, только к чему эти извращения? Зачем нам вообще эта консоль? Забудьте.

Приложение В. Другие числа

Этот раздел можно вообще не читать – потому он здесь, в приложениях. Если не прочитаете, ничего не потеряете. Если прочитаете, узнаете что-то новое и расширите кругозор. Узнать что-то новое, тем более относящееся к тому, чем вы интересуетесь, всегда полезно. В любом случае, я вас предупредил.

Занудства ради, речь в этом разделе не о том, какие бывают числа в математике, кому надо, тот знает. Речь о том, какие типы чисел определены в Питоне – и кому они вообще нужны. Если вы не забыли, то я говорил, а математика подтверждает – между целыми числами и числами действительными, они же плавающие, незаметно промелькнули числа рациональные. Это такие числа, которые могут быть выражены как результат от деления двух целых чисел – $\frac{2}{4}, \frac{2}{3}, \frac{1}{1024}, \frac{7}{17}$. Дроби могут быть сокращаемыми, как $\frac{2}{4}$, или нет, как все остальные в этом ряду – это непринципиально. Ещё дроби бывают неправильные, те, у которых числитель (сверху) больше знаменателя (снизу). Из неправильных дробей можно выделить целую часть. Неправильные, к примеру, $\frac{23}{17} = 1\frac{6}{17}, \frac{99}{1} = 99$. Питон позволяет иметь дело с рациональными дробями именно как с рациональными дробями. То есть из школьной арифметики помним $\frac{2}{3} + \frac{3}{5} = \frac{19}{15} = 1\frac{4}{15}$. Как это выглядит в Питоне – если использовать встроенный тип *дроби*?

```
import fractions

a = fractions.Fraction (2,3)
b = fractions.Fraction (3,5)

c = a + b
print a,b,c

2/3 3/5 19/15
```

Заметив, что неправильная дробь так и осталась неправильной, присмотримся к остальному. Чтобы вообще получить возможность работать с дробями, необходимо подключить модуль, который называется

import fractions. Дроби сами собой ниоткуда не возникают, надо писать имя модуля и имя создающей дробь функции, говоря по-другому – конструктора функции. В чём преимущество использования дробей? Это легко увидеть, проделав те же вычисления, но с обычными плавающими числами.

```
afl = 2.0/3.0
bfl = 3.0/5.0

cfl = afl + bfl
print afl,bfl,cfl

0.666666666667 0.6 1.26666666667
```

Несложно заметить, что всё пошло не так. Узнать в записи 1.26666666667 натуральную дробь $\frac{19}{15}$ очень и очень непросто. У питоновских натуральных дробей есть ещё одна очень интересная особенность. Особенность столь же интересная, сколь и бесполезная, если не сказать – вредная. Дроби не обязательно задавать в классической форме – с числителем и знаменателем. Общепринятой является и десятичная форма. Вот пример:

```
x = fractions.Fraction(0.125)
print x

1/8
```

Для тех, кто забыл, напоминаю $0.125 = \frac{1}{8}$. Всё работает отлично, пока. Теперь резко упрощаем задачу, не 0.125, а всего-навсего 0.1. Результат несколько сюрпризен:

```
y = fractions.Fraction(0.1)
print y

3602879701896397/36028797018963968
```

Ну, то есть, если посмотреть да призадуматься – то оно где-то так на так и выйдет, но хотелось бы чего-то другого, светлого и радостного – и однозначно чёткого. Причина понятна. Вы бесконечное количество раз встречали утверждение, что числа в компьютере хранятся в двоичном виде. Бесконечное количество раз вы это игнорировали, потому что – какая

разница? Это тот уникальный случай, когда разница есть. В натуральные дроби идеально преобразуются только те числа, которые представимы в натуральных дробях с знаменателем, являющимся степенью двойки. Обдумайте.

Далее мысль творцов Питона – или творца в единственном числе – разветвляется. Одна ветвь – сугубо программистская, другая – чисто математическая. Сначала о математике. Числа бывают натуральные, целые, рациональные, действительные. Всем этим математическим понятиям есть соответствия в Питоне. Математики, однако, на этом не остановились и выдумали комплексные числа. Что это такое? Введём обозначение $i = \sqrt{-1}$. Вам кто-то сказал, что из отрицательных чисел корни не извлекаются? Он не математик. В математике возможно всё, лишь бы оно не вступало в противоречие со всем, что создано предыдущими поколениями математиков. Теперь, когда мы ввели в обращение корень из минус единицы, введём и комплексные числа во всей их красе. Вот так они выглядят: $a + bi$, где a и b – самые обычные действительные (плавающие) числа. А что можно делать с комплексными числами? То же самое, что и с обычными. Сложение и вычитание: $(a_1 + b_1 i) + (a_2 + b_2 i) = (a_1 + a_2) + (b_1 + b_2)i$. Умножение чуть сложнее: $(a_1 + b_1 i) \times (a_2 + b_2 i) = (a_1 a_2 - b_1 b_2) + (b_1 a_1 + a_1 b_2)i$. Если я где-то в этой ерунде что-то перепутал, не надо меня судить слишком строго. Ошибки мои роли не играют – всё равно вместо нас все эти манипуляции производит Питон. Если что, там есть ещё и деление, но на это – заталкивание *таких* формул в текст – я уже не способен. И ещё один чисто терминологический момент. Если в $a + bi$ $a = 0$, то остаток от комплексного числа bi называется *мнимое* число. Если, наоборот, $b = 0$, то мы получаем хорошо знакомое, оно же действительное, оно же плавающее, число. Об этом нам придётся вспомнить чуть позже.

Теперь создадим два комплексных числа. Над глаголом *создадим* я долго думал. В традиционном языке программирования я бы написал *объявим*, но в Питоне переменные не объявляются, они именно что создаются.

```
x = 1 + 2j
y = 3 - 1j

plus = x + y
minus = x - y
mult = x * y
```

```
print 'x = ',x,'  y = ',y
print plus
print minus
print mult

x =  (1+2j)   y =  (3-1j)
(4+1j)
(-2+3j)
(5+5j)
```

Первое, на что стоит обратить внимание. Питон здесь, да и во многих других местах, несколько непоследователен и непредсказуем. Если дробные числа мы создавали явно и именно как дробные числа, то комплексные возникают как-то сами собой, неизвестно откуда. Хуже того, комплексных чисел в Питоне и нет как таковых. Странно, но это так. В Питоне есть только мнимые числа, то есть числа вида bi (это математическая запись), которые записываются в Питоне как bj, например $3j$. А комплексное число – всего-навсего сумма действительного числа и мнимого числа, то есть: $-2+3j$. Это не я придумал, это они.

Второе. Как-то так само собой получилось, что у нас во всех примерах исходные и расчетные комплексные числа состоят только из целых компонент. Разумеется, это не обязательно. Бывают и другие числа:

```
z = 2**0.5 + 3.14158j
print z

(1.41421356237+3.14158j)
```

Можно получить комплексное число с дробными составляющими и естественным путём – поделив X на Y.

```
div = x / y
print 'div = ', div

div =  (0.1+0.7j)
```

Кстати, для любознательных – хотя мнимое число это сколько-то корней из минус единицы, то есть $2+3i=2+3\times\sqrt{-1}$, извлечь корень из минус единицы в Питоне всё-таки нельзя. Вот такой парадокс.

Раздел получается довольно-таки длинным, но всё-таки, очень кратко ещё об одной разновидности чисел. Почему они вообще здесь, то есть в

Питоне, присутствуют? Не знаю, что называется – дань традиции. Только было это очень давно и в традиционных языках об этом давно уже забыли. Это называется *числа с фиксированной точностью.* Здесь опять начинается длинный и нудный и бесполезный разговор о том, что где-то внутри числа они все двоичные. Почти всегда, разговор этот ни о чём, программисту этого знать не нужно, но иногда, очень редко, *оно* вылезает.

Можно я опубликую здесь старый длинный замшелый советский анекдот? Можно? Спасибо. Он в точности описывает нашу ситуацию:

В исполком (сейчас – администрация) пришла жалоба: Напротив моего окна женская баня. Мне все видно и это отвлекает меня и вообще действует на мой моральный облик. Прошу предоставить мне новую квартиру.
Приехала комиссия, смотрят в окно.
— Ну и что? Ничего не видно!
— А вы на шкаф залезьте!
— Ну, залез, — говорит председатель, — всё равно не видно!
— Двигайтесь левее...
— Всё равно не видно!
— Ещё левее!
Председатель двигается левее и падает.
— Вот видите! А я так целый день!

К чему эта шутка? Пишем вот такой несложный код и получаем вот такой неожиданный результат:

```
a = 0.3 + 0.3 + 0.3 - 0.9
print a

-1.11022302463e-16
```

Далее непременно следуют знакомые разъяснения, что внутри компьютера числа хранятся в двоичном виде, что редкое десятичное число может быть записано в двоичном виде аккуратно и без мусора в хвосте и так далее и тому подобное. Всё это абсолютно верно, но продолжим опыты:

```
a = 0.3 + 0.3 - 0.6
print a

a = 0.4 + 0.4 + 0.4 - 0.8
```

```
print a

a = 0.5 + 0.5 + 0.5 - 1.5
print a

a = 0.15 + 0.15 + 0.15 - 1.45
print a

0.0
0.4
0.0
-1.0
```

Есть абсолютно достоверная история о не помню каком-то знаменитом физике-теоретике – то ли Эйнштейне, то ли Боре, то ли Гейзенберге. Физик-экспериментатор показал ему график, полученный в результате опытов, и попросил объяснить. Великий физик легко объяснил. Экспериментатор извинился и перевернул график вверх ногами. Великий физик поднапрягся, но успешно объяснил снова. Я не из великих – легко могу объяснить пример номер три и, с большим трудом, пример номер два. Кстати, если вы напишите код немного по-другому, то получите немного другой результат:

```
a = 0.4 + 0.4 + 0.4 - 1.2
print a

2.22044604925e-16
```

Моё мнение – всё это для программиста абсолютно неважно. Числа, с которыми он работает, если это реальные числа и получены из реальной жизни, всегда неточные. Точные числа бывают или в финансовых расчётах, или в чисто учебных задачах. Финансовые расчёты делаются финансовыми программами, это не ко мне. А вот для чисто не знаю каких, читай – учебных целей – Питон предлагает решение. Это не является изобретением Питона, это было давно. Но это было так давно, что всеми уже забыто – а в Питоне есть и теперь смотрится как новенькое. Это называется *числа с фиксированной точностью* – так длинно они называются в русских переводах, в английском оригинале они просто *Decimal*.

Возьмём наш первый нехороший пример и перепишем его для новых чисел:

```
import decimal
```

```
a = 0.3 + 0.3 + 0.3 - 0.9
print 'old.a = ', a

a = decimal.Decimal('0.3') + decimal.Decimal('0.3') +
decimal.Decimal('0.3') - decimal.Decimal('0.9')
print 'new.a', a

old.a =  -1.11022302463e-16
new.a 0.0
```

Что мы видим? Мы видим красивый, правильный и хороший ответ. С другой стороны, за этот хороший ответ нам пришлось заплатить приложением явно непропорциональных усилий – я не слишком сложно выражаюсь? Сначала нам явно пришлось попросить импортировать для нас модуль `decimal`. Мало того, там, где раньше мы писали просто `0.3`, теперь мы пишем `decimal.Decimal('0.3')`. И всё это только ради красивого ответа? Не совсем. Вся та дребедень, что справа, присвоена переменной `a`, которая слева. Теперь `a` тоже является числом с фиксированной точностью. Если вы попытаетесь написать что-то вроде

```
b = a + 0.33
```

то у вас ничего не получится – нельзя. Можно только вот так:

```
b = a + decimal.Decimal('0.33')
```

Число с фиксированной точкой – это как вирус, или, если это вам ближе, как вампир – кого укусил, тот тоже стал вампиром. Если в вычислениях задействовано число с фиксированной точностью, то и все дальнейшие вычисления должны содержать только числа с фиксированной точкой. В той области программирования, где я существую, это никому не нужно, но не могу же я говорить за всех.

Приложение С. Можно ли сделать EXE-файл?

Если вы написали программу, то судьба её может быть разной. Учебная программа, скорее всего, будет выброшена. Максимум, её текст будет оставлен в качестве образца, откуда можно списать, но выполняться программа не будет. Возможно, вы будете пользоваться программой сами. А может быть, этой программой будет пользоваться кто-то ещё. Понятно, что в этом случае программу надо этому кому-то передать. А как это сделать?

Когда-то давно были очень популярны Нортоновские Утилиты – Norton Utilities. Это был набор из пятнадцати-двадцати независимых программ, запускавшихся из-под единой оболочки. Самая сложная и важная из них называлось Disk Editor – Редактор Диска, с помощью которой можно было сделать *всё*. Был у нас сотрудник, попутно выполнявший функции сисадмина и ему эта программа была очень нужна. Поэтому сотрудник был очень рад, купив книгу по утилитам в целых двух томах. Он немедленно, предвкушая открытие сокровенного знания, открыл книгу на разделе о Редакторе. Там он прочёл следующее – *если вам нужна эта программа, значит, вам не нужна эта книга.*

Так вот, простейший путь выполнить программу на Питоне на чужом компьютере, это установить на чужом компьютере Питон. После этого чужой компьютер станет немного вашим и всё заработает. Если этот простой и естественный путь вас не устраивает, можно и по-другому.

Да, передать только один EXE-файл можно. Или почти один, с одной-двумя библиотеками. Но для этого надо скачать стороннюю программу. Сторонняя программа не в виде EXE, разумеется, она в виде исходных питонских кодов. Возможно, их придётся установить как модуль, возможно нет. Возможно, вам потребуется установить на компьютере Microsoft Visual Studio. Возможно, нет. Возможно, придётся произнести некоторые дополнительные заклинания и выполнить эротические танцы с бубном.

Что, безусловно, хорошо – все средства бесплатные.

Но я всё-таки решил отложить эту тему до следующей книги.

Приложение D. Философия Питона

Чтобы узнать всю философию Питона, достаточно в среде программирования – оболочке – набрать команду `import this`. В ответ вы получите следующее:

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Обратите внимание на две вещи. В оригинале это называется дзен Питона, но русские переводчики-программисты традиционно переводят *дзен* как *философия*. Во-вторых, команда одноразовая – второй раз она не сработает. Надо выйти и запустить IDLE снова.

Далее перевод. Мой. Не потому, что мой лучше, а чтобы не вляпаться в чьи-то авторские права.

Красивое лучше уродливого.
Сказанное лучше подразумеваемого.
Простое лучше сложного.
Сложное лучше усложнённого.
Цельное лучше составного.
Просторное лучше наполненного.
Пишите понятно.
Специальные случаи не настолько специальные, чтобы нарушать правила.
Хотя лучше, чтобы работало, чем сохранить невинность.

Ошибки должны вылезать наружу.
Если не заглушить их явно.
Если не понимаете – не пытайтесь угадать.
Должен быть один – и хорошо бы только один – способ сделать *это*.
Хотя способ бывает неочевиден – если ты не голландец.
видимо, тонкий намёк на легализацию легких наркотиков.
в стране вечных тюльпанов.
Сейчас лучше, чем никогда.
Одного танкового генерала спросили:
– Какой танк самый лучший?
– Тот, который здесь и сейчас, – ответил он.
Хотя никогда часто лучше, чем прямо сейчас.
нет, нет, нет, нет мы хотим сегодня.
Если реализацию трудно объяснить, то идея хреновата.
Если реализацию легко объяснить, то, может, и ничего.
Пространства имён обалденная идея – дайте две!
лично мне как-то безразлично.

Disclaimer, как говорится – я не со всем согласен.

Приложение Е. Все системы счисления на трёх страницах

Сначала математические разъяснения для тех, кто не знал да ещё и забыл – а что такое система счисления. Числа, с которыми мы обычно имеем дело, записаны в десятичной системе счисления. Это настолько само собой подразумевается, что на это даже не надо специально указывать. Обычно система счисления помечается мелкими циферками справа внизу, вот так: 123_{10}. Так положено, но, разумеется, всегда пишут просто 123. Что это означает? $123_{10} = 1\times10^2 + 2\times10^1 + 3\times10^0 = 123$. На всякий случай напоминаю, что *любое* число в нулевой степени всегда равно единице.

А если нам хочется систему счисления, в основании которой лежит восьмёрка? Сейчас это встречается редко, но раньше многие компьютеры работали именно в восьмеричной системе счисления. Подходим к замене десятки на восьмёрку чисто формально: $123_8 = 1\times8^2 + 2\times8^1 + 3\times8^0 = 64_{10} + 16_{10} + 3_{10} = 83_{10}$. Дальше, если имеется в виду именно десятичное число, я не буду подписывать систему счисления. Если система не указана – значит десять. Обратите внимание и подумайте – в восьмеричной системе гам не нужны цифры 8 и 9. Ведь в десятичной системе нет цифры, обозначающей величину 10.

Гораздо популярнее восьмеричной шестнадцатеричная система счисления. С ней программисту приходиться встречаться часто, где именно, пока уточнять не будем, поверьте на слово. Если в восьмеричной системе у нас обнаружились лишние цифры, то в шестнадцатеричной цифр явно не хватает. По этой причине вводятся новые обозначения: $A_{16} = 10_{10}; B_{16} = 11_{10}; C_{16} = 12_{10}; D_{16} = 13_{10}; E_{16} = 14_{10}; F_{16} = 15_{10}$. Обратите внимание, что цифры для шестнадцати нет, как нет в десятичной системе специальной цифры для десяти. Теперь очень легко сосчитать: $123_{16} = 1\times16^2 + 2\times16^1 + 3\times16^0 = 256 + 32 + 3 = 291$. Как я и обещал, если указания основания системы счисления нет, значит она десятичная. Ещё неплохо запомнить: $FF_{16} = 255$.

Двоичная система заметно менее употребительна, но иногда встречается. Опять-таки, честное слово, я знаю где и зачем она встречается, но для этой книги это лишнее.

Во многия знания многие печали © Екклесиаст

Как легко догадаться, в двоичной системе всего две цифры: ноль и единица – поэтому число наподобие 123 записать в этой системе просто невозможно. Перевод в десятичную систему выглядит так:
$1011111_2 = 2^6 + 0 + 2^4 + 3^3 + 2^2 + 2^1 + 2^0 = 64 + 16 + 8 + 4 + 2 + 1 = 95$.

Теперь от том, как всё это богатство записать в Питоне. Напоминаю, если раньше об этом не говорил – внутри все числа одинаковы, то, чем сейчас занимаемся, это всего лишь форма записи, мы выбираем ту форму, которая нам сейчас удобнее. В порядке возрастания основания системы счисления – двоичная, восьмеричная и шестнадцатеричная.

```
x2  = 0b1011111
x8  = 0o123
x16 = 0x123

print 'x2 = ',x2, '    x8 = ', x8, '    x16 = ', x16
```

Обратите внимание на тонкую разницу между нулём и буквой ***o*** во второй строке. Когда я был ещё маленьким программистом, программы мы писали на бумаге, на специальных бланках, и отдавали специальным девушкам, на перфорацию. Они пробивали текст на перфокартах. Чтобы девушки ничего не перепутали, нули полагалось перечёркивать – из правого верхнего угла в нижний левый.

А если всё это сложить? Двоичное, восьмеричное и шестнадцатеричное? Что будет? В какой системе счисления? Проверим:

```
xAll = x2 + x8 + x16
print 'xAll = ', xAll

xAll =  469
```

Безо всякой романтики, мы получили в ответе простое десятичное число. А если нам хочется красиво, не десятично, так, как и было? Легко, Питон о вас уже подумал.

```
print 'x2 = ', bin(x2), '    x8 = ',  oct(x8), '    x16 = ',  hex(x16)

x2 =  0b1011111     x8 =  0123     x16 =  0x123
```

Разобраться несложно, вы справитесь. Я уверен.

А теперь очень полезная таблица, которая очень пригодится любому настоящему программисту. И которую надо, страшно сказать, заучить наизусть.

$0000_2 = 00_{10} = 0_{16}$
$0001_2 = 01_{10} = 1_{16}$
$0010_2 = 02_{10} = 2_{16}$
$0011_2 = 03_{10} = 3_{16}$
$0100_2 = 04_{10} = 4_{16}$
$0101_2 = 05_{10} = 5_{16}$
$0110_2 = 06_{10} = 6_{16}$
$0111_2 = 07_{10} = 7_{16}$
$1000_2 = 08_{10} = 8_{16}$
$1001_2 = 09_{10} = 9_{16}$
$1010_2 = 10_{10} = A_{16}$
$1011_2 = 11_{10} = B_{16}$
$1100_2 = 12_{10} = C_{16}$
$1101_2 = 13_{10} = D_{16}$
$1110_2 = 14_{10} = E_{16}$
$1111_2 = 15_{10} = F_{16}$

Для чего это понадобится – для изучения файлов, что у них внутри. Для изучения памяти – что у неё внутри. Для изучения всего остального – что у него внутри. И вообще, это нужно почти всегда и почти везде – если всерьёз программировать.

Приложение F. Всё о битах и байтах

Есть такое понятие – единица измерения. В чём измеряется длина? Правильно, основная единица измерения длины метр. А чего-нибудь поменьше можно? Можно – дециметр, сантиметр, миллиметр. А где эта последовательность кончается? Я точно не знаю, надо посмотреть в интернетах. Ну не знаю я, какая самая маленькая единица измерения длины. И, уверен, вы тоже не знаете.

Есть такое понятие – информация. И у неё, информации, разумеется, тоже есть своя единица измерения. Только, в отличие от длины, у информации *есть* самая маленькая единица измерения и на её основе строятся единицы измерения побольше. Единицы измерения больше – можно, меньше – нельзя.

Эта единица измерения называется *бит*. Бит отвечает на вопрос – да или нет? Определение бита дал великий английский драматург Уильям Шекспир – *Быть или не быть*? *To be or not to be*? $2B \vee \neg 2B$?

Вопрос формулируется очень строго и чётко – день или не день? Нельзя спросить – день или ночь? А вдруг вечер или утро? Только вопрос, или да, или нет. Булевские переменные есть в большинстве языков. Эти булевские переменные отвечают именно на вопрос – да или нет? Содержат они ровно один бит информации. Поскольку вы уже чётко усвоили всё, касающееся систем счисления, нетрудно понять, что бит это одна двоичная цифра. Имеем:

$0_2 = False$

$1_2 = True$

Но бит настолько маленькая единица измерения информации, что нет переменных, которые содержали бы ровно один бит. Булевские переменные есть, а содержащих один бит переменных нет. Вот такой парадокс.

Основной единицей является не бит, а *байт*. Байт – это восемь бит или, что то же, восемь двоичных цифр. Что содержат байты? В первую очередь, целые числа, причём не просто целые, а беззнаковые целые, иначе говоря,

натуральные. Формат хранения их прост и незатейлив – просто пишем число в двоичном виде, вот так:

$00000000_2 = 0_{10}$

$00000001_2 = 1_{10}$

$11111111_2 = 255_{10}$

Ничего большего, чем восемь единиц, поместить в байт нельзя. Значит 255 – максимальное число, которое байт может выразить. Двоичная запись длинная – это плохо. Что ещё хуже, трудно не ошибиться в подсчёте ноликов и единичек. Надо что-то делать. Как нетрудно заметить, восемь это два раза по четыре, а четыре двоичные цифры в точности соответствуют одной шестнадцатеричной. Результат:

$00000000_2 = 00_{16}$

$00000001_2 = 01_{16}$

$11111111_2 = FF_{16}$

Именно в такой форме обычно представляют то, что содержит память, что записано в файле, что передано по каналу связи. В тех, разумеется, случаях, когда вам точно надо это знать. Итак, один байт может хранить целое число в диапазоне 0..255. Немедленно возникают два вопроса. А если нам нужны отрицательные числа? А если нам этого диапазона мало – а его всегда мало?

Ответ на первый вопрос. Ответ на вопрос – положительное или отрицательно – содержит ровно один бит информации, или одну двоичную цифру. Если из восьми двоичных цифр одна ушла на кодировку знака, значит, диапазон нашего числа сократился в два раза. Это понятно. Что сложнее, запись положительных чисел производится как обычно, а вот запись отрицательных – в так называемом дополнительном коде. Обрабатывать их так удобнее, а вот понять глазами, что там записано – сложнее.

Ответ на второй вопрос. Целые числа из одного байта есть во всех традиционных языках, но только как вспомогательный инструмент. Когда-то давно стандартом было два байта, теперь четыре. То есть если у вас есть целая переменная – чуть не сказал, если мы объявили целую переменную – то в ней будет четыре байта, причём со знаком. Какой диапазон она сможет вместить, посчитайте сами. Впрочем, бывают целые числа и длиннее.

Теперь вернёмся к булевским переменным. Хотя в теории она должна занимать только один бит, никто не мелочится. В большинстве языков под неё выделяется один байт. Понимается это так – если ноль, то ложь, если единица, то истина. Это стандартно, но обычно к процессу подходят более гибко – если не ноль, а что угодно, то истина.

Формат записи плавающих чисел сложный и очень сложный. Ни разу не встречал человека, который мог бы, взглянув на плавающее число в шестнадцатеричном виде, определить, что в нём хранится. Хуже того, этих форматов минимум два, в смысле минимум два распространённых. Стандартное плавающее число занимает четыре байта, но бывают и больше.

О строках. Здесь грустно. Раньше было не очень грустно – один символ это один байт. Теперь не угадаешь, может и два. Называется это словом *Юникод*. Хорошая новость – закодировать можно почти любой символ, включая китайские иероглифы. Кстати, базовый китайский компьютерный стандарт включает 6900 иероглифов. Расширенный – плюс ещё столько же.

Плохая новость – по внешнему виду не угадаешь, какая кодировка, традиционная или Юникод. Чтобы всё запутать окончательно, надо наложить на это отсутствие стандартного метода записи строк в файл и насладиться результатом. Что там за ужас с хранением строк в памяти, я упоминать не буду.

Теперь очень важное понятие – *слово*. Относится оно, скорее, к железной составляющей компьютера. Процессор не может обратиться к отдельному биту. Он и к отдельному байту обратиться не может. Он может обратиться только к слову. Обычно слово – четыре байта. Что отсюда следует для простого программиста? Использование вместо переменной в четыре байта аналогичной переменной в один байт экономит место, но, скорее всего, не экономит время. Обработка четырёхбайтового целого займёт ровно столько же времени, сколько и однобайтового. Разумеется, есть вариант – вы можете пожертвовать *своим* временем и написать какую-то хитрую обработку по четыре однобайтовых числа за один раз. Оно вам надо?

Более крупные единицы измерения информации служат только и именно для измерения её количества. То есть объёма файлов, памяти, дисков и так далее. Ни с какими простыми переменными они не соотносятся. Нет такой простой переменной, которая занимала бы килобайт. Сложных – сколько угодно. Массив или список легко займут хоть килобайт, хоть мегабайт.

Килобайт это 1024 байта, что равно 2^{10} . С математической или программистской точки зрения, смесь двоек и десяток выглядит как-то неорганично. Скорее всего, кому-то хотелось иметь какой-то аналог весового килограмма. Единицы крупнее:

Мегабайт = 2^{10} килобайт = 2^{20} байт

Гигабайт = 2^{10} мегабайт = 2^{20} килобайт = 2^{30} байт

Терабайт = считайте сами

Килобайт – это не тысяча байт, это чуть больше. Хорошо это или плохо? Сейчас всем всё равно, а раньше регулярно возникали ситуации. Покупает человек за деньги, к примеру, жёсткий диск, к примеру, ёмкостью 500 мегабайт. Устанавливает его в компьютер и обнаруживает, что мегабайт только 476 с чем-то. Украли! В каком-то смысле да, украли. Продавцы жёстких дисков считали один мегабайт за миллион байт, а компьютер не обманешь, он требовал именно 2^{20}.

А в попугаях-то я длиннее буду!
© кстати, Питон, из мультфильма «*38 попугаев*»

И да, я знаю, что всё не так просто – и с байтами, и с терабайтами, и со словами, и со строками. Особенно, конечно, со строками.

Приложение G. Обмен данными с другими программами через файлы

Логично завершить книгу, обсудив, как стыковать программы на Питоне с программами на других языках или как написать единую программу частично на Питоне, а частично на C++ или Дельфи. Написать одну программу на нескольких языках сложно. Обмениваться данными между двумя программами, написанными на разных языках, много проще. Самый простой способ – обмениваться данными через файлы. Это медленно, это не очень надёжно, это негибко – но это очень просто и очень понятно. Этим и займемся.

И повторюсь – эта, последняя в книге задача, ни разу не учебная. Это задача реальная и абсолютно необходимая.

Опять-таки, если вас интересует только и исключительно программирование на Питоне и только на нём, этот раздел можно смело пропустить. Если в ваших планах участие в больших проектах, где задействованы разные среды и языки программирования, то этот раздел лучше прочитать.

Это единственный раздел изо всей книги, где приводится пример на другом языке. Цель этого раздела – продемонстрировать, как наши файлы из Питона будут читаться в другом языке и, главное, какие при этом могут возникнуть проблемы. В качестве другого языка будет использован Delphi седьмой версии. Раньше в среде Дельфи использовался язык программирования Object Pascal, теперь нет. Нет, он никуда не делся, его легко и непринужденно переименовали в язык Delhi. Следующие, после седьмой, версии загромождали бы изложение наличием двухбайтовой кодировки символов.

Запишем бинарный файл в Питоне, это мы умеем, затем прочитаем в Дельфи. Содержание файла:

количество целых чисел
    сами целые числа в соответствующем количестве
одно плавающее число
две строки

Программа на Питоне с комментариями, которые будут длинными:

```
import struct
#-----------------------------------------------
def strToFile( S, F):
    length = len(S)

    data = struct.pack( '<i4', length)
    F.write(data)

    for i in xrange(0,length):
        ch = S[i]
        data = struct.pack( '<s1', ch)
        F.write(data)
#-----------------------------------------------

f = open( 'd:/py.bin', 'wb')

numOfInts = 10
data = struct.pack( '<i4', numOfInts)
f.write(data)

for i in xrange(0,numOfInts):
    what = i*2 + 1
    data = struct.pack( '<i4', what)
    f.write(data)

x = 0.99**100
data = struct.pack( '<f4', x)
f.write(data)

stroka_1 = 'And the silken sad uncertain rustling of each purple curtain'
strToFile( stroka_1, f)

stroka_2 = 'Эх раз, ещё раз, ещё много много раз'
strToFile( stroka_2, f)

f.close()
```

Запись строки в файл оформлена в виде функции – повторно писать почти один и то же код как минимум дважды откровенно лень. Неплохо бы оформить аналогичным образом и запись в файл для других типов переменных. Займитесь этим. Вы сосчитали в уме, сколько будет `x = 0.99**100`, то есть $x = 0.99^{100}$? Я сосчитал и ошибся всего в полтора раза. Попробуйте и вы.

Теперь самый важный вопрос. Почему строки пишутся именно так, как они пишутся здесь? Ответ – а как? Универсального способа не *вообще*.

Стандартный питонский способ через формат s не подходит – читающая программа мало того, что не догадается, что перед ней строка. Даже если каким-то чудом она это поймёт, ей неизвестно будет, сколько в строке символов. Второй формат, который p, кажется более перспективным. Ведь он специально для совместимости с Паскалем, который, как всем известно, то же Дельфи, только в профиль. Но есть нюанс. Или два. Мы можем записать – и прочитать – таким способом только короткие строки, до 255 символов. И, само собой, другим языкам программирования этот формат совершенно неизвестен.

Поэтому мы будем писать строки в файл в формате, который хотя и не универсален, но понятен и встречается часто. Сначала длина строки как четырёхбайтовое целое. Затем сами символы, по байту на символ. Точно так же, как и в Питоне, чтение строки оформим в виде функции. Вот программа для чтения:

```
   var
      F                          : file;
      numOfInts                  : integer;
      int                        : integer;
      fl                         : single;
      stroka_1, stroka_2         : string;
      i                          : integer;
{.....................................................}
function strFromFile( var F : file) : string;
   var
      length                     : integer;
      ch                         : char;
      i                          : integer;
begin
     BlockRead( F, length, 4);

     result:='';
     for i:=1 to length do begin
        BlockRead( F, ch, 1);
        result:=result + ch;
     end;
end;
{.....................................................}
begin
     AssignFile( F, 'd:\py.bin');
     ReSet( F, 1);

     BlockRead( F, numOfInts, 4);
     for i:=1 to numOfInts do begin
        BlockRead( F, int, 4);
        ShowMessage(IntToStr(int));
```

```
    end;

    BlockRead( F, fl, 4);

    stroka_1:=strFromFile(F);
    stroka_2:=strFromFile(F);

    CloseFile(F);
end;
```

Не будем обсуждать сравнительные достоинства и недостатки Питона и Паскаля и вообще как-то комментировать программу. Единственное, что отмечу – если в питонской программе целые числа возникали из ниоткуда, то в паскалевской они исчезают в никуда – выводятся на экран, причём каждое – в отдельное окно отдельным сообщением. Это я к тому, что эта часть обеих программ совершенно условна и в реальной практике должна быть заменена на что-то осмысленное.

Теперь обратная задача. Запишем файл в Паскале, только проще: целое, плавающее, строка. Запись строки, разумеется, оформим как процедуру.

```
   var
      len                    : integer;
      ch                     : char;
      i                      : integer;
begin
    len:=Length(s);
    BlockWrite( F, len, 4);

    for i:=1 to len do begin
       ch:=s[i];
       BlockWrite( F, ch, 1);
    end;
end;
{.....................................................}
begin
    AssignFile( F, 'd:\pas.bin');
    ReWrite( F, 1);

    int:=12345;
    BlockWrite( F, int, 4);

    fl:=123.45;
    BlockWrite( F, fl, 4);

    stroka:='И много много радости детишкам принесла';
    strToFile( F, stroka);

    CloseFile(F);
```

```
end;
```

Обратите внимание на переименование переменной из `length` в `len`. Первое имя в Паскале – имя встроенной функции, а второе – нет. В Питоне наоборот. А теперь чтение в Питоне, тоже совершенно симметричное:

```
# -*- coding: cp1251 -*-
import struct

def strFromFile(F):
    result = ''
    data = F.read(4)
    length = struct.unpack('<i4',data)

    for i in xrange(0,length[0]):
        data = F.read(1)
        ch = struct.unpack( '<s1', data)
        result = result + ch[0]

    return result

F = open( 'd:/pas.bin', 'rb')

data = F.read(4)
int = struct.unpack( '<i4', data)
print int

data = F.read(4)
fl = struct.unpack( '<f4', data)
print fl

stroka = strFromFile(F)
print stroka

F.close()
```

Напоминаю – при чтении из бинарного файла данные читаются не просто так, а в кортеж. Потом их приходится оттуда выковыривать. Это относится только к Питону.

Издательство «СОЛОН-Пресс» представляет

Заказывайте на сайте издательства

www.solon-press.ru

ООО «СОЛОН-Пресс»
123001, г. Москва, а/я 82
Телефоны: (495) 617-39-64, 617-39-65
E-mail: kniga@solon-press.ru,
www.solon-press.ru

ISBN 978-5-91359-334-4