Чаплыгин А. Н.

Учимся программировать вместе с Питоном

(Start with Python)

Revision: 226

Содержание

Предисловие

Книга получилась немного сложнее, чем планировалось: в целом она соответствует уровню старших классов и начальных курсов ВУЗов, но и младшеклассники при поддержке преподавателя тоже смогут без особого труда освоить материал книги.

Благодарности

Данная книга распространяется так же свободно, как и устная речь. Поэтому она так же свободно развивается, без ограничений какими-либо лицензиями, кроме OPL. Эта книга появилась во многом благодаря вкладу людей, перечисленных в данном списке:

- Шляков Дмитрий
- Бельченко Александр
- Рудских Вячеслав
- Школьников Дмитрий
- Волошин Евгений
- Бречалов Дмитрий
- Откидач Денис
- Ситников Илья
- Олищук Андрей
- Чуранов Михаил
- Пушилин Сергей
- Литвинова Елена
- Липанин Антон
- Жуков Борис

Большое спасибо энтузиастам движения свободного программного обеспечения (http://opensource.org/), живущим по всему миру, за то, что они делают.

Отдельная благодарность проектной команде открытого офисного пакета OpenOffice.org (http://OpenOffice.org/, http://OpenOffice.ru/) за хороший продукт, в котором и была написана эта книга.

Введение

В школе учительница спрашивает учеников, кем работают их родители. Руку тянет девочка:
– У меня папа доктор.
– Это хорошая профессия, – говорит учительница.
– А у меня папа водитель, – хвастается один из мальчишек.
– Какая интересная профессия. А чем твой папа занимается? – спрашивает учительница Вовочку, на что тот невозмутимо отвечает:
– Он в борделе на пианино играет.
Учительница в шоке; в тот же день она идет к Вовочке домой и возмущенно говорит отцу:
– Как вы можете в таких условиях воспитывать ребенка?! Вы действительно играете на пианино в борделе?!
Папа Вовочки смущенно:
– Видите ли, я программист. Специализируюсь на сетях TCP/IP, пишу распределенные сетевые приложения для операционных систем UNIX, но как это объяснить семилетнему мальчугану?

Профессия программиста (да и любого другого компьютерного специалиста) со стороны, быть может, выглядит несколько загадочно. «Чем эти люди занимаются? Сидят целыми днями у компьютера и рассказывают друг другу странные анекдоты» – еще недавно так рассуждало большинство. Но на сегодняшний день информационные технологии проникли практически во все сферы деятельности человека: от таких уже привычных вещей, как мобильные телефоны, до космических технологий. Компьютеры упрощают работу с документами и помогают оптимизировать бизнес-процессы. Благодаря ничем не приметным людям, просиживающим за компьютером ночи напролет, мы можем общаться в реальном времени с друзьями и бизнес-партнерами, находящимися в любой точке мира с помощью интернета. Дзен-буддисты говорят: «Работы мастера не видно».

Данная книга ставит целью научить не просто писать программы, но и думать как компьютерный специалист. Подобный способ мышления сочетает в себе подходы, используемые в математике, естественных науках и инженерном деле. Подобно математикам, компьютерные специалисты используют формальные языки для записи идей, алгоритмов и операций над специфическими объектами. Подобно инженерам, они проектируют объекты, собирая различные компоненты в системы, и выбирают решения из возможных альтернатив. Подобно естествоиспытателям, они изучают поведение комплексных систем, строят гипотезы, ставят эксперименты.

Отдельный наиболее важный навык компьютерного специалиста – умение решать задачи. Этот навык подразумевает способность их формулировать, творчески подходить к поиску возможных решений и четко излагать выбранный вариант. Как оказывается, процесс обучения программированию является прекрасной возможностью выработать все эти умения.

Итак, приступим. Подробное описание процесса установки интерпретатора Питона и всего, что вам потребуется для работы с книгой, можно прочесть в следующем разделе. Рекомендуется ознакомиться с ним прежде, чем переходить к изучению первой главы, чтобы

у вас не возникало проблем в процессе освоения материала книги. Но не забывайте, что главное – это желание учиться, умение задавать вопросы и искать на них ответы.

§1. Базовые знания

Для того, чтобы освоить материал данной книги, от вас потребуются базовые навыки работы с компьютером, а именно:

- Работа с файлами и папками (директориями);
- Запуск программ;
- Редактирование текстовых файлов;
- Если у вас на компьютере установлена UNIX-система, то будет очень полезно уметь работать в консоли;
- Работа в интернете – там можно найти много полезной информации;
- Очень пригодится хотя бы базовое знание английского языка.

Имейте ввиду, что офисные пакеты (MS Office, OpenOffice.org, StarOffice и им подобные) в программировании вам не помогут. Программы набираются в простых текстовых редакторах типа MS Notepad (он же Блокнот).

Хорошо, если вы знаете, что такое двоичный код и умеете переводить числа из одной системы счисления в другую, всему этому обычно учат в школьных курсах информатики.

Неважно, какая операционная система установлена у вас на компьютере – Питон имеет реализации под все самые распространенные платформы: Windows, UNIX (GNU/Linux, FreeBSD и др.) и даже для Mac OS X. Более того, программы, написанные в одной операционной системе, будут успешно выполняться в любой другой при наличии установленного интерпретатора Питона!

Питон входит в комплект поставки большинства дистрибутивов GNU/Linux, но не факт, что это последняя версия. Поэтому стоит все-таки заглянуть на страницу Питона и ознакомиться с новостями. Питон, как и большинство проектов сообщества open source, развивается очень динамично.

Все примеры в данной книге проверены в интерпретаторе Питона версии 2.4.3.

§2. Где достать интерпретатор языка Питон?

Интерпретатор языка Питон распространяется свободно на основании лицензии Python Software Foundation (PSF) Licence (http://www.python.org/psf/license/), которая, в некотором роде, даже более демократична, чем GNU GPL (GNU General Public License: http://gnu.org/copyleft/). Официальный сайт проекта языка Питон располагается по адресу http://python.org/. Здесь же в разделе «Downloads» можно скачать свежую версию для вашей операционной системы.

Процесс установки интерпретатора зависит от того, какой операционной системой Вы пользуетесь. Будем считать, что вы в состоянии установить Питон самостоятельно. При возникновении проблем задавайте вопросы своим более опытным товарищам или на форумах в интернете.

§3. Среда разработки

В стандартный комплект поставки Питона входит интегрированная среда разработки IDLE, в которой редактировать программы будет намного удобнее, чем в простом текстовом редакторе. IDLE написан на Питоне с использованием платформонезависимой библиотеки

Tcl, поэтому легко запускается в любой операционной системе, для которой существует реализация Питона. IDLE также имеет встроенную систему отладки, позволяющую запускать программу построчно, что облегчает процесс поиска ошибок. Если по какой-то причине IDLE вас не устраивает, то можете попробовать другие среды разработки.

В UNIX-системах есть множество редакторов, имеющих свои прелести и недостатки, как консольные (vi, emacs, встроенный редактор mc), так и графические (vim, emacs, kate, редактор IDLE и др.).

Для начала самым простым выбором может стать Kate – он входит в комплект поставки оконной среды KDE последних версий. Kate поддерживает множество кодировок, умеет подсвечивать синтаксические конструкции программы, имеет встроенный эмулятор консоли и специальный модуль, облегчающий работу с программами, написанными на Питоне: браузер кода, который включается через меню «Настройка» в разделе «Приложение → Модули».

На сайте языка Питон есть страница со списком сред разработки, которые вам могут подойти: http://wiki.python.org/moin/IntegratedDevelopmentEnvironments. При выборе среды разработки стоит обращать внимание на следующие вещи:

1. Поддержка кодировки UTF-8
2. Подсветка синтаксиса
3. Способность текстового редактора проставлять автоотступы

Это тот минимум, который нам понадобится на первых порах. В последствии, общаясь с более опытными программистами, вы, наверняка, сможете выбрать что-то более удобное и мощное, но этот выбор будет уже осознанным.

И, наконец, для запуска программ в Windows очень пригодится файловый менеджер FAR – из него гораздо удобнее запускать консольные программы, чем из стандартного окна командного интерпретатора Windows. В ОС Windows FAR, пожалуй, является наиболее подходящей средой для запуска программ на Питоне.

§4. Запуск программ, написанных на Питоне

§4.1. В UNIX-подобных ОС

Для того, чтобы запустить программу, написанную на Питоне, в UNIX-системах необходимо вызвать интерпретатор Питона и передать ему в качестве параметра название файла, в котором находится запускаемая программа:

```
$ python my_program.py
```

Кроме того, в операционных системах UNIX есть возможность в программе указать, какой интерпретатор необходимо вызвать для ее выполнения. Для этого в первой строке программы нужно написать:

```
#! /usr/bin/env python
```

Затем нужно сделать файл со скриптом исполняемым:

```
$ chmod u+x my_program.py
```

После этого скрипт можно будет выполнять, просто набрав в командной строке его имя:

```
$ my_program.py
```

или, если первый вариант не работает:

```
$ ./my_program.py
```

Последовательность символов `#!` программисты обычно читают как «she-bang!».

§4.2. В ОС Windows

В Windows программы, написанные на Питоне запускаются привычным способом – найдите при помощи Проводника файл, содержащий программу, и дважды щелкните на его иконке левой кнопкой мыши. Питон при установке связывает расширение файла `.py` с интерпретатором Питона, поэтому при двойном щелчке на таком файле будет запущен Питон, который выполнит программу, записанную в файле. Если Ваша программа не имеет графического интерфейса, то результаты работы программы будут выводиться в консольное окно. После завершения работы программы это окно будет сразу же закрыто, поэтому вы можете не успеть увидеть результат ее работы. Избежать этого можно, добавив в конце программы следующую строку:

```
raw_input("Press any key to exit")
```

Это заставит интерпретатор дождаться нажатия клавиши `[Enter]`, прежде чем завершить программу.

Если же вы задали другое расширение, то метод запуска двойным щелчком не сработает. В Windows питон-программы всегда должны иметь расширение `.py` или `.pyw`. Расширение `.pyw` используется для программ, использующих графический интерфейс. Для интерпретации таких программ используется оконный вариант интерпретатора Питона.

Другой вариант – это открыть окно командного интерпретатора (или запустить FAR) и выполнить следующую команду:

```
C:\Examples> python my_program.py
```

Этой командой мы запускаем интерпретатор Питона и указываем ему, программу из какого файла он должен выполнить.

Теперь мы во всеоружии и готовы приступить к изучению программирования на языке Питон. Переходим к следующей главе, в которой рассматриваются базовые понятия.

Глава 1. Базовые понятия

Для начала придется разобраться с некоторыми базовыми понятиями. Не стоит их заучивать – достаточно их понять хотя бы на интуитивном уровне. Позднее вы начнете использовать их на практике, и все встанет на свои места. Это, пожалуй, одна из самых утомительных частей книги.

§1.1. Алгоритмы и программы

Понятие алгоритма является одним их центральных понятий всей компьютерной дисциплины. Слово «алгоритм», в сущности, является синонимом слов «способ» или «рецепт». Можно говорить, в этом смысле, об алгоритме нахождения корней уравнения по его коэффициентам, или об алгоритме разложения натурального числа на простые множители. Если в основе алгоритмов лежат простые вычисления, то такие алгоритмы называют численными. Впрочем, довольно часто рассматриваются и нечисленные алгоритмы. Например, в роли исходных данных и результатов могут выступать последовательности символов: тексты, формулы и т.д. В роли операций – не привычные операции сложения, умножения и подобные им, а операции сцепления строк или операции замены одних символов на другие по некоторой таблице соответствий. Примером может служить кодирование текста азбукой Морзе. Существуют алгоритмы построения сложных графических объектов и их преобразования. Для того, чтобы научить компьютер что-то делать, нужно предварительно составить алгоритм.

Алгоритм – это описанный со всеми подробностями способ получения результатов, удовлетворяющих поставленным условиям, по исходным данным.

Программа – это последовательность машинных инструкций, описывающая алгоритм. Разумеется, для того, чтобы написать программу, нужно придумать алгоритм. Компьютерные программы обычно составляются на специальных языках программирования.

§1.2. Языки программирования и уровни абстракции

Существует несколько подходов к программированию. Изначально вычисления описывались на уровне машинных команд в двоичном коде. Логику подобных программ было довольно трудно уловить из-за того, что программисту приходилось уделять внимание таким вопросам, как, например, сколько ячеек памяти необходимо выделить для хранения того или иного значения. Для сложения двух чисел необходимо было предварительно вычислить адреса ячеек памяти, в которых хранились складываемые значения, и только после этого произвести операцию сложения двоичных чисел. Такой подход к программированию иногда называют адресным.

Прочесть и разобраться, как работает программа, написанная в двоичных кодах, было очень сложно, не говоря уже о том, чтобы найти и исправить в ней ошибку. Поэтому для упрощения своей работы программисты придумали *мнемокоды* или *мнемоники* (от греч. Mnemonikos ← mnemon – запомнить) – буквенные обозначения машинных двоичных команд, которые проще запомнить, чем последовательности нулей и единиц. Для упрощения работы с ячейками памяти стали использовать понятие переменной.

Переменная – в программировании это буквенное обозначение области памяти, в которой хранится некоторое значение.

Для перевода мнемокодов в машинные инструкции и имен переменных в адреса ячеек памяти использовалась специальная программа – транслятор. Языки мнемо-кодов получили название ассемблеров.

Технологии разработки продолжали развиваться, специалисты искали новые подходы и вскоре стали выкристаллизовываться идеи, которые впоследствии легли в основу так называемого структурного подхода. Было замечено, что все вычисления сводятся следующим элементарным действиям:

- Ввод данных с клавиатуры, из файла или с какого-либо устройства;
- Вывод данных на экран, в файл, на принтер или другое устройство;
- Выполнение некоторых операций над числами, строками или другими объектами;
- Выбор ветви выполнения программы на основе принятого решения (например, по результатам сравнения двух значений);
- Повторение группы операций чаще всего с изменением одного или нескольких параметров.

Скоро вы увидите, что это действительно так. Параллельно начали появляться новые трансляторы, которые преобразовывали в машинные команды программы, написанные на языках, основанных на этих базовых операциях. Такие языки стали называть структурными или языками высокого уровня. Программы на высокоуровневых языках описывают вычислительный процесс на более высоком уровне абстракции, т.е. позволяют программисту абстрагироваться от особенностей машинной реализации.

Проиллюстрируем это двумя простыми примерами. Первая программа, имитирует низкоуровневый язык:

```
Занести в регистр edx адрес первого числа
Занести в регистр-счетчик 10
Занести в регистр eax 0
Метка lp:
Сравнить значение регистра eax со значением, хранящимся по адресу
взятому из регистра edx
Если оно меньше или равно, то перейти к метке l_next
Занести в регистр eax значение из ячейки, адрес которой хранится
в edx
Метка l_next:
Прибавить к значению регистра edx 4
Если в регистре-счетчике не 0, то уменьшить значение в регистре-
счетчике на 1 и перейти к метке lp
Записать в переменную Max значение из регистра eax
```

Вторая программа имитирует высокоуровневый:

```
Занести в переменную Max 0
Повторять пока переменная i изменяется от 1 до 10 с шагом +1:
Если i-е число больше Max, то
   занести его в переменную Max
Вывести значение переменной Max.
```

Обе программы находят максимальное из десяти заданных натуральных чисел. Названия операций изменены на русскоязычные аналоги (такой способ записи иногда называют псевдокодом), но смысл действий сохранен, так что понять их различие несложно.

Чаще всего программы, написанные на языках высокого уровня работают медленнее, т.к. трансляторы строят не самый оптимальный машинный код по тексту программы. Но зато языки высокого уровня обладают многими преимуществами: их легче читать и понимать, кроме того, они имеют свойство переносимости. Это означает то, что высокоуровневые программы могут выполняться на различных типах компьютеров или под управлением различных операционных систем, причем с минимальными изменениями или вовсе без таковых, в то время как низкоуровневые программы обычно пишутся под определенный тип компьютеров или операционную систему, и для переноса таких программ на другую платформу их приходится переписывать.

Обратите внимание на то, что в первой программе результат не выводится. Способ вывода в низкоуровневой программе будет разным в зависимости от платформы, на которой будет выполняться данная программа. Более того, на процессоре Intel 80286 эта программа работать тоже не сможет, т.к. регистры `edx` и `eax` появились только в Intel 80386.

Классическими структурными языками являются C[1] и Pascal. Они позволяют описывать вычислительные процессы на более высоком уровне абстракции, чем ассемблеры, но, тем не менее, им присущ ряд недостатков. В частности, программисту, пишущему на этих языках, все же приходится явно указывать тип переменных, с которыми будут работать его программы, для того, чтобы транслятор знал, сколько памяти выделять под их хранение. Кроме того, в переменных предназначенных для хранения целых чисел невозможно сохранить, например, строку.

В последнее время стали приобретать популярность так называемые языки очень высокого уровня. Они позволяют абстрагироваться от типов переменных, что открывает новые возможности. С ними мы познакомимся в процессе изучения языка Python, который является одним из представителей этой группы языков программирования.

§1.3. Формальные и естественные языки

Разберемся, чем языки программирования отличаются от нашего родного языка, на котором мы разговариваем с детства. Существует два вида языков: естественные и формальные.

К естественным относятся языки, на которых разговаривают люди: русский, английский, французский, арабский и другие. Скорее всего, они возникли естественным путем, когда в древности люди пытались друг другу что-то объяснить. Все они довольно сложные, хотя мы часто этого просто не замечаем. Для того чтобы упростить изучение иностранных языков, люди придумали правила и словари, в которых словам одного языка приводятся соответствия из других языков. Самыми сложным естественным языком считается санскрит: первый сборник правил санскрита (грамматика Панини, называемая «Восьмикнижие», IV век до н.э.) содержал более 4000 грамматических правил. В санскрите 8 падежей, 3 числа в именах, несколько сотен глагольных и отглагольных форм, имеются средства свободного образования многочисленных производных слов. Кроме того, в санскрите можно найти до нескольких десятков слов, символизирующих один и тот же объект, но отражающих различные смысловые оттенки, поэтому его выразительные возможности чаще всего превосходят средние потребности.

Один из самых простых языков, на которых можно разговаривать, это эсперанто. Эсперанто придумал врач-офтальмолог Лазарь (Людовик) Маркович Заменгоф в XIX веке.

1 Читается как «си».

Уже в детстве у Заменгофа появилась идея о том, что один общий язык помогал бы народам лучше понимать и больше уважать друг друга.

У Заменгофа были отличные способности к языкам, и еще школьником кроме родных русского, польского и идиша он изучил немецкий, французский, английский, латынь, древнегреческий, древнееврейский – всего около четырнадцати языков. Вскоре он убедился, что ни древние, ни современные языки не годятся в качестве общего. Тогда он задумал создать новый язык, который не принадлежал бы ни одному народу, был бы легок для изучения, но в то же время не уступал бы национальным языкам в гибкости и выразительности. И это ему удалось – вся грамматика эсперанто (вместе с фонетическими правилами) умещается на двух печатных страницах, все слова международные, а по выразительности эсперанто не уступает (и в чем-то даже превосходит) русский. Esperanto estas tre interesa kaj esprima lingvo.

Можно ли эсперанто назвать естественным языком? Пожалуй, нет. Несмотря на то, что он живет своей жизнью и развивается естественным путем, грамматика эсперанто по-прежнему жестко формализована. Поэтому он, скорее, относится к формальным языкам.

Формальными называют языки, придуманные людьми для решения специфических задач. Например, формальным языком является набор специальных знаков и правил для записи математических формул. Химики так же используют свой формальный язык для записи химической структуры веществ. **Языки программирования – формальные языки, предназначенные для описания алгоритмов.**

Формальные языки характерны тем, что имеют четкие синтаксические правила. Например, `3+3=6` является синтаксически правильной математической записью, а `3=+6$` – нет. H_2O – синтаксически правильная химическая формула вещества, а $_2Zz$ – нет.

Когда вы читаете предложение на русском языке или выражение на формальном языке, вы определяете его структуру, часто неосознанно. Этот процесс называется *синтаксическим анализом* или *синтаксическим разбором*. Эквивалентный англоязычный термин – *parsing*.

Например, когда вы читаете фразу «Мама мыла раму», вы по пробелам определяете начало и конец слов и только после этого находите подлежащее («мама») и сказуемое («мыла»). Разобрав синтаксическую структуру, вы можете понять ее смысл – семантику.

Любой транслятор перед тем, как преобразовать программу в понятный для компьютера вид, выполняет синтаксический анализ. При синтаксическом анализе транслятор разбирает синтаксическую структуру выражений, и находит так называемые *символы* (*tokens*) – синтаксически неделимые части. В данном контексте символами могут быть названия переменных, числа, знаки операций, ключевые слова, обозначения химических элементов.

Использование символа `$` в формуле `3=+6$` не имеет смысла, и это является одной из причин, почему оно неверно с точки зрения математики. Та же проблема в «химической» формуле $_2Zz$: в таблице Менделеева нет элемента с обозначением `Zz`.

Второй тип синтаксических ошибок связан с неправильной структурой выражений, т.е. последовательностью символов. Выражение `3=+6$` имеет неверную структуру, т.к. сразу после знака равенства не может следовать знак сложения. Аналогично, в молекулярных формулах используются нижние индексы, но они не могут идти перед обозначением химического элемента.

Хотя формальные и естественные языки имеют много общего, они имеют ряд важных отличий:

1. *Однозначность*

   В естественных языках множество идиом и метафор; часто люди определяют

значение фраз в зависимости от ситуации, в которой они используются. Формальные языки разработаны так, чтобы исключить неоднозначность выражений. Это означает, что выражение должно иметь только одно значение вне зависимости от контекста.

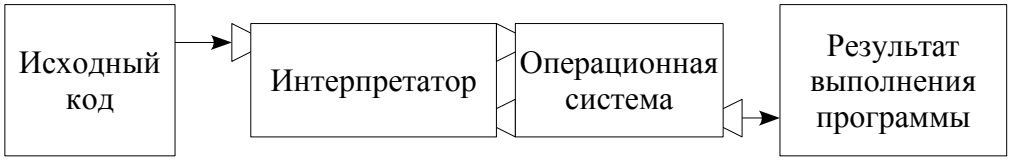
2. *Избыточность*
   Для того чтобы избавиться от неоднозначности и избежать недопониманий в естественных языках используется избыточность: определения и дополнения. Формальные языки краткие и максимально выразительные.

Большинство людей, привыкших к естественным языкам, обычно с трудом привыкают к формальным языкам (и программам в том числе). Но стоит помнить, что плотность смысла в таких языках больше, поэтому они медленнее читаются и понимаются. В формальных языках очень важна структура, поэтому чтение справа налево или снизу вверх – не лучший способ их понять. И, наконец, мелочи важны. Маленькие орфографические и пунктуационные ошибки (и опечатки тоже), которые в естественных языках могут быть проигнорированы, в формальных языках будут иметь большое значение, вплоть до изменения смысла на противоположное.

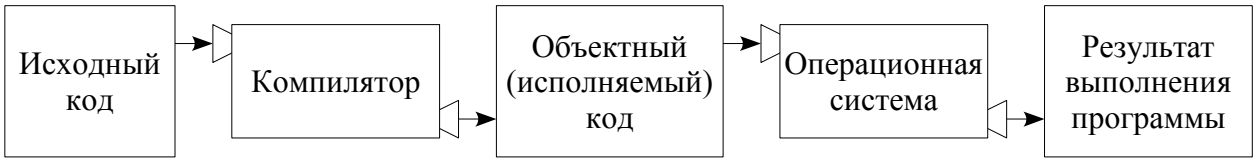
В математике есть целый раздел, посвященный теории формальных языков. Именно на математическом аппарате этой теории основаны синтаксические анализаторы трансляторов.

§1.4. Интерпретаторы и компиляторы

Существует два типа трансляторов, преобразовывающих исходный код программ в машинные команды: *интерпретаторы* и *компиляторы*. Интерпретатор читает высокоуровневую программу (или *исходный код*) и, напрямую взаимодействуя с операционной системой, выполняет ее. Преобразование и выполнение программы выполняется построчно.



В отличие от интерпретаторов, компиляторы полностью преобразовывает исходный код программы в машинный код (или так называемый *объектный код*), который операционная система может выполнить самостоятельно. Это позволяет выполнять скомпилированные программы даже на тех компьютерах, на которых нет компилятора. Кроме того, такие программы выполняются быстрее за счет того, что компьютеру не приходится каждый раз перед запуском программы выполнять ее разбор и преобразование в понятный для себя вид.



Впрочем, современные интерпретаторы тоже способны сохранять *промежуточный код*, на выполнение которого затрачивается меньше времени за счет экономии на синтаксическом разборе исходного кода. Тем не менее, такой промежуточный код понятен только интерпретатору, поэтому для запуска программы его наличие на компьютере все равно необходимо.

Надо сказать, что при современных мощностях компьютеров и объемах памяти разница в скорости выполнения программ интерпретаторами и компиляторами уже почти незаметна, но процесс разработки и отладки программ на интерпретируемых языках намного проще[2].

Язык Питон является интерпретируемым, т.к. написанные на нем программы выполняет интерпретатор.

Для Windows-версии Питона существует любопытный проект: py2exe, позволяющий создавать независимые exe-файлы из скриптов.

По сути py2exe не является собственно компилятором – он преобразовывает скрипт в промежуточный код при помощи самого Питона и помещает в исполняемый файл необходимую часть интерпретатора вместе с этим промежуточным кодом и кодом всех используемых в программе модулей.

Таким образом, вы получаете exe-файл и один или несколько dll-файлов, содержащих вашу программу вместе со средой выполнения. Теперь вам не нужен отдельный интерпретатор Питона для запуска этой программы.

Однако, полученный исполняемый файл может работать только в операционной системе Windows. Если ваша программа предназначена для выполнения только в этой системе, и вы не хотите требовать от пользователей непременной установки Питона, то py2exe может оказаться весьма полезным инструментом.

Получить дополнительную информацию и скачать py2exe можно здесь: http://www.py2exe.org/. Там же можно найти ссылки на другие аналогичные проекты.

§1.5. Первая программа

Настало время запустить интерпретатор Питона и написать первую программу. Существует два способа использования интерпретатора: командный режим и режим выполнения программ из файлов. Если в командной строке интерпретатора Питона вы наберете команду, то интерпретатор тут же выведет результат ее выполнения:

```
$ python
Python 2.3+ (#1, Sep 23 2003, 23:07:16)
[GCC 3.3.1 (SuSE Linux)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> print "Hello world!"
Hello world!
```

Первая строка примера – команда, которая запускает интерпретатор Питона в операционной системе UNIX. В операционной системе Windows для вызова интерпретатора достаточно в меню «Пуск → Программы» вызвать среду разработки IDLE. Следующие две строки – информация от интерпретатора (номер версии, авторские права – так он представляется). Четвертая начинается с приглашения интерпретатора `>>>`, которое обозначает, что он готов выполнять команды. Мы набрали команду `print "Hello world!"`, т.е. дали указание вывести на экран строку `Hello world!`, и в следующей строке интерпретатор вывел то, что мы просили.

2 Компилируемые языки пока незаменимы в системах реального времени, в которых даже малейшая задержка может повлечь за собой катастрофу (например, такие системы используются для управления маневрами космических кораблей и протеканием сложных физических процессов в лабораторных и производственных условиях).

Мы также можем записать программу в файл и использовать интерпретатор для того, чтобы ее выполнить. Такой файл называют *сценарием* или *скриптом* (от англ. script – сценарий). Например, используя текстовый редактор, создадим файл `prog1.py` со следующим содержанием:

```
print "Hello world!"
```

Названия файлов, содержащих программы на Питоне, принято завершать последовательностью символов `.py` (те, кто работают в операционных системах DOS или Windows, назовут это *расширением*). Для того чтобы выполнить программу, мы должны передать интерпретатору в качестве параметра название скрипта:

```
$ python prog1.py
Hello world!
```

В других программных средах метод запуска программ может отличаться, но принцип вызова интерпретатора останется таким же.

Упражнение. *Попробуйте изменить свою первую программу так, чтобы в ней появились синтаксические ошибки: сначала ошибка, связанная с нераспознанной синтаксической единицей (т.е. непонятным словом), а затем – с неправильной структурой программы (можно попробовать поменять местами синтаксические единицы).*
Поэкспериментируйте.

§1.6. Что такое отладка?

Программирование – довольно сложный процесс, и вполне естественно, когда программист допускает ошибку. Так повелось, что программные ошибки называют «*багами*» (от англ. bug – жучок). В русскоязычной литературе это слово вы не встретите, но в сленге программистов оно используется достаточно часто наряду со словом «глюк». Процесс поиска и устранения ошибок в англоязычной литературе принято обозначать термином *debugging*, мы же будем называть его *отладкой*.

Существует три типа ошибок, которые могут возникнуть в программах: *синтаксические ошибки* (*syntax errors*), *ошибки выполнения* (*runtime errors*) и *семантические ошибки* (*semantic errors*). Чтобы находить и исправлять их быстрее, имеет смысл научиться их различать.

§1.6.1. Синтаксические ошибки (syntax errors)

Любой интерпретатор сможет выполнить программу только в том случае, если программа синтаксически правильна. Соответственно компилятор тоже не сможет преобразовать программу в машинные инструкции, если программа содержит синтаксические ошибки. Когда транслятор находит ошибку (т.е. доходит до инструкции, которую не может понять), он прерывает свою работу и выводит сообщение об ошибке.

Для большинства читающих синтаксические ошибки не представляют особой проблемы. Например, часто встречаются стихотворения без знаков препинания, но мы без труда можем их прочесть, хотя это часто порождает неоднозначность их интерпретации. Но трансляторы (и интерпретатор Питона не исключение) очень придирчивы к синтаксическим ошибкам.

Даже если в вашей программе Питон найдет хотя бы незначительную опечатку, он тут же выведет сообщение о том, где он на нее наткнулся, и завершит работу. Такую программу он не сможет выполнить и поэтому отвергнет. В первые недели вашей практики разработки

программ вы, скорее всего, проведете довольно много времени, разыскивая синтаксические ошибки. По мере накопления опыта вы будет допускать их все реже, а находить – все быстрее.

§1.6.2. Ошибки выполнения (runtime errors)

Второй тип ошибок обычно возникает во время выполнения программы (их принято называть *исключительными ситуациями* или, коротко – *исключениями*, по-английски *exceptions*). Такие ошибки имеют другую причину. Если в программе возникает исключение, то это означает, что по ходу выполнения произошло что-то непредвиденное: например, программе было передано некорректное значение, или программа попыталась разделить какое-то значение на ноль, что недопустимо с точки зрения дискретной математики. Если операционная система присылает запрос на немедленное завершение программы, то также возникает исключение. Но в простых программах это достаточно редкое явление, поэтому, возможно, с ними вы столкнетесь не сразу.

§1.6.3. Семантические ошибки (semantic errors)

Третий тип ошибок – семантические ошибки. Первым признаком наличия в вашей программе семантической ошибки является то, что она выполняется успешно, т.е. без исключительных ситуаций, но делает не то, что вы от нее ожидаете.

В таких случаях проблема заключается в том, что семантика написанной программы отличается от того, что вы имели в виду. Поиск таких ошибок – задача нетривиальная, т.к. приходится просматривать результаты работу программы и разбираться, что программа делает на самом деле.

§1.6.4. Процесс отладки

Старый глюк лучше новых двух.
Народная программерская поговорка

Умение отлаживать программы является очень важным навыком для программиста. Процесс отладки требует больших интеллектуальных усилий и концентрации внимания, но это одно из самых интересных занятий.

Отладка очень напоминает работу естествоиспытателя. Изучая результаты своего предыдущего эксперимента, вы делаете некоторые выводы, затем в соответствии с ними изменяете программу, запускаете ее, и снова приступаете к анализу полученных результатов. Если полученный результат не совпадет с ожидаемым, то вам придется снова разбираться в причинах, которые повлекли за собой эти несоответствия. Если же ваша гипотеза окажется правильной, то вы сможете предсказать результат модификаций программы и на шаг приблизиться к завершению работы над ней или, быть может, это заставит вас еще больше уверовать в свое заблуждение.

Поэтому для проверки работоспособности программы не достаточно проверить ее один раз – нужно придумать все возможные наборы входных данных, которые могут как-то повлиять на устойчивость вашей системы. Такие наборы входных данных называют граничными значениями.

Иногда процесс написания и отладки программ разделяют не только во времени, но и между участниками команды разработчиков. Но в последнее время все большую популярность приобретают так называемые гибкие методологии разработки. В них кодирование не отделяется от отладки: программисты, пишущие код, также отвечают и за

подготовку тестов и выявление как можно большего количества ошибок уже в процессе кодирования. Это позволяет им в полной мере насладиться своей работой.

Ядро Linux, исходный код которого содержит миллионы строк, начиналась с простой программы, с помощью которой Линус Торвальдс (Linus Torvalds) изучал возможности параллельного выполнения задач на процессоре Intel 80386. «Одной из ранних программ Линуса была программа, которая переключалась между двумя процессами: печатанием последовательностей AAAA и BBBB. Позже эта программа превратилась в Linux» (Larry Greenfield, The Linux Users' Guide Beta Version 1).

Итак, программирование – это процесс постепенной доработки и отладки до тех пор, пока программа не будет делать то, что мы хотим. Начинать стоит с простой программы, которая делает что-то простое, а затем можно приступать к наращиванию ее функциональности, делая небольшие модификации и отлаживая добавленные куски кода. Таким образом, на каждом шаге у вас будет работающая программа, что, в какой-то мере, позволит вам судить том, какую часть работы вы уже сделали.

В последующих главах у вас будет возможность попрактиковаться в отладке программ. Кое-какие полезные советы по отладке вынесены в *Приложение А* – обязательно почитайте его когда у вас будет свободное время, желательно, ближе к концу изучения шестой главы.

Глава 2. Переменные, операции и выражения

И вот, наконец, мы приступаем собственно к программированию. Можете сразу запустить интерпретатор Питона. Не повредит, если вы проверите работоспособность примеров на своем компьютере и поэкспериментируете с ними. Если найдете какую-нибудь ошибку или опечатку – пишите отчет об ошибке в систему отчетов и предложений на нашем сайте: http://book.it-arts.ru/project/issues/58. Возможно, ваше имя появится в списке разработчиков книги.

§2.1. Значения и типы

Все программы работают со значениями. Значением может быть число или строка. Например, в первой программе мы уже печатали на экране строковое значение `"Hello world!"`. Аналогичным образом мы можем вывести и число:

```
>>> print 12
12
```

`"Hello world!"` и `12` принадлежат к различным типам: `str` (от англ. string – строка) и `int` (от англ. integer – целое число)[3]. Интерпретатор отличает строку от числа по кавычкам, в которые она заключена.

Если есть целые числа, то, по идее, должны быть и дробные. Давайте попробуем такую команду:

```
>>> print 2,4
2 4
```

Вот и первый пример семантической ошибки: мы предполагали, что интерпретатор выведет десятичное число, но он воспринял запятую как разделитель между двумя целыми числами и вывел их, разделив пробелом. Вообщем, он нас неправильно понял, ведь разделителем дробной и целой частей числа в Питоне, как и в большинстве других языков программирования, служит точка[4]:

```
>>> print 2.4
2.4
```

Если вы не уверены в том, к какому типу принадлежит значение, это можно проверить так:

```
>>> type("Hello world!")
<type 'str'>
>>> type(12)
<type 'int'>
>>> type(2.4)
<type 'float'>
```

Строковый тип называется в Питоне `str`, целочисленный носит название `int`, а дробный – `float` (от англ. floating-point number – число с плавающей точкой).

3 В некоторых версиях Питона эти типы носят несокращенные названия: `string` и `integer`.

4 Такой непривычный для нас способ разделения целой и дробной части является американским стандартом записи десятичных дробей.

Упражнение. *Проведите самостоятельно следующий эксперимент: проверьте типы значений* `"12"` *и* `"2.4"`*? Какого они типа и почему?*

Упражнение. *Что произойдет, если строку* `"8.53"` *попытаться преобразовать в целое число с помощью функции* `int()`*? Как решить эту проблему?*

§2.2. Преобразование типов

В предыдущем разделе мы научились выводить на экран целые числа – это довольно просто делается, не так ли? Но для интерпретатора эта задача выглядит несколько сложнее, т.к. ему перед выводом числовых значений приходится преобразовывать их в строки.

Программист тоже может это делать – Питон имеет целую коллекцию встроенных функций, которые умеют преобразовывать значения одного типа в другой. Например, функция `int()` преобразовывает значение в целочисленный тип. Если преобразование произвести невозможно, то возникает исключение:

```
>>> int("32")
32
>>> int("Hello")
Traceback (most recent call last):
  File "", line 1, in ?
ValueError: invalid literal for int(): Hello
```

Функция `int()` может приводить к целому типу и дробные числа, но не забывайте, что при преобразовании она просто отбрасывает дробную часть:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

Функция `float()` преобразовывает целые числа и строки в дробный тип:

```
>>> float(32)
32.0
>>> float("3.14159")
3.14159
```

И, наконец, функция `str()` отвечает за преобразование к строковому типу. Именно ее предварительно запускает команда `print`:

```
>>> str(32)
'32'
>>> str(3.14149)
'3.14149'
```

Может показаться странным, что Питон различает целое число `1` от дробного `1.0`: это одно и то же число, но оно принадлежит к различным типам. От типа значения зависит способ его хранения в памяти.

§2.3. Переменные

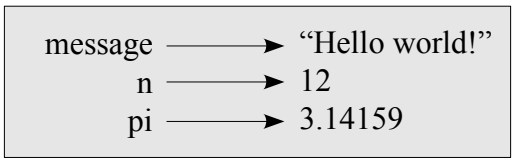
Как любой другой язык программирования Питон поддерживает концепцию переменных, но с небольшим отличием. Если в языках C++ или Pascal переменная – это имя ячейки памяти, в которой хранится значение, то в Питоне переменная – это ссылка на ячейку памяти. Различие, на первый взгляд, несущественное, но на самом деле это немного другой подход к организации хранения объектов в памяти. Впрочем, нас пока это не особо волнует.

Для того, чтобы «запомнить» значение достаточно присвоить его переменной. Это делается с помощью специального оператора присваивания который обозначается знаком равенства (`=`).

```
>>> message = "Hello world!"
>>> n = 12
>>> pi = 3.14159
```

В данном примере переменной `message` присваивается (или сопоставляется) значение `"Hello world!"`, переменной `n` присваивается значение `12`, а переменной `pi` – `3.14159`.

Самый простой способ графического представления переменных: написать имя переменной, нарисовать стрелку, на другом конце которой дописать присвоенное значение. Такие рисунки иногда называют *диаграммами состояния* (*state diagram*), т.к. они отображают состояние, в котором находится переменная, т.е. какое значение ей в данный момент присвоено.



Команда `print` работает и с переменными:

```
>>> print message
Hello world!
>>> print n
12
>>> print pi
3.14159
```

Как видите, команда `print` выводит не имена переменных, а их значения. Переменные, так же как и значения, имеют тип. Давайте это проверим с помощью функции `type()`:

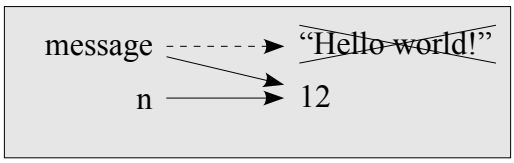
```
>>> type(message)
<type 'str'>
>>> type(n)
<type 'int'>
```

```
>>> type(pi)
<type 'float'>
```

Тип переменной совпадает с типом присвоенного ей значения. Рассмотрим еще один пример:

```
>>> message = "Hello world!"
>>> n = 12
>>> type(message)
<type 'str'>
>>> message = n
>>> print message
12
>>> type(message)
<type 'int'>
```

Этот пример интересен по двум причинам. Во-первых, в нем использована возможность присваивать значение одной переменной другой. Конструкция `message = n` работает аналогично присваиванию переменной значения: переменной `message` присваивается значение переменной `n`. При этом значение `12` хранится в памяти только один раз – Питон довольно экономно расходует память.



Во-вторых, как видно из примера, переменная `message` после присваивания ей значения `n` поменяла свой тип. Далеко не каждый язык программирования «умеет» это делать так просто.

§2.4. Имена переменных и ключевые слова

Как заметил Фредерик Брукс (Frederick P. Brooks, Jr)[5], самое захватывающее в профессии программиста то, что он работает с идеей в чистом виде: он записывает абстрактные идеи с помощью формальных языков, чтобы те облегчали труд другим людям. Для преобразования абстрактной, непроявленной в материальном мире идеи в код программы необходимо выделить сущности и действия, придумать им названия, чтобы ими управлять, проследить связи между ними и их свойствами. Вы, наверное, догадались, что именами сущностей, с которыми работает программист, служат переменные. Поэтому стоит выбирать осмысленные названия переменных.

Имена переменных могут быть произвольной длины, но старайтесь выбирать не слишком короткие и не слишком длинные имена – от этого зависит читабельность программы.

5 Фредерик Брукс (Frederick P. Brooks, Jr) руководил знаменитым проектом IBM OS/360; его перу принадлежит принцип No Silver Bullet (NSB – «Серебрянной пули нет», в смысле универсального средства борьбы с проектами- «монстрами»). Речь идет о его книге «Мифический человеко-месяц, или как создаются программные системы».

При составлении имен переменных в Питоне можно использовать любые латинские буквы, цифры и знак _ (знак подчеркивания). Знак подчеркивания может использоваться для разделения слов составляющих имя переменной: например, `user_name` или `full_price`. Но названия переменных не могут начинаться с цифры.

```
>>> 1message = "Hello world!"
  File "<stdin>", line 1
    1message = "Hello world!"
            ^
SyntaxError: invalid syntax
>>> price_in_$ = 300
  File "<stdin>", line 1
    price_in_$ = 300
              ^
SyntaxError: invalid syntax
>>> class = "Computer Science 101"
  File "<stdin>", line 1
    class = "Computer Science 101"
          ^
SyntaxError: invalid syntax
```

Разберем эти три примера. Первое же выражение интерпретатору не понравилось, и он отметил знаком `^`, где именно у него возникли претензии: он указал на имя переменной `1message`. Действительно, имя `1message` является некорректным, ведь оно начинается с цифры. Аналогичная ситуация с именем `price_in_$`: оно содержит недопустимый символ `$`. Но что интерпретатору не нравится в третьем выражении? Давайте попробуем изменить имя переменной `class` на что-нибудь похожее, например, `class_`:

```
>>> class_ = "Computer Science 101"
>>> print class_
Computer Science 101
```

Теперь все в порядке. В чем же дело? Почему имя `class` вызвало ошибку, а имя `class_` – нет? Какие есть предположения? Поставим еще один эксперимент:

```
>>> print = "Some message"
  File "<stdin>", line 1
    print = "Some message"
          ^
SyntaxError: invalid syntax
```

Знакомая ситуация, не так ли? Проанализируем то, что мы получили. В качестве имени переменной мы пытались использовать команду `print` и получили аналогичную ошибку, значит слово `class`, скорее всего, тоже является командой или каким-то служебным словом.

Действительно, слова `class` и `print` являются так называемыми *ключевыми словами*. Всего в Питоне версии 2.3. зарезервировано 29 ключевых слов:

`and`	`del`	`for`	`is`	`raise`
`assert`	`elif`	`from`	`lambda`	`return`
`break`	`else`	`global`	`not`	`try`
`class`	`except`	`if`	`or`	`while`
`continue`	`exec`	`import`	`pass`	`yield`
`def`	`finally`	`in`	`print`	

Полезно иметь этот список под рукой, чтобы заглянуть в него, когда вдруг интерпретатору не понравится одно из имен переменных.

Имейте также ввиду, что интерпретатор различает большие и маленькие буквы, т.е. `message` и `Message` будут разными переменными.

Упражнение. *Напишите программу, которая подтверждает, что интерпретатор Питона различает строчные и заглавные буквы в именах переменных.*

§2.5. Выражения

В первой главе мы уже сталкивались с понятием выражения в общем виде и использовали их в программе. Давайте дадим определение этого термина. *Выражение* – это последовательность синтаксических единиц, описывающая элементарное действие на языке программирования. Например, `print "Hello world!"` и `message = n` являются выражениями.

Когда вы набираете выражение в командной строке, интерпретатор выполняет его и выводит результат, если таковой имеется. Результатом выражения `print "Hello world!"` является строка: `Hello world!`. Выражение присваивания ничего не выводит.

Скрипты обычно содержат последовательность выражений, результаты выражений выводятся по мере их выполнения. Например, запустим скрипт, содержащий следующие выражения:

```
print 1
x = 2
print x
```

Этот скрипт выведет следующее:

```
1
2
```

И снова выражение присваивания не порождает никакого вывода.

§2.6. Выполнение выражений

По сути, выражение – это последовательность значений, переменных и операторов. Если вы напишите выражение, то интерпретатор, выполнив его, выведет на экран:

```
>>> 1 + 1
2
```

Значение само по себе рассматривается как выражение, так же как и переменная:

```
>>> 17
17
>>> x = 2
>>> x
2
```

Но выполнение и вывод результата выполнения выражения не совсем то же самое:

```
>>> message = "Hello world!"
>>> message
"Hello world!"
>>> print message
Hello world!
```

Когда Питон выводит значение выражения в командном режиме, он использует тот же формат, что используется при вводе этого значения. Например, при выводе строк он заключает их в кавычки. Команда `print` также выводит значение выражения, но в случае со строками, она выводит содержимое строки без кавычек.

В скриптах, выражения вообще ничего не выводят, если в них нет инструкции `print`. Например, следующий скрипт не выводит ни одной строки:

```
17
3.2
"Hello, World!"
1 + 1
```

Упражнение. *Измените скрипт из предыдущего примера так, чтобы он вывел значения всех четырех выражений.*

В командном режиме интерпретатор Питона результат последнего выражения сохраняет в специальной переменной `_` (знак подчеркивания). Вы можете просмотреть результат выполнения последнего выражения и использовать его в своих выражениях:

```
>>> 1.25 + 1
2.25
>>> print _
2.25
>>> 5 + _
7.25
```

Поэтому Питон довольно удобно использовать как простой калькулятор. В последующих разделах мы разберемся с тем, какие операции способен выполнять интерпретатор Питона.

§2.7. Операторы и операнды

Операторами называют специальные символы (или последовательности символов), обозначающие некоторые операции. Например, знаком `+` обозначают операцию сложения, а знаком `*` – умножение. Значения, над которыми выполняется операция, называют операндами.

Все нижеследующие выражения, с точки зрения Питона, корректны:

```
20+32   hour-1   hour*60+minute   minute/60   5**2   (5+9)*(15-7)
```

Значения большинства из них понять нетрудно. Значения символов `+`, `-`, `*` и `/` в Питоне такие же, как в математике. Скобки используются для группировки операций, а двумя звездочками (`**`) обозначается операция возведения в степень.

Если операндом является переменная, то перед вычислением выражения производится подстановка на ее место значения, на которое указывает эта переменная.

Сложение, вычитание, умножение и возведение в степень работают привычным для нас способом, но действие операции деления несколько отличается. Это иллюстрирует следующий пример:

```
>>> minute = 59
>>> minute/60
0
```

Значение переменной `minute` равно `59`; результат деления `59` на `60` должен быть `0.98333`, а не `0`. Причиной этого несоответствия является то, что Питон выполняет целочисленное деление.

Когда оба операнда – целые числа, и Питон считает, что результат тоже должен быть целым. Поэтому целочисленное деление всегда отбрасывает дробную часть.

Как получить дробный результат? Достаточно принудительно преобразовать один из операндов в дробное число:

```
>>> minute = 59
>>> float(minute) / 60
0.983333333333
```

Другой вариант:

```
>>> minute = 59
>>> minute / 60.0
0.983333333333
```

Если один из операндов принадлежит типу `float`, то второй автоматически преобразовывается к этому типу, как к более сложному.

§2.8. Порядок операций

Большинство языков программирования позволяют группировать в одном выражении несколько операций. Это удобно, например, если нужно посчитать процентное соотношение двух величин:

```
>>> print 100 * 20 / 80, "%"
25 %
```

В этом примере рассчитывается процентное соотношение двух чисел: `20` и `80`. После результата выражения выводится символ `%` – интерпретатор вычисляет арифметическое выражение и выводит результат, а затем дописывает строку, стоящую после запятой.

Когда в выражении более одного оператора, последовательность выполнения операций зависит от порядка их следования в выражении, а также от их приоритета. Приоритеты операторов в Питоне полностью совпадают с приоритетами математических операций.

Самый высокий приоритет у скобок, которые позволяют изменять последовательность выполнения операций. Таким образом, операции в скобках выполняются в первую очередь. Например, `2*(3-1)` равно `4`, `(1+1)**(5-2)` – `8`. Скобки удобно использовать и для того, чтобы выражения было легче читать, даже если их наличие в выражении никак не отражается на результате: `(100*20)/80`.

Следующий приоритет у операции возведения в степень, поэтому `2**1+1` равно `3`, а не `4`, и выражение `3*1**3` даст результат `3`, а не `27`.

Умножение и деление имеют одинаковый приоритет, более высокий, чем у операций сложения и вычитания. `2*3-1` равно `5`, а не `4`; `2/3-1` равно `-1`, а не `1` (результат целочисленного деления `2/3=0`).

Операторы с одинаковым приоритетом выполняются слева направо. Так что в выражении `100*20/80` умножение выполняется первым (выражение приобретает вид `2000/80`); затем выполняется деление, выдающее в результате значение `25`. Если бы операции выполнялись справа налево, то результат получился бы другим.

Упражнение. *Измените выражение `100*20/80` так, чтобы последовательность выполнения операций была обратной. Какой результат вы получили после его выполнения и почему?*

§2.9. Простейшие операции над строками

Вообще говоря, над строками нельзя производить те же операции, что и над числами, поэтому следующие примеры работать не будут:

```
message-1   "Hello"/123   "Hello"*"world!"   "15"+2
```

Но оператор `+` работает со строками, хотя обозначает другую операцию: *конкатенацию* или *сцепление строк*.

```
>>> str1 = "Hello"
>>> str2 = " world"
>>> print str1 + str2 + "!"
Hello world!
```

Обратите внимание, что вторая строка начинается с пробела. Пробел такой же символ, как и любой другой.

Оператор `*` тоже можно использовать по отношению к строкам, но при условии, что одним из операндов будет целое число. В этом случае оператор `*` символизирует *операцию повторения строки* (или *итерацию*). Например, `'Fun'*3` выдаст результат `'FunFunFun'`.

Можно проследить аналогию между операциями над числами и операциями над строками: так же, как `4*3` эквивалентно `4+4+4`, `'Fun'*3` эквивалентно `'Fun'+'Fun'+'Fun'`. Но с другой стороны, конкатенация и повторение имеют несколько существенных отличий от сложения и умножения.

Упражнение. *Все операции в математике классифицируются по их свойствам (коммутативность, ассоциативность и т.п.). Какими свойствами, присущими сложению и умножению, не обладают конкатенация и повторение?*
Не ленитесь заглянуть в математический справочник – понимание свойств объектов, которыми оперируете, даст вам большие преимущества.

Упражнение. *Какой результат будет получен после выполнения выражения* `"hello"+" world"*3`*?Какие выводы можно сделать на основе этого результата?*

§2.10. Композиция

Напоследок разберемся еще с одной важной концепцией: композицией. Сами того не замечая, мы с нею уже сталкивались.

Весь окружающий нас мир состоит из составных частей. Например, дома состоят из кирпичей или строительных блоков, кирпичи, в свою очередь состоят из мелких крупиц песка, а те – из молекул и атомов. Для того, чтобы моделировать явления окружающего мира, инструмент построения моделей должен также иметь возможность собирать мелкие объекты в системы, которые в свою очередь будут объединяться в еще более сложные комплексы.

Средством моделирования (т.е. описания различных объектов и ситуаций) для программиста являются языки программирования. Во всех высокоуровневых языках реализована возможность составления систем из более простых элементов, или *композиция*.

Например, мы знаем, как сложить два числа и как выводить полученное значение. Таким образом, мы можем это сделать в одном выражении:

```
>>> print 17 + 3
20
```

На самом деле сложение и вывод значения происходят не в то же самое время: сначала вычисляется выражение, после чего выводится результат. Но вы можете выводить таким образом результат любого выражения, каким бы сложным оно не было.

Выражения могут строиться из других выражений несколькими способами. Рассмотрим такой пример:

```
>>> percentage = 100 * 20 / 80; print percentage
25
```

Во-первых, выражения могут быть вложенными друг в друга. В частности, выражение присваивания имеет такую структуру, что слева от знака присваивания должно стоять имя переменной, которой присваивается результат вычисления выражения, стоящего справа от

оператора присваивания. В свою очередь, выражение в правой части тоже сложное: вычисление проводится в порядке, определяемом приоритетами операций. Операция деления в нашем примере в качестве первого операнда принимает результат умножения – это тоже пример вложенности выражений.

Второй вариант компоновки выражений – следование. В примере два выражения (выражение присваивания и вывода значения переменной `percentage`) разделены точкой с запятой. Эти выражения выполняются друг за другом. Вообще говоря, любая программа представляет собой последовательность выражений. Если выражения расположены на отдельных строках, то ставить точку с запятой в конце каждого выражения не обязательно: в данном случае разделителем выражений служит символ завершения строки. Это правило относится к Питону, но в C++ и Паскале выражения обязательно должны заканчиваться точкой с запятой.

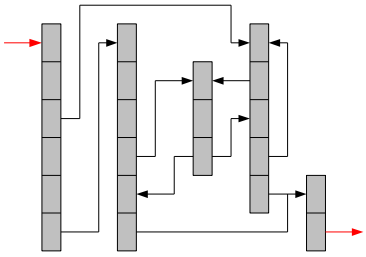
В последствии вы увидите, что композиция может применяться для комбинирования любых синтаксических единиц и выражений, в том числе и вызовов подпрограмм.

Упражнение. *Попробуйте выполнить в командной строке интерпретатора Питона команду:* `100 * 20 / 80 = percentage`*. Какой результат вы получили и почему?*

Глава 3. Функции

§3.1. Подпрограммы

Еще во времена адресного программирования было замечено, что некоторые части программы можно использовать более одного раза, если изменить значение какого-то параметра. Например, в программе может несколько раз вычисляться значение по определенной формуле, но каждый раз значения, подставляемые в формулу, меняются. Программисты стали обобщать решения небольших повторяющихся подзадач и обращаться к частям программы с этими решениями там, где это было нужно. Программы перестали быть линейными, поэтому стали напоминать пачку макарон, т.к. состояли из цепочек инструкций, а переключение между ними осуществлялось с помощью команд условных или безусловных переходов и меток, определяющих начало каждой «макаронины».



В какой-то мере, это заметно усложняло процесс отладки и расширения функциональности программ, но, с другой стороны, открывало перед программистами новые возможности: во-первых, программы стали занимать меньше места, и в них стало легче ориентироваться; во-вторых, исправлять ошибки стало намного проще, т.к. программистам теперь не нужно было искать все повторяющиеся с незначительным изменениями части кода, чтобы исправить ошибку найденную в одной из них – достаточно исправить этот недочет всего в одном месте.

Идея *повторного использования кода* очень важна в программировании, т.к. без нее поддержка и развитие современных программных продуктов с учетом их сложности представляет собой почти неразрешимую задачу. По крайней мере, стоимость владения программами была бы без нее намного выше, и их использование стало бы в большинстве случаев нерентабельным.

В структурных языках программирования концепция повторного использования кода реализована несколькими различными способами. Один из них – написание *подпрограмм*. Подпрограммы представляют такой же набор инструкций, как и сама программа, но они решают менее сложные задачи. Подпрограммы могут получать на входе некоторый набор параметров и возвращать какое-то значение на выходе.

В языке Pascal подпрограммы бывают двух типов: *процедуры* и *функции*. В Питоне же, так же как и в C++, подпрограммы реализованы в более общем виде, поэтому такого разделения нет – в этих языках есть только функции.

Иногда подпрограммы группируются в так называемые *модули* – специальные файлы, в которых хранится описание этих функций, которые можно подключать и использовать в любой части проекта или даже в других проектах. О том как создавать и подключать модули в Питоне подробно рассказано в *Приложении B*. Но пока приступим к освоению функций.

§3.2. Вызовы функций

С вызовами функций мы уже сталкивались, когда выполняли определение и преобразование типов значений и переменных:

```
>>> type("Hello world!")
<type 'str'>
>>> int("32")
32
```

Функция, определяющая тип значения или переменной, называется `type()`; преобразуемое значение или переменная должны следовать после названия функции в скобках – его называют аргументом функции. Как видно из примера, вызов функции `int()` осуществляется аналогичным образом.

Таким образом, для вызова функции достаточно набрать ее имя и в скобках перечислить параметры, а если передавать параметры не требуется, то в скобках можно ничего не писать. Например, в функции `str()` первый параметр является необязательным, т.е. его можно не указывать:

```
>>> str()
''
```

§3.3. Справочная система

В Питоне определено множество функций, позволяющих решать различные задачи. Некоторые из них сгруппированы по назначению и вынесены в отдельные модули. В следующем разделе мы научимся импортировать модули и использовать функции из них, но, для начала, разберемся с тем, как определять, для чего предназначена та или иная функция. Этот навык вам очень пригодится в дальнейшем, т.к. часто довольно трудно догадаться о назначении функций по их названию.

Для упрощения работы программиста в Питоне предусмотрена встроенная переменная `__doc__` (начинается и заканчивается парами символов подчеркивания), в которой обычно хранится минимальная справочная информация:

```
>>> print str.__doc__
str(object) -> string
Return a nice string representation of the object.
If the argument is a string, the return value is the same object.
>>>
```

Функция `str()` «представилась» и вывела информацию о себе: какие параметры она принимает, какого типа значение возвращает, и коротко «рассказала», что она делает.

Начиная с версии 2.2, в Питоне появилась справочная система, реализованная в виде функции `help()`. Данная функция в качестве параметра принимает имя любого объекта (в том числе, модуля или функции) и выводит справочную информацию о нем.

Упражнение. *Ознакомьтесь со справочной информацией функции `str()`. Имейте ввиду, что в Питоне версий ниже 2.2 функция `help()` работать не будет. Выход из справочной системы осуществляется клавишей `[Q]`.*

---

§3.4. Импорт модулей и математические функции

Помимо простейших операций над числами, которые мы уже рассматривали, Питон способен вычислять значения более сложных математических функций: тригонометрических, логарифмических и др. Для того, чтобы получить доступ к этим функциям необходимо импортировать или подключить специальный модуль, в котором хранятся определения этих функций. Сделать это можно с помощью такой инструкции:

```
>>> import math
>>>
```

Данная команда импортирует модуль `math`. Питон поддерживает частичный импорт модулей (подробнее об этом можно прочесть в Приложении В), но нам пока это не нужно. Итак, мы импортировали модуль с описанием математических функций и теперь попробуем вызвать из него одну из них. Для начала определим, какие функции и константы определены в модуле:

```
>>> dir(math)
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',
'atan2', 'ceil',
'cos', 'cosh', 'e', 'exp', 'fabs', 'floor', 'fmod', 'frexp',
'hypot', 'ldexp',
'log', 'log10', 'modf', 'pi', 'pow', 'sin', 'sinh', 'sqrt', 'tan',
'tanh']
>>>
```

В результате выполнения этой команды интерпретатор вывел все имена, определенные в данном модуле. В их числе есть и переменная `__doc__`. Для того, чтобы обратиться к переменной или функции из импортированного модуля необходимо указать его имя, поставить точку и написать необходимое имя:

```
>>> print math.__doc__
This module is always available. It provides access to the
mathematical functions defined by the C standard.
>>> print math.pi
3.14159265359
>>> decibel = math.log10(17.0)
```

Первая строка примера выводит описание модуля; четвертая – значение константы π, а шестая – вычисляет значение логарифма 17 по основанию 10. Помимо десятичного логарифма в модуле определен и натуральный логарифм (т.е. с основанием e = 2.71828182845904530): `math.log()`. Еще один пример:

```
>>> height = math.sin(45)
>>> print height
0.850903524534
```

Если вы изучали тригонометрию, то, наверное, помните, что синус 45 градусов равен квадратному корню из 2 деленному на 2. Давайте проверим результат вычисления функции:

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

Результаты получились разные. Почему? Разработчики допустили ошибку? Вполне вероятно – никто не застрахован от ошибок. Но, прежде чем писать отчет об ошибке и отправлять его разработчикам, давайте ознакомимся со справкой по данной функции:

```
>>> help(math.sin)
Help on built-in function sin:

sin(...)
    sin(x)

    Return the sine of x (measured in radians).
(END)
```

Из справки видно, что в качестве параметра тригонометрические функции принимают значение угла в радианах. Нажимаем `[Q]` для выхода из справочной системы с чувством гордости: теперь мы знаем как работает функция `math.sin()`; проведем преобразование в радианы:

```
>>> degrees = 45
>>> angle = degrees * math.pi / 180.0
>>> height = math.sin(angle)
>>> print height
0.707106781187
```

Теперь результаты совпали – все верно.

Упражнение. *Выполните в командной строке интерпретатора Питона версии 2.2 или выше следующие инструкции:*

```
>>> import math
>>> print math.__doc__
...
>>> help(math)
```

Ознакомьтесь с описанием модуля и входящих в него функций. Вся справка на экране не поместится, но вы сможете прокручивать текст с помощью клавиш-стрелок. Выход из справочной системы осуществляется клавишей `[Q]`.
Если возникнет желание, можете «поиграться» с этими функциями, написав несколько простеньких программок с их использованием. Где их можно применять?

§3.5. Композиция

Во второй главе мы уже рассматривали композицию применительно к арифметическим операциям и простым выражениям. Но композиция позволяет комбинировать и вызовы функций: функция в качестве параметра может принимать любое выражение, возвращающее значение, которое попадает в область ее определения; с другой стороны, вызов функции (точнее, результат ее выполнения) может быть использован в качестве операнда:

```
>>> x = math.sin(math.asin(math.sqrt(2) / 2.0) + math.pi / 2)
>>> print x
0.70710678118654757
>>>
```

Разберем этот пример. Первая операция, с которой сталкивается синтаксический анализатор при разборе этого выражения, это присваивание значения выражения, стоящего справа от знака `=`, переменной `x`. Перед присваиванием интерпретатор должен вычислить значение этого выражения, т.е. посчитать синус значения выражения `math.asin(math.sqrt(2) / 2.0) + math.pi / 2`, которое, в свою очередь равно сумме значений подвыражений `math.asin(math.sqrt(2) / 2.0)` и `math.pi / 2`. Первое из них вычисляется аналогичным образом: сначала рассчитывается значение выражения, стоящего в скобках, а затем вызывается функция `math.asin()`, которой полученное выражение передается в качестве аргумента. Второе подвыражение равно π/2. Подсчитав сумму этих двух подвыражений, интерпретатор передает полученное значение в качестве аргумента функции `math.sin()`, а результат присваивает переменной `x`.

Напишите программу, вычисляющую значения корней квадратного уравнения $ax^2+bx+c=0$, `a<>0` *по коэффициентам этого уравнения. Коэффициенты должны заноситься в начале программы в переменные* `a`, `b` *и* `c`.
Попробуйте переменным `a`, `b` *и* `c` *задать значения* `1`, `0` *и* `-1` *соответственно. Программа должна вывести значения* `1.0` *и* `-1.0`. *Проверьте совпали ли ваши результаты с ожидаемыми.*
Экспериментируя с другими коэффициентами, имейте ввиду, что существуют уравнения не имеющие решений (у таких уравнений дискриминант меньше нуля), поэтому не пугайтесь, если в вашей программе возникнет ошибка вида: `ValueError: math domain error`. *Немного позже мы узнаем, как этого избежать.*

---

§3.6. Создание функций

До сих пор мы использовали встроенные функции Питона и функции из модулей, которые входят в комплект его поставки. Но мощь структурных языков программирования заключается в том, что мы можем создавать свои собственные функции, причем делается это довольно просто. В структурном программировании *функция* представляет собой именованную последовательность выражений, выполняющих необходимую операцию. В Питоне существует специальный *оператор определения функции*, имеющий следующий синтаксис:

```
def ИМЯ_ФУНКЦИИ(СПИСОК_ПАРАМЕТРОВ):
    ПОСЛЕДОВАТЕЛЬНОСТЬ_ВЫРАЖЕНИЙ
```

Правила выбора имен функций полностью аналогичны правилам выбора имен переменных, описанным в предыдущей главе. Список параметров определяет набор значений, которые могут быть переданы функции в качестве исходных данных – параметры

перечисляются через запятую. Первая строка определения обычно называется *заголовком функции*, обозначенным ключевым словом `def` (от англ. «define» – «определить»). Заголовок функции в Питоне завершается двоеточием. После него может следовать любое количество выражений, но они должны быть записаны со смещением относительно начала строки. Для такого смещения можно использовать один или несколько пробелов, но, пожалуй, удобнее всего вместо пробелов использовать символ табуляции (клавиша `[Tab]`). Напишем и протестируем нашу первую функцию; для завершения последовательности выражений, составляющих *тело функции* достаточно еще раз нажать клавишу `[Enter]` (в редакторе, поддерживающем автоотступы, еще придется нажать `[Backspace]`, чтобы убрать автоматически добавленный отступ):

```
>>> def printAnything(object):
...     print object
...
>>> printAnything("Some string")
Some string
>>> number = 15
>>> printAnything(number)
15
```

В первых двух строках примера мы определили функцию `PrintAnything()`, которая печатает объект, переданный в качестве параметра. Мы можем передать этой функции строку или число, что мы и попробовали сделать в четвертой и седьмой строках примера. Полученные результаты совпали с ожидаемыми, но давайте не будем на этом останавливаться и поставим еще один эксперимент.

Что если передать в качестве параметра имя функции? Вызовем `PrintAnything()`, указав в скобках ее же имя:

```
>>> printAnything(printAnything)
<function printAnything at 0x80d7554>
>>>
```

Питон вывел информацию о функции: ее имя и адрес в памяти, где она хранится! В данном случае адрес равен `0x80d7554`, но у вас на компьютере этот адрес может быть другим. Теперь давайте откроем текстовый редактор и напишем такую программу:

```
def printHello():
    print "Hello!"


def runFunction(function_name):
    function_name()


runFunction(printHello)
```

Сохраним программу в файле с именем `prog2.py` и запустим ее из командной строки (в операционной системе Windows достаточно двойного клика левой кнопки мыши на пиктограмме этого файла[6]):

```
$ python prog2.py
Hello!
```

Наверное, вы уже поняли, что сделал Питон, когда мы вызвали функцию `runFunction()` и передали ей в качестве параметра имя функции `printHello()`: он заменил `function_name` на значение переданного параметра и вызвал таким образом функцию `printHello()`. Такой гибкостью может похвастаться редкий язык программирования. В языках C++ или Pascal это делается далеко не тривиальным способом.

Имейте также ввиду, что вызывать функцию до ее определения нельзя – это приведет к возникновению исключения.

Упражнение. *Переделайте программу вычисления корней квадратного уравнения в функцию `printRoots()`. Параметрами данной функции должны быть коэффициенты квадратного многочлена: a, b и c. Рассчитав корни уравнения, функция должна вывести их на печать.*

Упражнение. *Напишите функцию, рассчитывающую расстояние между двумя точками на плоскости (координаты (x1, y1) и (x2, y2) даны).*

$$distance(x_1, y_1, x_2, y_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

§3.7. Параметры и аргументы

Итак, мы научились создавать собственные функции и различать определения и вызовы функций. Теперь определимся с терминологией: чем отличается параметр функции от ее аргумента? Разница есть и большая, но вы могли ее не уловить.

Параметрами функции называют список переменных, которым присваиваются передаваемые при вызове этой функции значения. А сами передаваемые значения называют *аргументами*. В своем теле функция оперирует параметрами, но не аргументами. Разберем пример из предыдущего раздела:

```
>>> def printAnything(object):
...     print object
...
>>> printAnything("Some string")
Some string
>>> number = 15
>>> printAnything(number)
15
```

Здесь переменная `object` является параметром, а строка `"Some string"` и переменная `number` – аргументами. Таким образом, правильно будет сказать «передать аргумент» или

6 Не забудьте в последней строке программы написать команду ожидания нажатия клавиши, как это описано в разделе «Запуск программ, написанных на Питоне» (см. Введение).

«передать значение в качестве параметра», но не «передать функции параметр». Старайтесь не путать эти понятия.

§3.8. Локальные переменные

Вернемся к функции поиска корней квадратного уравнения `PrintRoots()`. Она выглядит приблизительно так:

```
def printRoots(a, b, c):
    D = b**2 - 4 * a * c
    import math
    x1 = (-b + math.sqrt(D)) / 2 * a
    x2 = (-b - math.sqrt(D)) / 2 * a
    print 'x1 =', x1, '\nx2 =', x2
```

Последовательность символов `\n` в последней строке тела функции дает указание команде `print` перейти на новую строку. В теле функции определено три переменные: `D`, `x1` и `x2`. Такие переменные называют *локальными*, т.к. к ним можно обратиться только в самом теле функции. После завершения работы функции они стираются из памяти. Это может проиллюстрировать следующая программа:

```
def printRoots(a, b, c):
    D = b**2 - 4 * a * c
    import math
    x1 = (-b + math.sqrt(D)) / 2 * a
    x2 = (-b - math.sqrt(D)) / 2 * a
    print 'x1 =', x1, '\nx2 =', x2


printRoots(1.0, 0, -1.0)
print D
```

Если запустить ее, то интерпретатор сразу после завершения работы функции `PrintRoots()` выведет ошибку вида:

```
$ python prog5.py
x1 = 1.0
x2 = -1.0
Traceback (most recent call last):
  File "prog5.py", line 12, in ?
    print D
NameError: name 'D' is not defined
```

Почему было выбрано такое поведение? Как вы думаете? Модифицируем нашу программу таким образом:

```
def printRoots(a, b, c):
    D = b**2 - 4 * a * c
    import math
    print "In function D = ", D
    x1 = (-b + math.sqrt(D)) / 2 * a
    x2 = (-b - math.sqrt(D)) / 2 * a
    print "x1 =", x1, "\nx2 =", x2

D = 'test'
print "Before function call D = ", D
printRoots(1.0, 0, -1.0)
print "After function call D = ", D
```

Запустим программу и проанализируем результат.

```
$ python prog5.py
Before function call D = test
In function D = 4.0
x1 = 1.0
x2 = -1.0
After function call D = test
```

Итак, после определения функции `printRoots()` мы присвоили переменной `D` значение `"test"` и проверили результат: вывелась строка `"test"`. Затем вызвали функцию `printRoots()`, в теле которой после присвоения переменной `D` результата вычисления формулы дискриминанта так же выводится значение переменной `D`: `4.0`. После завершения работы функции мы снова выводим значение переменной `D`: на этот раз присвоенное значение снова равно строке `"test"`.

На самом деле, `D` в основной программе и `D` в теле функции `printRoots()` – это две разные переменные. Проверить это можно, немного модифицировав нашу программу. Воспользуемся функцией `id()`. Она возвращает адрес объекта в памяти. Измененная программа будет выглядеть приблизительно так:

```
def printRoots(a, b, c):
    D = b**2 - 4 * a * c
    import math
    print "In function D = ", D, "\nAdress:", id(D), "\n"
    x1 = (-b + math.sqrt(D)) / 2 * a
    x2 = (-b - math.sqrt(D)) / 2 * a
    print "x1 =", x1, "\nx2 =", x2

D = "test"
```

```
print "Before function call D = ", D, "\nAdress:", id(D), "\n"
printRoots(1.0, 0, -1.0)
print "After function call D = ", D, "\nAdress:", id(D), "\n"
```

Результат работы программы будет следующим:

```
$ python prog5.py
Before function call D = test
Adress: 135287104

In function D = 4.0
Adress: 135293780

x1 = 1.0
x2 = -1.0
After function call D = test
Adress: 135287104
```

На вашем компьютере адреса могут быть другими, но выводы вы сможете сделать такие же: адреса `D` в основной программе и `D` в теле функции `printRoots()` различаются. Что и требовалось доказать.

Упражнение. *Являются ли параметры функции локальными переменными?*

§3.9. Поток выполнения

Как уже было сказано, с появлением функций (и процедур) программы перестали быть линейными, в связи с этим возникло понятие *потока выполнения* – последовательности выполнения инструкций, составляющих программу.

Выполнение программы, написанной на Питоне всегда начинается с первого выражения, а последующие выражения выполняются один за другим сверху вниз. Причем, определения функций никак не влияют на поток выполнения, т.к. тело любой функции не выполняется до тех пор, пока не будет вызвана соответствующая функция.

Когда интерпретатор, разбирая исходный код, доходит до вызова функции, он, вычислив значения аргументов, начинает выполнять тело вызываемой функции и только после ее завершения переходит к разбору следующей инструкции.

Все довольно просто, хотя проследить поток выполнения в сложных программах зачастую оказывается задачей нетривиальной, т.к. из тела любой функции может быть вызвана другая функция, которая тоже может в своем теле содержать вызовы функций и т.д. Тем не менее, интерпретатор Питона помнит откуда была вызвана каждая функция, и рано или поздно, если походу выполнения не возникнет никаких исключений, он вернется к исходному вызову, чтобы перейти к следующей инструкции.

Итак, представим, что у нас имеется три функции: `f1()`, `f2()` и `f3()`. Причем они вызывают друг друга таким образом:

```
def f1():
    print "f1() begins"
    print "Hello world!"
    print "f1() ends"

def f2():
    print "f2() begins"
    f1()
    print "f2() ends"

def f3():
    print "f3() begins"
    f2()
    print "f3() ends"

print "Main program begins"
f3()
print "Main program ends"
```

Для того, чтобы проследить последовательность выполнения операций, мы добавили команды которые будут сигнализировать, когда начинается и заканчивается каждая функция. Итак, запустим программу и посмотрим, что у нас получилось:

```
$ python prog6.py
Main program begins
f3() begins
f2() begins
f1() begins
Hello world!
f1() ends
f2() ends
f3() ends
Main program ends
```

Разберем полученный результат. Как уже было сказано, определения функций никак не влияют на выполнение основной программы. Исключением является случай, когда интерпретатор сталкивается с синтаксической ошибкой в теле функции: даже если еще не настало время ее выполнения, он выведет сообщение об ошибке и завершит работу.

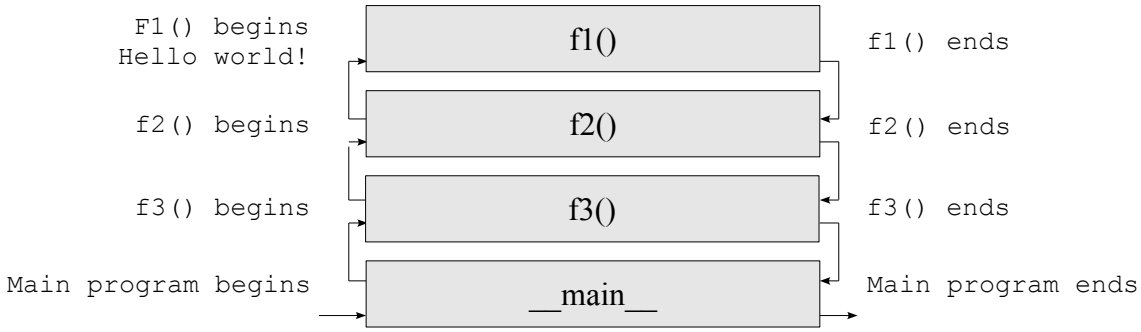
Основная программа, начинается с вывода сообщения о своем начале, затем вызывается функция `f3()`, которая, «представившись» вызывает, функцию `f2()`. Та, в свою очередь, поступает аналогичным образом, вызывая `f1()`. `f1()` тоже выводит сообщение о своем

начале, затем тестовое приветствие миру и сообщение о своем завершении. Поток выполнения возвращается в тело функции `f2()` к тому месту, откуда была вызвана `f1()`, после чего выполняется следующая инструкция, выводящая сообщение о том, что функция `f2()` завершает свою работу. Как вы уже догадались, поток выполнения возвращается в тело функции `f3()`, а после ее завершения – в основную программу.

§3.10. Стековые диаграммы

Для наглядности можно рисовать так называемые *стековые диаграммы*. Называются они так не случайно: они изображают *стек вызовов*.

Стек – это специальная структура данных, напоминающая стопку тарелок, потому что взаимодействие со стеком производится по специальному правилу *LIFO* (от англ. Last In First Out – «последним вошел, первым вышел»). Возвращаясь к аналогии со стопкой тарелок, мы кладем тарелки одну на другую, а берем их, начиная сверху. Немного позже мы научимся моделировать стек и некоторые другие структуры данных, которые часто используются для решения различных задач, а пока, ограничившись интуитивным пониманием стека, изобразим стековую диаграмму для примера из предыдущего раздела.



На данной диаграмме поток выполнения указан стрелками, подпрограммы – прямоугольными блоками; справа и слева от стековой диаграммы дописаны строки, выводимые программой. Внизу «стопки» находится основная программа (она обозначается как `__main__`); соответственно завершиться она не может до тех пор, пока не завершатся все функции, которые в стеке вызовов находятся «над ней». Таким образом, использование стека для отслеживания потока выполнения является наиболее естественным и простым решением. В процессе выполнения программы стек вызовов может разрастаться и опустошаться по многу раз.

Когда в одной из функций возникает исключение, часто бывает довольно трудно понять, при каких условиях (или при каком наборе параметров) произошел сбой, т.к. одна и та же функция может вызываться из различных мест программы по многу раз. Поэтому для облегчения процесса отладки интерпретатор Питона выводит не только номер строки, где возникло исключение, но и весь стек вызовов на момент возникновения исключения. Давайте немного подправим нашу программу так, чтобы в функции `f1()` было сгенерировано исключение деления на ноль, заменив вывод приветствия мира на выражение `a = 1/0`:

```
def f1():
    print "f1() begins"
    a = 1/0
```

```
    print "f1() ends"
...
```

Теперь запустим эту программу:

```
$ python prog6.py
Main program begins
f3() begins
f2() begins
f1() begins
Traceback (most recent call last):
  File "prog6.py", line 17, in ?
    f3()
  File "prog6.py", line 13, in f3
    f2()
  File "prog6.py", line 8, in f2
    f1()
  File "prog6.py", line 3, in f1
    a = 1/0
ZeroDivisionError: integer division or modulo by zero
```

После сообщения о начале выполнения функции `f1()` интерпретатор вывел так называемый *traceback* (русскоязычные программисты обычно так и говорят «трейсбэк» или просто «трейс»). Traceback содержит не только названия функций из стека вызовов, но и номера строк, из которых был произведен каждый вызов. Кроме номеров указывается еще и название файла с программой, ведь проект может содержать не один модуль. Так что найти и исправить ошибку становится гораздо проще.

§3.11. Функции, возвращающие результат

Напоследок познакомимся с еще одной важной возможностью функций. Нетрудно представить себе ситуацию, когда результаты работы функции могут понадобиться для дальнейшей работы программы. Но после завершения работы функции все переменные, которые были инициализированы в ее теле, уничтожаются, т.к. они являются локальными. Поэтому при необходимости вернуть результат работы функции в подпрограмму, из которой она вызывалась, для его дальнейшей обработки применяется команда `return`. Выглядит это приблизительно так:

```
>>> def getSum(x,y):
...     z = x + y
...     return z
...
>>> print getSum(1,3)
```

```
4
>>>
```

Как видите, возвращаемое функцией значение воспринимается интерпретатором, как результат выражения, вызывающего эту функцию.

В Питоне функции способны возвращать несколько значений одновременно[7]. Для примера возьмем функцию вычисления корней квадратного уравнения:

```
>>> def PrintRoots(a, b, c):
...     D = b**2 - 4 * a * c
...     import math
...     x1 = (-b + math.sqrt(D)) / 2 * a
...     x2 = (-b - math.sqrt(D)) / 2 * a
...     return x1, x2
...
>>> print PrintRoots(1.0, 0, -1.0)
(1.0, -1.0)
```

Кроме того, результаты выполнения функции мы можем присваивать сразу нескольким переменным:

```
>>> x1, x2 = PrintRoots(1.0, 0, -1.0)
>>> print "x1 =", x1, "\nx2 =", x2
x1 = 1.0
x2 = -1.0
>>>
```

Скорее всего, вы уже догадались, что точно таким же образом значения возвращают и встроенные функции Питона.

7 Кстати, в Паскале и C++ вернуть из функции сразу несколько значений с помощью стандартных средств невозможно. Это еще одно удобство, предоставляемое Питоном.

Глава 4. Компьютерная графика

/* модуль turtle*/

Глава 5. Логические выражения, условия и рекурсия

§5.1. Комментарии в программах

Любой дурак может написать программу, которую поймет компилятор. Хорошие программисты пишут программы, которые смогут понять другие программисты.

Мартин Фаулер (Martin Fowler), «Рефакторинг» ("Refactoring")

По мере увеличения размеров ваших программ рано или поздно вы столкнетесь с одной проблемой: их станет сложнее читать. В идеале программа должна читаться так же легко, как если бы она была написана на естественном языке, но, с одной стороны, естественные языки все же не имеют такой четкой формальной формы описания объектов и процессов, а с другой – чрезмерная формализованность зачастую приводит к усложнению описания простых вещей. Поэтому для повышения понятности кода, его полезно снабжать комментариями на естественном языке, и большинство языков программирования, не исключая Питон, предоставляют такую возможность.

Комментирование кода считается правилом «хорошего тона», поэтому не забывайте про комментарии – этим вы и себе облегчите жизнь.

Когда над программой работает один программист, то отсутствие комментариев компенсируется хорошим знанием кода (если, конечно, этот код был написан в сознательном состоянии), но при работе в команде, за редкими исключениями, комментарии просто необходимы. Кроме того, через какое-то время вы сами не сможете разобраться в своей программе, если в ней не будет никаких дополнительных замечаний.

В Питоне комментарии помечаются символом `#` – строки, начинающиеся с этого символа, просто игнорируются интерпретатором и никак не влияют на ее трансляцию и выполнение:

```
# Подсчет процентного соотношения двух величин: 20 и 80
print 100 * 20 / 80, "%"
```

Комментарий может следовать и после инструкций, т.е. начинаться не с самого начала строки:

```
>>> print 100 * 20 / 80, "%"   # целочисленное деление
25 %
```

Кроме того, комментариями стоит снабжать и функции. Для этого предусмотрен еще один способ комментирования:

```
def printTwice(value):
    """Описание функции printTwice()
    Данная функция получает значение и выводит его дважды, разделив
    пробелом."""

    print value, value
```

Как видите, комментарии с описанием функций должны находиться сразу после заголовка функции. Они заключаются с двойные кавычки три раза и могут занимать несколько строк. Более того, в Питоне предусмотрена возможность вывода этих комментариев. Для этого достаточно воспользоваться встроенной переменной `__doc__` (начинается и заканчивается парами символов подчеркивания):

```
>>> print printTwice.__doc__
Описание функции printTwice()
    Данная функция получает значение и выводит его дважды, разделив
    пробелом.
```

Этот же комментарий выведет команда справочной системы Питона `help(printTwice)`.

Изучайте описания встроенных функций и при написании своих функций старайтесь комментировать их в том же стиле, чтобы другим программистам было проще сориентироваться – к такой структуре уже все привыкли.

Не забывайте, что плохо документированный код программы является признаком эгоистичного программиста.

§5.2. Простые логические выражения и логический тип данных

В третьей главе мы научились использовать функции, тем получили мощный способ управления потоком выполнения: и наши программы перестали быть линейными. Функции выполняются без проверки каких-либо условий, как только поток выполнения программы доходит до вызова соответствующей функции. Но практически в любой программе возникает потребность в выполнении той или иной части кода программы в зависимости от введенных пользователем данных, результатов промежуточных вычислений и т.п.

Во всех языках программирования высокого (и сверхвысокого) уровня имеется возможность разветвления программы; при этом выполняется одна из ветвей программы в зависимости от истинности или ложности какого-либо условия. Прежде чем переходить к инструкции, осуществляющей выбор ветки программы, по которой пойдет поток выполнения, разберемся с логическими выражениями.

Логическими выражениями называют выражения, результатом которых является *истина* (`True`) или *ложь* (`False`). В простейшем случае любое утверждение может быть истинным или ложным. Например, истинность утверждения «на улице идет дождь» зависит от того, какая на улице погода в данный момент. Или «2+2 равно 4» – истинное, а «2+2 равно 5» – ложное выражение.

Операции сравнения в программировании встречаются чаще других, поэтому с них мы и начнем. Для проверки равенства двух значений в Питоне используется оператор `==` (два знака равенства без пробела между ними).

```
>>> x = 2 + 2
>>> x == 4
True
>>> x == 5
False
```

Другие операции сравнения:

```
>>> x != 5              # x не равен 5
True
>>> x > 5               # x больше 5
False
>>> x < 5               # x меньше 5
True
>>> x >= 4              # x больше либо равен 4
True
>>> x <= 4              # x меньше либо равен 4
True
```

Разумеется, результат сравнения двух значений мы можем записать в переменную:

```
>>> y = x == 5
>>> print y
False
```

Обратите внимание, что приоритет операций сравнения меньше приоритета арифметических операций, но больше, чем у операции присваивания.

Проверим тип переменной, в которую мы записали результат сравнения:

```
>>> type(y)
<type 'bool'>
```

Как видите, в Питоне предусмотрен специальный логический тип данных, но появился он только в версии 2.2.2. До этого в Питоне истине соответствовало число `1`, а ложь кодировалась как `0`; команда `type(4 != 5)` возвращала значение `<type 'int'>`.

Упражнение. *Поэкспериментируйте с преобразованием типа значений в логический тип. Несколько предложений для экспериментов:*

```
>>> bool(2)
>>> bool(-1)
>>> bool(1.0)
>>> bool("string")
>>> bool(true)
```

Объясните полученные результаты.

§5.3. Логические операторы

Логические выражения могут быть более сложными и состоять из нескольких простых выражений. Для объединения простых выражений в более сложные используются *логические операторы*: `and`, `or` и `not`.

Значения их полностью совпадают со значением английских слов, которыми они обозначаются. Выражение `x and y` будет истинным только в том случае, когда `x` – истина **и**

`y` – истина. Во всех остальных случаях выражение будет ложью. Выражение `x or y` будет истиной, если хотя бы один из операндов – истина. Оператор `not` символизирует отрицание: `not x` – истина, если `x` – ложь, и наоборот: `not x` – ложь, если `x` – истина. Обратите внимание, что оператор `not` унарный, т.е. он работает только с одним операндом.

Упражнение. *Чему должен равняться `x` для того, чтобы выражение `not not x` было ложным?*

Для простоты составим таблицы со списков всех возможных комбинаций значений операндов логических выражений и самих выражений (такие таблицы в математической логике называются *таблицами истинности*):

x	y	x and y	x or y
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

x	not x
0	1
1	0

Упражнение. *С помощью таблиц истинности проверьте следующие свойства логических операций:*

`x and y` эквивалентно `y and x`
`x or y` эквивалентно `y or x`
`x and x and x` эквивалентно `x`
`x or x or x` эквивалентно `x`
`not x and not y` эквивалентно `not (x or y)`

§5.4. Выполнение по условию и «пустота»

Теперь мы готовы к изучению *оператора проверки условия*, позволяющего организовать разветвление программы при выполнении того или иного условия. В общем виде он выглядит так:

```
if ЛОГИЧЕСКОЕ_УСЛОВИЕ:
    ПОСЛЕДОВАТЕЛЬНОСТЬ_ВЫРАЖЕНИЙ
```

Первым идет ключевое слово `if` (англ. «если»); за ним – логическое выражение; затем двоеточие, обозначающее конец заголовка оператора, а после него – любая последовательность выражений или *тело условного оператора*, которое будет выполняться в случае, если условие в заголовке оператора истинно. Простой пример:

```
x = 2
if x > 0:
    print "x is positive"

if x < 0:
    print "x is negative"
```

```
print "Stopping..."
```

Сохраним, запустим эту программу и разберем результат ее выполнения:

```
$ python prog7.py
x is positive
Stopping...
```

На первой строке мы определили переменную `x`, присвоив ей значение `2`. Затем с помощью условного оператора проверили на истинность выражение `x > 0` – значение `x` оказалось больше нуля, поэтому Питон выполнил тело этого условного оператора: вывел строку `x is positive`. Далее следует еще один оператор `if`, проверяющий выполнение условия `x < 0`, которое является ложным, и поэтому тело этого условного оператора Питон пропустил и выполнил следующую за ней инструкцию.

Скорее всего, вы уже знаете, где можно использовать условный оператор. Первое, что приходит в голову: функция расчета корней квадратного уравнения. Не все квадратные уравнения имеют действительные корни, поэтому если дискриминант квадратного уравнения меньше нуля, то производить дальнейшие вычисления не нужно. Модифицируем функцию `printRoots()` так, чтобы она правильно обрабатывала ситуацию с отрицательным дискриминантом:

```
>>> def PrintRoots(a, b, c):
...     D = b**2 - 4 * a * c
...     if D < 0:
...         return None, None
...     import math
...     x1 = (-b + math.sqrt(D)) / 2 * a
...     x2 = (-b - math.sqrt(D)) / 2 * a
...     return x1, x2
>>>
>>> print PrintRoots(3, 2, 1)
(None, None)
```

Итак, уравнение $3x^2 + 2x + 1 = 0$ корней не имеет, т.е. его дискриминант оказался отрицательным, поэтому Питон выполнил ветку программы с инструкцией `return None, None`. Здесь мы воспользовались тем, что выполнение подпрограмм завершается сразу после выполнения инструкции `return`.

Обратите внимание на то, что инструкция возвращает не пару значений `(0, 0)`, т.к. с точки зрения математики, это неверно: корни неопределены и функция должна вернуть пустоту. Для этих целей и предусмотрена встроенная константа `None`.

Упражнение. *Напишите функцию `compare(x, y)`, сравнивающую `x` и `y` и возвращающую следующие значения:*

$$compare(x, y) = \begin{cases} 1, если\ x > y, \\ 0, если\ x = y, \\ -1, если\ x < y \end{cases}$$

§5.5. Ввод данных с клавиатуры

Программы, которые мы писали до этого, немного скучные, т.к. они каждый раз делают одно и то же, не спрашивая у пользователя, над какими данными следует произвести те или иные действия.

Но это поправимо. В Питоне имеются встроенные функции, которые позволяют получать данные с клавиатуры. Самая простая из них – `raw_input()`. Вызов этой функции останавливает выполнение программы и заставляет компьютер дождаться, пока пользователь не введет данные с клавиатуры. Завершение ввода осуществляется нажатием клавиши `[Enter]`[8]: сразу после нажатия этой клавиши, функция `raw_input()` завершается и возвращает значение введенное с клавиатуры:

```
>>> text = raw_input()
What are you waiting for?
>>> print text
What are you waiting for?
>>>type(text)
<type 'str'>
```

Прежде чем предоставлять пользователю возможность ввода данных хорошо бы объяснить ему, что именно нужно ввести, с помощью строки *приглашения* (по-английски *prompt*). Для этого достаточно передать эту строку в качестве параметра функции `raw_input()`:

```
>>> name = raw_input("What is your name? ")
What is your name? Arthur, King of the Britons!
>>> print name
Arthur, King of the Britons!
```

Если нам нужно, чтобы пользователь ввел целое число, то мы можем использовать функцию `input()`.

```
>>> prompt = "How old are you?"
>>> age = input(prompt)
How old are you? 17
>>> print age
17
```

Введенная пользователем строка цифр будет приведена к типу `int` и присвоена переменной `age`. Но если пользователь введет строку букв, то возникнет исключительная ситуация (исключение):

8 Или [Return] на компьютерах Apple.

```
>>> age = input(prompt)
How old are you?
I'm 17th years old.
SyntaxError: invalid syntax
```

Чуть позже мы научимся избегать аварийное завершение программы и обрабатывать исключения, а пока перейдем к следующему очень захватывающему разделу (не забудьте выполнить упражнение).

Упражнение. *Измените программу вычисления корней квадратного уравнения так, чтобы пользователь сам мог вводить коэффициенты* `a`, `b` *и* `c`.

§5.6. Альтернативные ветки программы (Chained conditionals)

Условный оператор `if` имеет расширенный формат, позволяющий проверять несколько независимых друг от друга условий и выполнять один из блоков, поставленных в соответствие с этими условиями. В общем виде оператор выглядит так:

```
if ЛОГИЧЕСКОЕ_УСЛОВИЕ_1:
    ПОСЛЕДОВАТЕЛЬНОСТЬ_ВЫРАЖЕНИЙ_1
elif ЛОГИЧЕСКОЕ_УСЛОВИЕ_2:
    ПОСЛЕДОВАТЕЛЬНОСТЬ_ВЫРАЖЕНИЙ_2
elif ЛОГИЧЕСКОЕ_УСЛОВИЕ_3:
    ПОСЛЕДОВАТЕЛЬНОСТЬ_ВЫРАЖЕНИЙ_3
...
else:
    ПОСЛЕДОВАТЕЛЬНОСТЬ_ВЫРАЖЕНИЙ_N
```

Работает эта конструкция следующим образом. Сначала проверяется первое условие и, если оно истинно, то выполняется первая последовательность выражения. После этого поток выполнения переходит строке, которая следует после условного оператора (т.е. за последовательностью выражений N). Если первое условие равно `False`, то проверяется второе условие (следующее после `elif`), и в случае его истинности выполняется последовательность 2, а затем снова поток выполнения переходит к строке, следующей за оператором условия. Аналогично проверяются все остальные условия. До ветки программы `else` поток выполнения доходит только в том случае, если не выполняется ни одно из условий.

Ключевое слово `elif` происходит от англ. «else if» - «иначе если». Т.е. условие, которое следует после него проверяется только тогда, когда все предыдущие условия ложны.

Чтобы было понятнее, напишем небольшую программу, иллюстрирующую использование условного оператора с альтернативными ветками.

```
choice = raw_input('Input your choice, please (1 or 2): ')
if choice == "1":
    function1()
```

```
elif choice == "2":
    function2()
else:
    print "Invalid choice!"
print "Thank you."
```

Сначала программа просит пользователя ввести единицу или двойку, затем условный оператор проверяет введенное значение на равенство со строкой `'1'` (если не помните, функция `raw_input()` возвращает строковое значение). Если условие истинно, то вызывается функция `function1()`, после чего программа выводит строку `"Thank you."` и завершается; в противном случае значение переменной `choice` сравнивается со строкой `'2'`: если условие выполняется, то вызывается функция `function2()`, затем выводится строка `"Thank you."` и программа завершается; иначе – программа выводит сообщение о том, что введенное значение некорректно, затем «благодарность» и завершается.

В англоязычной литературе такие разветвленные условные операторы обычно называют *chained conditionals*, т.е. цепочками условий.

Упражнение. *Измените функцию* `compare(x, y)` *так, чтобы в ней использовался всего один оператор условия.*

§5.7. Пустые блоки

В процессе работы над программой следует стараться после каждого изменения иметь работающую программу, но предположим, что вы запланировали использование условного оператора с несколькими условиями, успев написать только один из блоков условного оператора. При этом встает вопрос, как ее отладить, если программа не выполняется из-за синтаксической ошибки. Например, рассмотрим программу:

```
choice = raw_input('Enter your choice, please:')
if choice == "1":
    function1_1()
    finction1_2()
elif choice == "2":
elif choice == "3":
elif choice == "4":
else:
    print "Invalid choice!"
```

Блоки для случаев, когда значение переменной `choice` равно `"2"`, `"3"` или `"4"`, еще не написаны, поэтому программа не выполняется из-за ошибки `Expected an indented block`.

Избежать этого можно с помощью ключевого слова `pass`, которое можно вставить на место отсутствующего блока:

```
choice = raw_input('Enter your choice, please:')
if choice == "1":
```

```
    function1_1()
    finction1_2()
elif choice == "2":
    pass
elif choice == "3":
    pass
elif choice == "4":
    pass
else:
    print "Invalid choice!"
```

Это ключевое слово может использоваться и в качестве тела функции:

```
def f1():
    pass
```

Эта функция ничего не делает – поток выполнения, «заглянув в нее», сразу переходит к выполнению следующей за вызовом этой функции инструкции.

§5.8. Вложенные условные операторы (Nested conditionals)

В блоках условного оператора тоже могут встречаться условные операторы, ведь они, с точки зрения Питона, ничем не отличаются от других инструкций. Например:

```
if x == y:
    print x, "and", y, "are equal"
else:
    if x < y:
        print x, "is less than", y
    else:
        print x, "is greater than", y
```

Первый логический оператор содержит две ветви, вторая из которых тоже разветвляется. Но не стоит слишком злоупотреблять вложенными логическими операторами, т.к. это может привести к ухудшению читабельности программы. Здесь на помощь нам приходят логические операции. Рассмотрим пример программы:

```
if x < 1:
    pass
else:
    if x < 10:
        print "x is between 1 and 10"
```

Этот код работает так: если `x < 1`, то ничего не происходит, т.к. поток выполнения, встретив инструкцию pass, перейдет к строке, следующей после условного оператора; в противном случае выполнится проверка логического выражения `x < 10`, и если оно

истинно, то интерпретатор выведет строку `"x is between 1 and 10"`. Можно написать семантически идентичный условный оператор, но гораздо короче:

```
if not x < 1 and x < 10:
    print "x is between 1 and 10"
```

Можно ли сделать его еще проще? Хотелось бы избавиться от логического оператора отрицания в выражении, проверяемом оператором `if`.

Выражение `not x < 1` истинно, если `x` не меньше единицы, или `x` больше либо равно единицы – `x >= 1`:

```
if x >= 1 and x < 10:
    print "x is between 1 and 10"
```

В такой записи незнакомый с кодом программист разберется гораздо быстрее, не так ли? Но не будем на этом останавливаться. В математике имеется более короткий способ записи условия принадлежности `x` некоторому отрезку числовой прямой: $1 \leqslant x < 10$. Адаптируем ее под операторы сравнения, которые имеются в языке Питон:

```
>>> x = 3
>>> if 1 <= x < 10:
...     print "x is between 1 and 10"
...
x is between 1 and 10
>>>
```

Питон прекрасно нас понял. Надо сказать, что такой возможности нет ни в Паскале, ни в C++, ни в других распространенных языках.

Что ж, пришло время попрактиковаться.

Упражнение. *Упростите выражение:*

```
if x < -5:
    print "x is NOT between -5 and 5"
else:
    if x < 5:
        print "x is between -5 and 5"
    else:
        print "x is NOT between -5 and 5"
```

Упражнение. *Объясните почему условие 10 < x < 0 никогда не будет истинным.*

§5.9. Рекурсия

Разобравшись с логическими операциями и условным оператором, перейдем к изучению очень мощной концепции, которая называется *рекурсией*. Для того, чтобы лучше понять, где она применяется, рассмотрим задачу, которая сама приведет нас к ней.

Помимо привычных нам арифметических в математике есть и другие операции. В частности, факториал: например, $3!=1*2*3=6$. *3!* читается как «три факториал». В общем виде $n!=1*...*n$, где $n\in\mathbb{N}\cup\{0\}$ [9], причем $0!=1$.

Растут значения факториала очень быстро, например $5!=120$, $7!=5040$, а $10!=3628800$. Поэтому логично было бы поручить вычисление факториалов компьютеру.

Прежде чем приступать к реализации функции факториала давайте проанализируем формулу, по которой она вычисляется. При условии, что мы не знаем чему равно число *n* применить формулу $n!=1*...*n$ «в лоб» не получится. Поэтому давай-те ее проанализируем:

$n!=1*2*...*n$,

$(n+1)!=1*2*...*n*(n+1)=n!*(n+1)$, иначе говоря:

$n!=(n-1)!*n$

Это уже что-то. По сути, зная значение $(n-1)!$, мы можем вычислить *n!* всего за одну операцию. Аналогично, $(n-1)!=(n-2)!*(n-1)$. Попробуем реализовать эту идею на Питоне. Мы уже говорили о том, что из тела одной функции можно вызывать другую функцию. Почему бы не попробовать вызвать из функции эту же самую функцию? Это как раз то, что нам надо, давайте попробуем:

```
>>> def fact(n):
...     return fact(n-1)*n
...
>>> fact(3)
  File "<stdin>", line 2, in fact
  [...]
  File "<stdin>", line 2, in fact
RuntimeError: maximum recursion depth exceeded
```

Хм, вызывать функцию из самой себя, видимо, можно, но интерпретатор вывел длинный список обратной трассировки и сообщение об исключении: `RuntimeError: maximum recursion depth exceeded` – «Достигнута максимальная глубина рекурсии». Вот и упоминание рекурсии. Сформулируем определение – возможно, это нам поможет в поиске решения проблемы.

Рекурсивными функциями являются функции, получаемые в результате конечного числа применений самих себя для целочисленных значений аргумента (параметра). Важно заметить то, что такие функции применимы лишь к целочисленным значениям, но не к дробным.

Мы передали целочисленный аргумент – `3`, но возникло исключение. Видимо, мы имеем дело с семантической ошибкой: программа делает не то, что мы подразумевали. Давайте попробуем повторить выполнение нашей программы в уме.

9 Число n принадлежит множеству натуральных чисел, объединенному с нулем, иначе говоря, может быть равным 0, 1, 2, 3 и т.д.

Итак, вызов `funct(3)` должен вернуть `funct(2)*3`, далее `funct(2)` – это все равно что `funct(1)*2`; затем идет вызов `funct(0)`, потом `funct(-1)`. Но `n` – должно быть натуральным числом или нолем! Получается, что, дойдя до граничного значения области допустимых значений `n`, наша программа продолжает выполнять вызов функции `funct()` как ни в чем ни бывало, и даже не думает останавливаться. Более того, программа «не знает» чему равен *0!* – мы забыли это указать, но все поправимо.

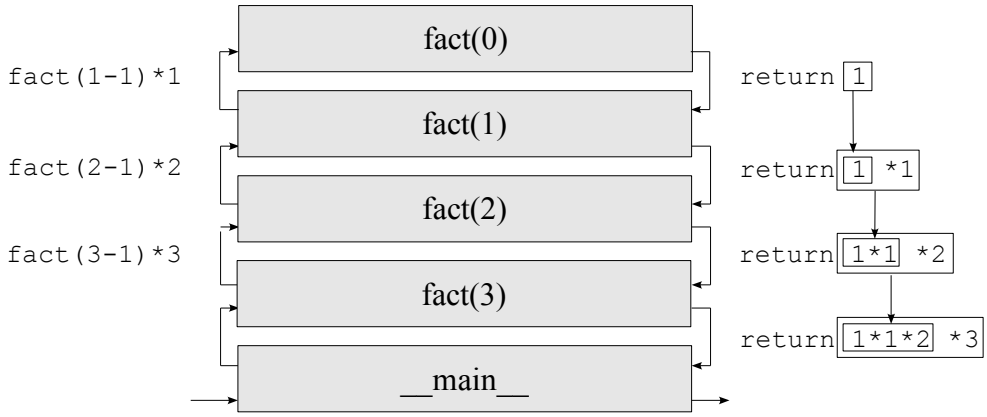
```
>>> def fact(n):
...     if n == 0:
...         return 1
...     return fact(n-1)*n
...
>>> fact(3)
6
```

Работает! Осталось только проверить, как функция поведет себя при некорректных значениях параметра (отрицательных или дробных, например), так что не забудьте выполнить упражнение.

Упражнение. *Проверьте, как функция `fact()` будет вести себя с отрицательными и дробными значениями параметра. Объясните, какие проблемы возникли и почему. Модифицируйте функцию так, чтобы избежать их*[10].

§5.10. Стековые диаграммы рекурсивных вызовов

Разберемся, какой путь проделывает поток выполнения в процессе вычисления факториала. Изобразим стековую диаграмму для вызова `fact(3)`.



Итак, теперь все должно быть понятно. До добавления условия возврата `1` при `x == 0` стек вызовов рос бесконечно, поэтому возникало исключение. Когда же мы добавили это условие, все встало на свои места: сначала стек рекурсивных вызовов увеличивается, пока не дойдет до граничного значения параметра, а затем начинает опустошаться – поток выполнения начинает возврат обратно в главную программу, собирая ответ по кусочкам (кстати, термин «рекурсия» происходит от позднелатинского «recursio» – «возвращение»). При этом результат каждого вызова передается в тело функции, находящейся в стеке на

10 Имейте ввиду, что функция может не справиться со слишком большими значениями (например, при n > 1000). Это связано с ограничениями, присущими вычислительной технике. Этот случай мы разберем в последующих разделах.

уровень ниже, где используется в дальнейших вычислениях. В результате этого трюка мы и получаем ответ.

Можно ли улучшить этот алгоритм? Вы, наверное, обратили внимание, что на втором шаге возврата мы умножаем `1` на `1` – бессмысленная операция. Кроме того, каждый рекурсивный вызов требует выделения некоторого количества памяти, да и сам вызов занимает какое-то время.

Вариант решения: изменить условие, при котором будет начинаться возврат на `x == 1`. Проверим его с помощью граничных значений: что произойдет при вызове `fact(0)`? Программа зациклится. Попробуем так:

```
def fact(n):
    if type(n) != type(1) or n < 0:    # Проверка корректности n
        return None
    if n == 1 or n == 0:
        return 1
    return fact(n-1)*n
```

Такой подход позволит избежать лишний рекурсивный вызов, но увеличивается сложность проверяемого условия, что сказывается на скорости выполнения программы. Конечно, на современных компьютерах разница не заметна, но это противоречие встречается довольно часто: программистам приходится выбирать между экономией памяти и скоростью выполнения. Обычно в такой ситуации приблизительно оценивается, как часто выполняется блок кода – от этого зависит влияние его усложнения на общее время выполнения программы.

Упражнение. *Как вы думаете, стоит ли в случае функции `fact()` производить лишнюю проверку? Объясните почему.*

§5.11. Максимальная глубина рекурсии

Мы уже сталкивались с бесконечной рекурсией, когда рекурсивная функция по тем или иным причинам не достигала условия возврата (в англоязычной литературе *base case* – «базовый случай»). Но в большинстве программных сред такие программы, конечно же, бесконечно выполняться не будут в силу конечного размера памяти компьютера. В Питоне для предотвращения нарушений работы компьютера, на котором выполняется программа, тоже имеется специальное ограничение – *maximum recursion depth* (*максимальная глубина рекурсии*).

Максимальную глубину рекурсии можно изменять, т.к. некоторые задачи требуют большего максимального размера стека вызовов. Для этого в модуле `sys` имеется две функции: `sys.getrecursionlimit()`, возвращающая текущую максимальную глубину рекурсии и `sys.setrecursionlimit()`, изменяющая ее значение.

```
>>> import sys
>>> sys.getrecursionlimit()
1000
>>> sys.setrecursionlimit(5)
>>> sys.getrecursionlimit()
```

```
5
>>>
```

По умолчанию, максимальная глубина рекурсии равна 1000.

Упражнение. *Напишите простую функцию, вызывающую саму себя, и поэкспериментируйте с изменением максимальной глубины рекурсии. Прокомментируйте свои результаты.*

§5.12. Числа Фибоначчи

Леонардо Фибоначчи (Леонардо Пизанский) – крупный средневековый итальянский математик (1170-1240), автор «Книги об абаке» (1202), которая несколько веков оставалась основным сборником сведений об арифметике и алгебре. Сегодня имя Фибоначчи чаще всего упоминается в связи с числовой последовательностью, которую он обнаружил, изучая закономерности живой природы.

Ряд этих чисел образуется следующими значениями 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 и так далее – каждое последующее число ряда получается сложением двух предыдущих.

Особенностью ряда чисел Фибоначчи является то, что по мере роста номера числа в последовательности, отношение предшествующего члена к последующему приближается к значению – 0.618[11] (древние греки называли это число «золотым сечением»), а последующего к предыдущему – к 1.618. «Золотым» это соотношение называют из-за того, что на нем же основан принцип музыкальной, цветовой и геометрической гармонии.

[NOTE: Про числа Фибоначчи можно много интересного написать, но я оставил это на потом]

Давайте напишем функцию, рассчитывающую n-й элемент ряда Фибоначчи. Итак, мы имеем обобщенную формулу: $f_1 = f_2 = 1, f_n = f_{n-1} + f_{n-2}, n \in \{\mathbb{N} \cup 0\}$. Можем ли мы применить здесь рекурсию?

Каждое число Фибоначчи представляет собой сумму двух предыдущих чисел Фибоначчи, которые рассчитываются по той же формуле, все числа Фибоначчи целые и положительные, кроме того, мы имеем граничное значение. Так что применить рекурсию в данном случае вполне можно.

```
>>> def fibonacci(n):
...     if n == 0:
...         return 0
...     if n == 1 or n == 2:
...         return 1
...     return fibonacci(n-1) + fibonacci(n-2)
...
>>> fibonacci(7)
13
```

11 Точное значение равно $\frac{\sqrt{5}-1}{2}$.

Замечательно. Теперь осталось добавить проверку типа и положительности параметра функции и готово.

```
def fibonacci(n):
    if type(n) != type(1) or n < 0:    # Проверка корректности n
        return None
    if n == 0:
        return 0
    if n == 1 or n == 2:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

Упражнение. *Создайте рекурсивную функцию, возвращающую сумму* $\sum_{i=1}^{n} i^3 = 1^3 + \ldots + n^3$.

Глава 6. Циклы

§6.1. Оператор цикла while

В предыдущей главе мы столкнулись с необходимостью повторения некоторых блоков кода в процессе выполнения вычислений. В некотором смысле, мы справились с этой проблемой, использовав рекурсию, но ее использование далеко не всегда оправдано. Во-первых, разобраться, как работает рекурсивная функция не так-то просто, а это означает, что расширение функциональности программы будет затруднено, особенно если над ней будет работать не один программист, а целая команда. Во-вторых, рекурсивные функции очень «прожорливы» – для их выполнения требуется много оперативной памяти, да и ограничение максимального размера стека вызовов тоже создает некоторые трудности.

Разумеется, не мы первые так подумали. В первой главе мы уже говорили о том, что еще во времена низкоуровневых языков программирования была замечена необходимость в повторах блоков кода. Поэтому все высокоуровневые языки программирования имеют специальные операторы, позволяющие организовывать циклы в программах. Такие операторы обычно так и называют: *операторами цикла*.

Каждый циклический оператор имеет *тело цикла* – некий блок кода, который интерпретатор будет повторять пока условие повторения цикла будет оставаться истинным.

В языке Питон оператор цикла выглядит так:

```
while УСЛОВИЕ_ПОВТОРЕНИЯ_ЦИКЛА:
    ТЕЛО_ЦИКЛА
```

Очень похоже на оператор условия – только здесь используется другое ключевое слово: `while` (англ. «пока»). Где его можно использовать? Первое, что приходит в голову: повтор ввода данных пользователем, пока не будет получено корректное значение:

```
correct_choice = False
while not correct_choice:
    choice = raw_input("Enter your choice, please (1 or 2):")
    if choice == "1" or choice == "2":
        correct_choice = True
    else:
        print "Invalid choice! Try again, please."
print "Thank you."
```

Перед началом цикла мы определили логическую переменную `correct_choice`, присвоив ей значение `False`. Затем оператор цикла проверяет условие `not correct_choice`: отрицание `False` – истина. Поэтому начинается выполнение тела цикла: выводится приглашение `"Enter your choice, please (1 or 2):"` и ожидается ввод пользователя. После нажатия клавиши `[Enter]` введенное значение сравнивается со строками `"1"` и `"2"`, и если оно равно одной из них, то переменной `correct_choice` присваивается значение `True`. В противном случае программа выводит сообщение `"Invalid choice! Try again, please."`. Затем оператор цикла снова проверяет условие и если оно по-прежнему истинно, то тело цикла повторяется снова, иначе поток выполнения переходит к следующему оператору, и интерпретатор выводит строку `"Thank you."`. Как видите, все довольно просто.

Упражнение. *Наберите программу в файле и поэкспериментируйте с ней: попробуйте вводить различные строки и цифры. Правильно ли работает программа?*

§6.2. Счетчики

Еще один вариант использования оператора цикла – вычисление формул с изменяющимся параметром. В одном из упражнений предыдущей главы было задание с помощью рекурсивной функции вычислить формулу $\sum_{i=1}^{n} i^3 = 1^3 + \ldots + n^3$ при заданном *n*. В данном случае изменяющимся параметром является *i*, причем *i* последовательно принимает значения в диапазоне от *1* до *n*.

С помощью оператора цикла `while` решение будет выглядеть так:

```
n = input("Input n, please:")
sum = 0
i = 1
while i <= n:
    sum = sum + i**3 #Тоже самое можно записать короче: sum += i**3
    i = i + 1        #Аналогично: i += 1
print "sum = ", sum
```

Разберемся, как работает эта программа. Сначала пользователь вводит `n` – граничное значение `i`. Затем производится инициализация переменных `sum` (в ней будет храниться результат, начальное значение – `0`) и счетчика `i` (по условию, начальное значение – `1`). Далее начинается цикл, который выполняется, пока `i <= n`. В теле цикла в переменную `sum` записывается сумма значения из этой переменной, полученная на предыдущем шаге, и значение `i`, возведенное в куб; счетчику `i` присваивается следующее значение. После завершения цикла выводится значение `sum` после выполнения последнего шага.

Следующее значение счетчика получают прибавлением к его текущему значению *шага цикла* (в данном случае шаг цикла равен `1`). Шаг цикла при необходимости может быть отрицательным, и даже дробным. Кроме того, шаг цикла может изменяться на каждой *итерации* (т.е. при каждом повторении тела цикла).

Теперь давайте попробуем оптимизировать нашу программу. Итак на каждой итерации в теле цикла мы вычисляем следующее значение счетчика. Но на последнем шаге мы вычислем его лишний раз – его новое значение уже использоваться не будет. Конечно, эта операция не требует каких-то особых ресурсов, но привычку обращать внимание на подобные вещи следует вырабатывать, т.к. впоследствии вы, наверняка, будете иметь дело с более сложными операциями. Как избежать выполнения лишнего вычисления? Что если вычислять значение i до вычисления формулы? Давайте попробуем.

```
n = input("Input n, please:")
sum = i = 0
while i <= n:
    i += 1
```

```
    sum += i**3
print "sum = ", sum
```

Обратите внимание, на запись вида `sum = i = 0`. Она работает аналогично простому присваиванию значения переменной, но в данном случае обе переменные получают одинаковое начальное значение.

Итак, теперь начальное значение `i` равно `0`, но на первом шаге мы сразу же прибавляем к нему единицу. В результате программа выдает те же результаты, что и предыдущая ее модификация. Изменилось ли количество операций?

Нет, не изменилось, но выглядит, пожалуй, немного изящнее: после завершения цикла `i = n`. Это может помочь избежать путаницы, если `i` будет использоваться далее в программе.

Упражнение. *Напишите программу, которая с помощью оператора цикла `while` выводит все четные числа от `0` до заданного пользователем `n`.*

---

§6.3. Бесконечные циклы

Изучая рекурсивные функции, мы столкнулись с проблемой бесконечного повторения блока кода (бесконечной рекурсией), которое возникает, если условие возврата не достижимо, например, из-за семантической ошибки в программе. Циклы тоже могут выполняться бесконечно и тоже, чаще всего, из-за семантических ошибок. Простейший пример:

```
i = 0
while i < 10:
    print i
```

Этот цикл будет выполняться бесконечно, т.к. условие `i < 10` всегда будет истинным, ведь значение переменной `i` не изменяется: такая программа будет выводить бесконечную последовательность нулей. В отличие от рекурсивных функций, у циклов нет ограничения количества повторений тела цикла, поэтому программа с бесконечным циклом будет работать непрерывно, пока мы не произведем аварийный останов нажатием комбинации клавиш `[Ctrl+C]`, не уничтожим соответствующий процесс средствами операционной системы, не будут исчерпаны доступные ресурсы компьютера (например, может быть достигнуто максимальное допустимое количество открытых файлов), или не возникнет исключение. В таких ситуациях программисты говорят, что программа *зациклилась*.

Найти зацикливание в программе иногда бывает не так-то просто, ведь в реальных программах обычно используется множество циклов, каждый из которых потенциально может выполняться бесконечно. Некоторые рекомендации по поиску зацикливаний вынесены в *Приложение А*.

Тем не менее, отсутствие ограничения на количество повторов тела цикла дает нам новые возможности. Многие программы представляют собой цикл, в теле которого производится отслеживание и обработка действий пользователей или запросов из других программных и автоматизированных систем. При этом такие программы могут работать без перебоев очень длительные сроки, иногда годами!

Циклы – это очень мощное средство реализации ваших идей, но они требуют некоторой внимательности.

§6.4. Альтернативная ветка цикла while

Язык Питон имеет много интересных и полезных особенностей, одной из которых является расширенный вариант оператора цикла:

```
while УСЛОВИЕ_ПОВТОРЕНИЯ_ЦИКЛА:
    ТЕЛО_ЦИКЛА
else:
    АЛЬТЕРНАТИВНАЯ_ВЕТКА_ЦИКЛА
```

Пока выполняется условие повторения тела цикла, оператор `while` работает так же, как и в обычном варианте, но как только условие повторения перестает выполняться, поток выполнения направляется по альтернативной ветке `else` – так же, как в условном операторе `if`, оно выполнится всего один раз.

```
>>> i = 0
>>> while i < 3:
        print i
        i += 1
    else:
        print "end of loop"


0
1
2
end of loop
>>>
```

Надо сказать, что другие языки программирования легко обходятся без альтернативных веток в операторах цикла – это приятная, но необязательная возможность.

Упражнение. *Выясните, как поведет себя оператор, если условие повторения цикла будет ложным.*

§6.5. Табулирование функций

С помощью оператора цикла `while` удобно строить таблицы значений различных функций. Большинство учебников по высшей математике имеют приложения с такими таблицами значений. До того, как появились компьютеры, математикам приходилось составлять таблицы значений тригонометрических, логарифмических и других функций вручную – это очень трудоемкий и утомительный процесс, что, в свою очередь, увеличивает вероятность возникновения ошибки в расчетах[12].

По сути такие таблицы представляют собой список значений функции при различных значениях ее параметра. Поэтому, когда науке стали доступны вычислительные машины, первой реакцией было желание автоматизировать *табулирование функций* (т.е. процесс построения таблиц значений функции). Логично для этих целей использовать циклы.

12 Конечно, таблицам, имеющимся в учебниках, доверять можно, т.к. они перепроверялись множество раз.

Рассмотрим такую программу:

```
import math
x = 1.0
while x < 10.0:
    print x, "\t", math.log(x)
    x += 1.0
```

Результат ее работы будет выглядеть так:

```
1.0     0.0
2.0     0.69314718056
3.0     1.09861228867
4.0     1.38629436112
5.0     1.60943791243
6.0     1.79175946923
7.0     1.94591014906
8.0     2.07944154168
9.0     2.19722457734
```

Строка `"\t"` обозначает *знак табуляции*. Благодаря нему значения выстраиваются в два столбца.

Разберем, как эта программа работает. Параметр `x` изменяется от `1.0` с шагом `1.0`, пока он меньше `10.0`. В теле цикла выводится текущее значение параметра x, затем знак табуляции и результат вычисления функции `math.log(x)`, т.е. натуральный логарифм от `x` ($\log_e x = \ln(x)$). При необходимости вычисления логарифма по основанию *2* мы можем воспользоваться формулой:

$$\log_2 x = \frac{\ln x}{\ln 2}$$

Наша программа будет выглядеть так:

```
import math
x = 1.0
while x < 10.0:
    print x, "\t", math.log(x)/math.log(2)
    x += 1.0
```

Результат будет таким:

```
1.0     0.0
2.0     1.0
3.0     1.58496250072
4.0     2.0
5.0     2.32192809489
```

```
6.0     2.58496250072
7.0     2.80735492206
8.0     3.0
9.0     3.16992500144
```

Как видите, *1*, *2*, *4* и *8* являются степенями *2*. Модифицируем нашу программу, чтобы найти другие степени *2*:

```
import math
x = 1.0
while x < 100.0:
    print x, "\t", math.log(x)/math.log(2)
    x *= 2.0
```

Таким образом, мы получили:

```
1.0     0.0
2.0     1.0
4.0     2.0
8.0     3.0
16.0    4.0
32.0    5.0
64.0    6.0
```

Благодаря символу табуляции позиция второй колонки не зависит от ширины первой – это хорошо видно на последних трех значениях.

Современные математики редко используют таблицы значений, но современным компьютерным специалистам (и программистам, в частности) полезно знать степени двойки, т.к. компьютер работает в двоичной арифметике, и это является причиной некоторых особенностей хранения и обработки данных.

Упражнение. *Измените программу так, чтобы программа вывела все степени 2 до 10-й включительно. Постарайтесь запомнить результаты. Почему в килобайте не 1000 байт, а 1024?*

§6.6. Специальные и экранируемые символы

В этом разделе мы немного отвлечемся от циклов, чтобы внести ясность в вопрос специальных текстовых последовательностей, с одной из которых мы уже столкнулись в предыдущем разделе: `"\t"`. Дело в том, что в кодовых таблицах (наборах цифровых кодов, обозначающих различные символы, которые компьютер может выводить на экран) есть целая группа так называемых *непечатаемых символов*. Непечатаемые символы используются для управления вводом/выводом. Самые часто используемые из них: знак табуляции, перенос на новую строку и знак «возврата каретки». Т.к. в кодовой таблице нет символов, отображаемых на экране, для их обозначения придумали специальные последовательности:

Последовательность	Назначение
`\t`	Табуляция

Последовательность	Назначение
`\n`	Перевод на новую строку
`\r`	Возврат «каретки» (курсора) в начало строки[13]

Таким образом, если в программе нужно вывести текст из нескольких строк, то на помощь приходят специальные последовательности:

```
>>> print "hello\rH\n\tworld"
Hello
        world
```

Разберем, что происходит при выводе строки `"hello\rH\n\tworld"`. Сначала выводится строка `"hello"` – только после нее интерпретатор встречает первую специальную последовательность: `"\r"` – символ возврата «каретки» на начало строки. Затем Выводится символ `"H"`, но происходит это на позиции первого символа! При этом уже имеющийся на этой позиции символ заменяется новым[14]. После этого последовательности `"\n"` и `"\t"` дают указание произвести перевод на новую строку и оставить отступ в одну табуляцию. Далее выводится оставшаяся часть строки `"world"`.

Разумеется, не стоит применять специальные последовательности там, где это не обязательно: например, проще было исправить первый символ строки вместо добавления `"\rH"`. Иначе вашу программу будет сложнее читать.

13 Термин «каретка» возник с появлением механических пишущих машинок, которые были устроены так, что после перехода на новую строку нужно было возвращать специальную подвижную часть механизма в положение, соответствующее началу строки. Эта подвижная часть и называлась «кареткой». В процессе печати «каретка» перемещалась справа налево вместе с листом бумаги, «подставляя» место для следующего символа под рычажки с их оттисками, приводимые в в движение клавишами машинки.

14 Кстати, опытные секретарши, работавшие с пишущими машинками, так исправляли опечатки в документах: они перемещали «каретку» в нужное положение и перепечатывали символ поверх старого, который обычно немного подтирали стирательной резинкой или закрашивали корректором.

Упражнение. *Напишите выражение на языке Питон, которое будет выводить строку такого вида:*

```
I can use
     tabs and
          new lines
in program output.
```

Теперь рассмотрим такую ситуацию: предположим, что нам необходимо вывести, например, такую строку `"Main \template/"`.

```
>>> print "Main \template/"
Main    emplate/
```

Интерпретатор распознал последовательность `"\t"` и вставил специальный символ, а это не то, чего мы хотели добиться.

В подобных ситуациях перед специальным символом `"\"` ставится так называемый *экранирующий символ*: еще одна косая черта `"\"`.

```
>>> print "Main \\template/"
Main \template/
```

Теперь все в порядке. А как вывести кавычки?

```
>>> print "I said: "Hello world""
  File "<stdin>", line 1
    print "I said: "Hello world""
                         ^
SyntaxError: invalid syntax
```

Интерпретатор воспринимает вторую кавычку, как завершение строкового значения, но следующее за ней продолжение строки некорректно с точки зрения синтаксиса Питона. Попробуем *экранировать* кавычки:

```
>>> print "I said: \"Hello world\""
I said: "Hello world"
```

Еще один вариант решения проблемы – использование одинарных кавычек:

```
>>> print 'I said: "Hello world"'
I said: "Hello world"
```

Главное, чтобы каждая неэкранированная кавычка имела пару и при этом не нарушалась синтаксическая структура выражения. Поэтому, работая с кавычками, будьте внимательны.

Кстати, в тексте программ тоже используются специальные символы. Более того, символ переноса на новую строку является признаком завершения выражения, или, иначе говоря, разделителем выражений. Интересно, что в Питоне его тоже можно экранировать.

Зачем? Ну, например, если выражение слишком длинное, и для удобства чтения программы вы хотите разбить его на несколько строк. Вот пример из реальной программы:

```
print "Значение переменной n должно быть целым положительным" \
      + " числом,\nпопробуйте еще раз."
```

В конце первой строки стоит символ `"\"` (обратите внимание: он находится за пределами строкового значения), который экранирует символ переноса строки. Таким образом разделитель выражений Питона игнорируется интерпретатором, и выражение благополучно продолжается на следующей строке.

§6.7. Числа Фибоначчи и оператор цикла while

Мир так устроен, что всегда существует множество путей достижения одной и той же цели, всегда есть выбор. В программировании это свойство тоже нашло свое отражение. Профессиональным программистам постоянно приходится анализировать плюсы и минусы различных подходов к решению задач и выбирать наиболее подходящие варианты в той или иной ситуации, стараясь выбрать наиболее оптимальное соотношение производительности, требуемого объема памяти, а также простоты использования и поддержки программы.

В предыдущей главе мы написали программу, вычисляющую числа Фибоначчи с помощью рекурсивной функции, но она имела ограничение, связанное с максимальной глубиной рекурсии. Это существенный недостаток (хотя решение, бесспорно, красивое).

Но теперь мы знаем о циклах и разберем еще один способ решения задачи нахождения чисел Фибоначчи. Итак, мы имеем следующую формулу: $f_1 = f_2 = 1, f_n = f_{n-1} + f_{n-2}, n \in \mathbb{N}$.

В ней используется три переменные. Обозначим их так: `fn` – результирующее значение функции, `fn1` и `fn2` – промежуточные значения функции используемые в формуле. Знак минуса мы не можем использовать в названиях переменных, помните? Определившись с обозначениями перейдем к анализу задачи.

На первых двух шагах значение функции равно `1`. Для расчета последующих чисел из ряда Фибоначчи применим оператор цикла `while`. На каждом шаге цикла будем вычислять значение `fn = fn1 + fn2`, а затем изменять значения в переменных `fn1` и `fn2`, подготавливая их для следующего шага цикла: на следующей итерации `fn2` будет равняться `fn1`, а `fn1` – `fn`. Разумеется, для контроля количества итераций цикла мы будем использовать переменную-счетчик `i`. Вот, что у нас должно получиться:

```
def fibonacciWithWhileLoop(n):
    """Функция вычисления чисел Фибоначчи с использованием
    оператора цикла while."""
    fn = fn1 = fn2 = 1
    i = 3
    while i <= n:
        fn = fn1 + fn2
        fn2 = fn1
        fn1 = fn
```

```
        i += 1
    return fn
```

Упражнение. *Проанализируйте процесс выполнения данной функции. Правильно ли она работает? Изменяется ли размер стека вызовов при работе данной функции?*
Что произойдет, если функции `fibonacciWithWhileLoop()` *передать в качестве параметра число 2.4? Доработайте программу так, чтобы этот случай корректно обрабатывался.*

Выполнили задание? Если лень заниматься этим сейчас, то лучше отложить книгу и пойти прогуляться. В лени и усталости нет ничего плохого – это вполне естественно для человека, и глупо это отрицать. Но «халтурить» не стоит. Как говорится, тяжело в учении, легко в бою.

Если с заданием справились, идем дальше.

§6.8. Вложенные операторы цикла и двумерные таблицы

Как вы уже, наверное, догадались, подобно логическим операторам, циклы могут быть вложены друг в друга. При этом циклы могут использовать различные переменные-счетчики.

Простейшее применение вложенных операторов цикла – построение двумерных таблиц, например:

```
i = 1
while i <= 10:
    j = 1
    while j <= 10:
        print i * j, "\t",
        j += 1
    print
    i += 1
```

Разберемся с тем, как работает эта программа. Цикл, проверяющий условие `i <= 10`, отвечает за повторение строк таблицы. В его теле выполняется второй цикл, который выводит произведение `i` и `j` десять раз подряд, разделяя знаками табуляции полученные результаты. При этом запятая, завершающая выражение, запрещает интерпретатору переводит курсор на новую строку. Также обратите внимание на команду `print` без параметра, следующую после внутреннего цикла. Она производит перевод на новую строку. В результат выполнения данной программы мы получим следующую таблицу:

```
1    2    3    4    5    6    7    8    9    10
2    4    6    8    10   12   14   16   18   20
3    6    9    12   15   18   21   24   27   30
4    8    12   16   20   24   28   32   36   40
5    10   15   20   25   30   35   40   45   50
6    12   18   24   30   36   42   48   54   60
7    14   21   28   35   42   49   56   63   70
```

```
8    16   24   32   40   48   56   64   72   80
9    18   27   36   45   54   63   72   81   90
10   20   30   40   50   60   70   80   90   100
```

Первый столбец и первая строка содержат числа от 1 до 10. На пересечении i-той строки и j-того столбца находится произведение `i` и `j`. Мы получили таблицу умножения.

Если мы добавим еще один цикл, то получим трехмерную таблицу произведений трех чисел. Выводить на экран ее, правда, будет неудобно – «послойно» разве что. Таким образом вложенные циклы позволяют строить таблицы любой размерности, что очень часто применяется в сложных научных расчетах. И в трехмерной графике, кстати, тоже.

§6.9. Классификация операторов цикла

В различных языках программирования существует четыре типа циклов. Мы рассмотрели один из них на примере оператора `while`. Такие циклы называются *циклами с предусловием*. Их характерная черта заключается в том, что Условие повторения цикла проверяется до выполнения его тела.

Второй тип циклов – *циклы с постусловием*. Их отличие состоит в том, что они имеют условие выхода из цикла, которое проверяется после выполнения тела цикла. Т.е. в таких циклах обязательно выполняется хотя бы одна итерация. Например, в языке Pascal цикл с постусловием выглядит так:

```
do                                {Цикл с постусловием в языке Pascal}
    ТЕЛО_ЦИКЛА
until УСЛОВИЕ_ВЫХОДА;
```

В Питоне цикла с постусловием нет, т.к. он легко заменяется циклом с предусловием – достаточно на первой итерации сделать истинным условие повторения цикла.

Еще один тип циклов – *арифметические циклы*. Такие циклы специально предназначены для работы с переменными-счетчиками. Например, в Pascal'е арифметический цикл выглядит так:

```
for i:=0 to[15] 9 do              {Арифметический цикл в языке Pascal}
    ТЕЛО_ЦИКЛА;
```

В C арифметический цикл немного сложнее – в его заголовке можно отдельно указывать начальное значение счетчика (причем, их может быть несколько), условие повторения цикла и операции, которые должны выполняться в конце каждой итерации (например, приращение счетчика):

```
for (i = 9; i <= 9; i += 1) {       // Арифметический цикл в языке C
    ТЕЛО_ЦИКЛА;
}
```

Арифметического цикла в Питоне тоже нет, но он имеет средство, которое прекрасно его заменяет: *цикл перебора элементов множества*. Такие циклы базируются на концепции итераторов, о которой мы еще поговорим немного позже. Такой цикл в Питоне выглядит следующим образом:

15 Для обратного отсчета может использоваться ключевое слово `downto`.

```
for i in range(0,10):
    ТЕЛО_ЦИКЛА
```

Немного позже мы к нему еще вернемся и узнаем на что он способен – по гибкости он превосходит обычные арифметические циклы.

§6.10. Управляющие структуры

Настало время оглянуться назад. В первой главе речь шла о том, что высокоуровневые программы строятся на нескольких видах конструкций. Давайте вспомним их и сопоставим каждую из них со знаниями приобретенными в пройденных главах. Итак, в высокоуровневые программы строятся на основе следующих типов операций:

- Ввод данных (пока мы умеем вводить данные только с клавиатуры с помощью функций `input` и `raw_input`);
- Вывод данных (данные мы выводили на экран; вывод производится с помощью команды `print`);
- Выполнение некоторых операций над числами, строками или другими объектами (большинство выражений, в том числе, с использованием функций относятся как раз к этим операциям);
- Выбор ветви выполнения программы на основе принятого решения (условный оператор `if`);
- Повторение группы операций чаще всего с изменением одного или нескольких параметров (операторы цикла `while` и `for`).

Кроме того, в третьей главе мы разобрались с функциями, предоставляющими нам множество преимуществ, в том числе:

- Любой часто повторяемой последовательности операций мы можем назначить имя и по нему вызывать ее на выполнение;
- Использование функций упрощает понимание программы и процесс ее отладки: мы можем отладить каждую функцию по отдельности с предельными значениями параметров, а затем собрать их в единую систему, устойчивую к ошибкам при вводе данных пользователем и другим непредвиденным ситуациям;
- Функции позволяют избежать дублирование кода, поэтому изменение хорошо спроектированной программы требует не так много усилий;
- Хорошо продуманные функции могут использоваться в других проектах, что повышает нашу производительность и позволяет в полной мере насладиться процессом решения текущей задачи;
- И, наконец, с помощью рекурсивных функций можно элегантно решать довольно сложные задачи, и программа при этом будет занимать всего несколько строк.

Итак, первая ступень пройдена: теперь мы владеем вопросами, на основе которых строится подавляющее большинство *приложений* – программ, позволяющих решать повседневные задачи (иногда их называют *прикладными программами*). В следующих главах мы продолжим изучать новые возможности Питона и концепции программирования, но самое трудное уже позади.

Глава 7. Строки

Разобравшись с основами управления потоком выполнения программ, перейдем к более подробному изучению типов данных, ведь для более эффективного управления данными полезно понимать, как они устроены. Начнем с одного из уже знакомых нам по предыдущим главам типов – строк (`str`).

§7.1. Оператор индексирования

Во второй главе мы уже научились выполнять некоторые операции над строками, но до этого момента мы работали со строками как с единым целым. Известные нам операции над строками (конкатенация и итерация) являются, по сути аналогами сложения и умножения чисел, но этот набор неполон – не хватает аналогов операций вычитания и деления.

Если вы делали упражнения, то должны были заметить, что операции конкатенации и итерации не обладают свойством коммутативности. В следствии этого, трудно представить, как применить к строкам операции вычитания и деления в том виде, в котором они применяются к числам.

Тем не менее, в программировании существуют целый набор специальных операций и функций, позволяющих работать с наборами символов, составляющими строки – *подстроками*.

Простейший из них – *оператор индексирования*. Данный оператор позволяет получить любой одиночный символ из строки. У него довольно простой синтаксис:

```
СТРОКА[ИНДЕКС]
```

Индекс, указываемый в квадратных скобках, представляет собой порядковый номер символа в строке:

```
>>> 'Hello!'[1]
'e'
```

Хм, это не то, что мы ожидали увидеть – интерпретатор почему-то вернул не первый, а второй символ строки. Объясняется эта семантическая ошибка очень просто. Дело в том, что компьютер начинает считать не с единицы, как привыкли мы с вами, а с нуля. Проверим:

```
>>> 'Hello!'[0]
'H'
```

Работает! Идем дальше.

§7.2. Длина строки и отрицательные индексы

Для удобной работы с оператором индексирования хорошо бы было знать длину строки. Впрочем, выяснить это довольно просто. Смотрите:

```
>>> len('Hello world!')
12
>>>
```

Обратите внимание, что пробел тоже считается. Теперь давайте попробуем вывести последний символ строки:

```
>>> a = 'Hello!'
>>> a[len(a)]


Traceback (most recent call last):
  File "<pyshell#16>", line 1, in -toplevel-
    a[len(a)]
IndexError: string index out of range
>>>
```

Т.к. нумерация символов в строках начинается с нуля, мы использовали недопустимое значение индекса – символа с таким индексом в этой строке нет, поэтому интерпретатор сгенерировал исключение `IndexError: string index out of range`. Исправим ошибку, воспользовавшись композицией: в качестве индекса может использоваться любое выражение, возвращающее целое число.

```
>>> a = 'Hello!'
>>> a[len(a)-1]
'!'
```

Использование функции `len()` в операторе индексирования выглядит громоздко, поэтому в Питоне предусмотрен более короткий вариант записи:

```
>>> a[-1]
'!'
```

Еще один маленький эксперимент. Попробуем подставить в качестве индекса другое отрицательное число:

```
>>> a[-5]
'e'
```

Таким образом, мы можем индексировать строку с обоих ее концов – это очень удобно. К слову, такая возможность есть в очень немногих языках программирования. А теперь небольшое упражнение для закрепления материала.

Упражнение. *Напишите программу, которая выводит длину введенной пользователем строки, а также первый, пятый и последний символ. Не забудьте предусмотреть случай, когда длина строки составляет меньше пяти символов.*

§7.3. Перебор и цикл for

Научившись обращаться с индексами, поэкспериментируем с циклами. Иногда возникает потребность перебора символов строки и выполнение операции над каждым из них. Для этого как раз и применяются циклы.

```
string = "hello"
index = 0
while index < len(string):
    letter = string[index]
```

```
    print letter
    index = index + 1
```

Данная программа перебирает все символы строки и выводит каждый из них на новой строке. При этом, благодаря строгому неравенству в условии `index < len(string)`, повтор тела цикла завершается на символе с индексом `len(string) - 1`, который является последним в строке.

Упражнение. *Напишите функцию, которая получает в качестве параметра строку и выводит символы, ее составляющие, в обратном порядке.*

Выполняя упражнение, вы, вероятно, заметили, что в циклах с индексами нетрудно запутаться. Поэтому в Питоне есть более удобный способ перебора элементов строк (и не только): цикл `for`. Наша программа будет выглядеть так:

```
for letter in string:
    print letter
```

Данный цикл не завершается до тех пор, пока каждый элемент строки `string` не будет поочередно присвоен переменной `letter`, начиная с первого. Такой подход избавляет программиста от необходимости думать над условием завершения цикла, что позволяет ему сосредоточиться на решаемой задаче.

§7.4. Срезы строк

В языках Си и Паскаль индексы позволяют получать доступ к одиночным символам строки, но в Питоне все гораздо интереснее: можно обращаться не только к одиночным символам, но и к целым подстрокам – *срезам* (по-английски, *slices*). Смотрите:

```
>>> s = "Peter, Paul, and Mary"
>>> print s[0:5]
Peter
>>> print s[7:11]
Paul
>>> print s[17:21]
Mary
```

В общем виде: `string[n:m]`,где `n` указывает индекс начала среза, а `m` – индекс конца среза. При этом начальный и конечный индексы разделяются двоеточием. Обратите также внимание, что символ с индексом `m` в срез не включается.

Существуют также сокращенные формы оператора построения среза. Если не указать начальный или конечный индекс среза то будет подразумеваться начало или конец строки соответственно:

```
>>> print s[:5]
Peter
>>> print s[17:]
Mary
```

Это заметно упрощает читабельность программы, т.к. отпадает необходимость в коде, вычисляющем длину строки, который не относится к сути реализуемого программой алгоритма.

Упражнение. *Чему будет соответствовать срез строки s[:]?*

Еще одна интересная возможность, которая появилась в интерпретаторе Питона, начиная с версии 2.3. Теперь для оператора индексирования можно указать еще и шаг выбора элементов в срез:

```
>>> print s[::2]
Ptr al n ay
```

Что у нас получилось? Срез состоящий из элементов с четными индексами (если учитывать, что нумерация начинается с нуля)!

Упражнение. *Создайте строку длиной 10-15 символов, и извлеките из нее следующие срезы:*

1. *Все символы кроме последних четырех;*
2. *Символы с индексами кратными трем;*
3. *Все символы строки с четными индексами за исключением первых четырех и последних пяти.*

Придумайте 2-3 собственных примера.

§7.5. Сравнение строк

Операторы сравнения работают и со строками. Когда возникает необходимость проверить равны ли две строки, можно выполнить следущее:

```
if word == "banana":
    print  "Yes, we have no bananas!"
```

Другая операция сравнения полезна для упорядочивания слов в алфавитном порядке:

```
if word < "banana":
    print "Your word," + word + ", comes before banana."
elif word > "banana":
    print "Your word," + word + ", comes after banana."
else:
    print "Yes, we have no bananas!"
```

Но следует заметить, что Питон работает со строчными и заглавными символами несколько необычно для нас людей. Заглавная буква считается большей, чем любая строчная. В результате получаем:

```
Your word, Zebra, comes before banana.
```

Для решения этой проблемы можно конвертировать строку к стандартному виду, например, во все строчные символы до выполнения сравнения. Более сложная проблема - заставить программу понять, в чем заключаются отличия между зеброй и бананом.

§7.6. Строки нельзя изменить

Для изменения символа в строке логично было бы использовать оператор индексирования (`[]`) слева от знака присваивания. Например:

```
greeting = "Hello, world!"
greeting[0] = 'J'             # ERROR!
print greeting
```

Но вместо ожидаемого вывода `Jello, world!`, этот код генерирует исключение: `TypeError: object doesn't support item assignment`.

Это означает, что строки в Питоне не могут быть изменены частично – строковый тип эту операцию не предусматривает. Лучшее, что можно сделать – это создать новую строку, которая является измененным оригиналом:

```
greeting = "Hello, world!"
newGreeting = 'J' + greeting[1:]
print newGreeting
```

Данное решение состоит в том, что бы сцепить новый символ со срезом строки `greeting`. Этот оператор не оказывает эффекта на начальную строку.

Упражнение. *Напишите программу, заменяющую 5-й (если начинать считать по-человечески, т.е. с единицы) символ строки `Hello, world!` на восклицательный знак.*

§7.7. Функция find

Рассмотрим такую функцию:

```
def find(str, ch):
    index = 0
    while index < len(str):
        if str[index] == ch:
            return index
        index = index + 1
    return -1
```

Что она делает? В некоторым смысле функция `find()` противоположна оператору индексирования `[]`. Отличие состоит в том, что вместо того, чтобы извлекать из строки символ по его индексу, она возвращает индекс первого вхождения символа в строке. Если символ не найден, то функция возвращает `-1`.

Это первый пример использования инструкции `return` внутри цикла. Если условие `str[index] == ch`, то функция возвращает значение немедленно, прерывая цикл преждевременно.

Если символ не появился в строке, то программа выходит из цикла «в штатном режиме» и возвращает `-1`.

Упражнение. *Измените код функции `find()` так, чтобы требовался третий параметр – индекс в строке, начиная с которого должен производиться поиск. Не*

забудьте про обработку ситуаций с некорректными и предельными значениями этого параметра (например, когда указанный индекс выходит за диапазон допустимых значений).

§7.8. Циклы и счётчики

Следующая программа считает количество символов, встречающихся в строке:

```
fruit = "banana"
count = 0
for char in fruit:
    if char == 'a':
        count = count + 1
print count
```

Программа демонстрирует интересный пример использования циклов. Переменная `count` инициализируется со значением `0` и затем инкрементируется[16] каждый раз когда символ `a` найден. Когда происходит выход из цикла, переменная `count` содержит результат – количество символов `a`. Переменные, в которых сохраняется количество элементов в процессе их перебора, удовлетворяющих некоторому условию, называют *счетчиками*.

Упражнение. *Создайте на основе кода примера функцию `countLetters` и обобщите ее так, чтобы она принимала строку и искомый символ в качестве параметров.*

Упражнение. *Перепишите эту функцию так, чтобы вместо просмотра строки использовать версию функции `find` с тремя параметрами, которую вы написали выполняя упражнение из предыдущего параграфа.*

§7.9. Модуль string

Модуль `string` содержит полезные функции, заметно упрощающие работу со строками. Как обычно, необходимо сначала импортировать модуль до его использования:

```
>>> import string
```

Модуль `string` включает функцию `find()`, которая делает то же самое, что и функция, написанная нами. Это распространенная ситуация в области программирования: прежде чем «изобретать велосипед», иногда стоит поискать готовые решения – всегда есть вероятность, что кто-то уже сталкивался с похожей задачей и решил ее лучше нас. К тому же разбираясь в чужом коде можно многому научиться.

Итак, для вызова функции из модуля, мы должны указать имя модуля и имя функции, используя точку в качестве разделителя:

```
>>> fruit = "banana"
>>> index = string.find(fruit, "a")
```

16 *Инкрементировать* – увеличивать значение чего-либо на единицу; *декрементировать* – уменьшать значение на единицу.

```
>>> print index
1
```

Такой способ вызова помогает избежать конфликтов (и семантических ошибок, которые могут стать их следствием) между именами встроенных или написанных нами функций и именами функций, импортированных из модулей. Используя точку в качестве разделителя, мы можем указать какую версию `find()` хотим использовать.

В подтверждение мысли об «изобретении велосипеда», стоит заметить, что функция `string.find()` является более общим решением, чем наш вариант. Во-первых, может искать подстроки, не только символы:

```
>>> string.find("banana", "na")
2
```

Также имеется возможность задать дополнительный аргумент, который будет определять индекс с которого надо начинать:

```
>>> string.find("banana", "na", 3)
4
```

Или может потребоваться пара дополнительных аргументов, которые определяют диапазон индексов:

```
>>> string.find("bob", "b", 1, 2)
-1
```

В последнем примере, поиск оказался неудачным, потому что символ `b` не входит в диапазон индексов от `1` до `2` (не включая `2`).

Упражнение. *Выведите на экран справку по модулю `string` и ознакомьтесь со справочной информацией следующих функций:*

1. *capitalize()*
2. *capwords()*
3. *count()*
4. *find()*
5. *lower()*
6. *replace()*
7. *upper()*

Поэкспериментируйте с ними.

§7.10. Классификация символов

Иногда возникает потребность проверить регистр символа или выяснить не является ли он цифрой. Модуль `string` предоставляет несколько специальных предопределенных переменных, которые могут быть полезны для этих целей.

Строка `string.lowercase` содержит все строчные символы. Аналогично в переменной `string.uppercase` хранятся все заглавные буквы. Попробуйте следущие примеры и посмотрите, какой результат выдаст интерпретатор:

```
>>> print string.lowercase
>>> print string.uppercase
```

```
>>> print string.digits
```

Мы можем использовать эти переменные и функцию `find()` для классифицирования символов. Например, если `find(lowercase, ch)` возвращает результат, отличный от `-1`, то символ `ch` должен быть строчным.

```
def isLower(ch):
  return string.find(string.lowercase, ch) != -1
```

Так же мы можем использовать оператор `in`, котрый определяет, входит ли символ в строку или нет:

```
def isLower(ch):
  return ch in string.lowercase
```

Альтернативой этому является оператор сравнения следующего вида:

```
def isLower(ch):
  return 'a' <= ch <= 'z'
```

Если `ch` между `a` и `z`, то это строчной символ.

Упражнение. *Какая версия `isLower()` будет самой быстрой? Какие еще причины, помимо скорости, Вы можете привести для предпочтения одного другому?*

Другая переменная, предопределенная в модуле `string`, содержит все пробельные (непечатаемые) символы, включая пробел, табуляцию (`"\t"`) и символы возврата каретки (`"\r"`) и новой строки (`"\n"`) :

```
>>> print string.whitespace
```

Как мы заметили в §6.6, пробельные символы перемещают курсор без печати чего-либо: они создают пустое пространство между видимыми символами.

§7.11. Строки unicode

Глава 8. Списки

Список – это упорядоченное множество значений, идентифицируемых индексом. Во многом списки схожи со строками, которые, по сути, являются упорядоченными множествами символов. Отличие списков и строк заключается в том, что элементы списка могут быть любого типа. Упорядоченные множества называют *последовательностями*.

§8.1. Создание списков

Существует насколько способов создания списков. Самый простой из них: перечислить элементы списка через запятую в квадратных скобках:

```
>>> [10, 20, 30, 40]
[10, 20, 30, 40]
>>> ["one", "two", "three"]
['one', 'two', 'three']
```

Первый пример – список четырех целых чисел, а второй – список трех строк. Элементы списков вовсе не обязательно должны быть одного типа. Следующий список содержит строку, целое и дробное числа и другой список:

```
>>> ["hello", 5, 2.0, [10, 20]]
['hello', 5, 2.0, [10, 20]]
```

Список, являющийся элементом другого списка, называют *вложенным*.

Кроме того, Питон предоставляет возможность быстрого создания списков целых значений, без необходимости их перечислять:

```
>>> range(1,5)
[1, 2, 3, 4]
```

В данном примере функция `range()` принимает два целых аргумента и возвращает список, который содержит все целые числа в промежутке между заданными значениями, включая первое и исключая второе.

Существует еще два способа вызова функции `range()`. Если ей передано только одно значение, то в результате она вернет список с целыми значениями от 0 до N, где N – значение параметра:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Если же `range()` вызвана с тремя аргументами, то последний из них интерпретируется как *размер шага*. Т.е. В результирующем списке значения будут идти не подряд, а через промежутки, равные шагу:

```
>>> range(1, 10, 2)
[1, 3, 5, 7, 9]
```

Список, который не содержит ни одного элемента, называют *пустым*. Он обозначается пустыми квадратными скобками (`[]`).

Ну, и наконец, как и любые другие значения, списки могут сохраняться в переменных:

```
numbers = [17, 123, 537]
empty = []
print numbers, empty
[17, 123, 537] []
```

§8.2. Списки и индексы

Синтаксис обращения к элементам списка точно такой же, как и при обращении к символам строк – используем оператор индексирования ([]).

```
>>> numbers[0]
17
>>> numbers[-1]
537
```

Применительно к спискам оператор индексирования работает точно так же, как и в случае строк. Индексом может быть любое выражение, возвращающее целое число, в том числе отрицательное. Если индекс меньше нуля, то отсчет индекса будет начат с конца списка.

Оператор построения среза списка работает также, как и со строками (см. *§7.4*):

```
>>> list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> list[1:3]
['b', 'c']
>>> list[:4]
['a', 'b', 'c', 'd']
>>> list[3:]
['d', 'e', 'f']
>>> list[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Единственное отличие заключается в том, что элементы списков (в отличие от символов строк) могут меняться:

```
>>> numbers[1] = 5
>>> numbers
[17, 5, 537]
>>> numbers[1:] = [6, 7]
>>> numbers
[17, 6, 7]
>>> numbers[0], numbers[2] = 1, 2
>>> numbers
[1, 6, 2]
```

Упражнение. *Придумайте наглядный способ графического представление списков. Проанализируйте примеры данного параграфа с помощью этой графической нотации и объясните результаты выполнения команд.*

§8.3. Длина списка

Нетрудно догадаться, что для вычисления длины списка мы можем использовать функцию `len()`. Одно из применений этой функции – определение длины списка в циклах, осуществляющих перебор элементов списка.

```
mylist = ['one', 'two', 'three', 'four', 'five']


i = 0
while i < len(mylist):
    print mylist[i]
    i += 1
```

Обратите внимание, что если список содержит в качестве элемента другой список, то этот вложенный список будет считаться как один элемент. Это видно из следующего примера:

```
>>> mylist = [[1, 'one'], [2, 'two'], [3, 'three'], 'four', 5]
>>> len(mylist)
5
```

Упражнение. *Напишите программу, которая считает и выводит количество элементов в каждого подсписка списка `mylist`. Что произойдет, если функции `len()` передать целочисленное значение? Измените программу так, чтобы избежать такой ситуации.*

Напишите программу, которая запрашивает количество элементов списка у пользователя, а затем поочередно предлагает пользователю ввести указанное количество элементов списка (с помощью функции `raw_input()`). По завершении вода программа должна вывести список.

§8.4. Принадлежность списку

Принадлежность элемента последовательности проверяется с помощью логического оператора `in` – с ним мы уже сталкивались в *§7.10*. Этот оператор замечательно работает и со списками:

```
>>> list = [[1, 'one'], [2, 'two'], [3, 'three'], 'four', 5]
>>> 'four' in mylist
True
>>> [1, 'one'] in  mylist
True
>>> 6 in mylist
False
```

Как видите, в качестве первого операнда оператора `in` может выступать и список, и если такой список является элементом второго операнда, что результатом выражения станет «истина» (`True`).

§8.5. Списки и цикл for

В §7.3 мы уже разобрались, как работает цикл `for`. Со списками он работает также, как и со строками. Обобщенный синтаксис конструкции такой:

```
for ПАРАМЕТР_ЦИКЛА in ПОСЛЕДОВАТЕЛЬНОСТЬ:
    ТЕЛО_ЦИКЛА
```

Данное выражение эквивалентно следующей конструкции:

```
i = 0
while i < len(ПОСЛЕДОВАТЕЛЬНОСТЬ):
    ПАРАМЕТР_ЦИКЛА = ПОСЛЕДОВАТЕЛЬНОСТЬ[i]
    ТЕЛО_ЦИКЛА
    i += 1
```

Как видите, такая запись более компактна и легко читается. Например:

```
for fruit in fruits:
    print fruit
```

Знающие английский читатели легко прочитают это выражение так: «For (every) fruit in (the list of) fruits, print (the name of the) fruit» – «Для каждого фрукта из (списка) фруктов, вывести (название) фрукта».

К тому же, цикл `for` избавляет нас от необходимости вводить дополнительную переменную-счетчик.

С помощью цикла `for` мы можем последовательно перебирать элементы любой последовательности (например, строки или списка[17]), причем по принципу композиции допускается использование любых выражений, возвращающих последовательность:

```
for number in range(20):
    if number % 2 == 0
    print  number
```

или

```
for fruit in ["banana", "apple", "quince"]:
    print "I like to eat " + fruit + "s!"
```

Первый пример выводи одинадцать четных чисел (включая ноль) от нуля до 20, а второй – выражает энтузиазм автора программы относительно употребления в пищу перечисленных фруктов.

17 Впоследствии мы увидим, что это замечание справедливо и для других типов данных: словарей, кортежей и других типов, для которых определены правила перебора элементов.

§8.6. Операции над списками

Вы, наверняка, заметили, что списки очень похожи на строки. В данном разделе мы обнаружим еще несколько сходств.

Итак, оператор «+» производит сцепление (конкатенацию) списков:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
```

Пример, пожалуй, понятен и без пояснений. По аналогии со строками оператор «*» повторит список из первого аргумента указанное количество раз:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Упражнение. *Сравните математические свойства операций над списками со свойствами их аналогов для строк (см. упражнение из §2.9).*

§8.7. Изменение списков

В отличие от строк, элементы список могут изменяться. Мы можем изменять один из элементов списка, используя оператор индексирования в левом операнде оператора присваивания:

```
>>> fruit = ["banana", "apple", "quince"]
>>> fruit[0] = "pear"
>>> fruit[-1] = "orange"
>>> print fruit
['pear', 'apple', 'orange']
```

Оператор построения среза тоже можно использовать аналогичным образом:

```
>>> a_list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> a_list[1:3] = ['x', 'y']
>>> print a_list
['a', 'x', 'y', 'd', 'e', 'f']
```

Кроме того, можно удалять элементы списка, присваивая срезу в качестве значения пустой список:

```
>>> a_list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> a_list[1:3] = []
>>> print a_list
```

```
['a', 'd', 'e', 'f']
```

Добавлять элементы можно точно таким же образом, причем длина вставляемого списка может отличаться от длины среза, которому он присваивается в качестве значения:

```
>>> a_list = ['a', 'd', 'f']
>>> a_list[1:1] = ['b', 'c']
>>> print a_list
['a', 'b', 'c', 'd', 'f']
>>> a_list[4:4] = ['e']
>>> print a_list
['a', 'b', 'c', 'd', 'e', 'f']
```

Упражнение. *Проверьте, что произойдет, если длина среза будет больше 1, и если второй операнд оператора построения среза будет превосходить длину исходного списка. Можно ли сцепить два списка с помощью оператора построения среза? Если да, то как?*

§8.8. Удаление элементов списка

Конечно, удалять элементы из списков с помощью срезов не очень-то удобно, да и ошибиться несложно. Но в Питоне предусмотрен альтернативный вариант, который более читабелен: оператор `del`. С ним все выглядит гораздо проще:

```
>>> a = ['one', 'two', 'three']
>>> del a[1]
>>> a
['one', 'three']
```

Оператор `del` поддерживает отрицательные индексы и генерирует исключение, если переданный индекс оказывается за пределами допустимого диапазона.

Кроме того, в качестве параметра оператор может принимать срезы:

```
>>> a_list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del a_list[1:5]
>>> print a_list
['a', 'f']
```

Как обычно, в данном случае в срез попадают все элементы от первого индекса до предпоследнего включительно.

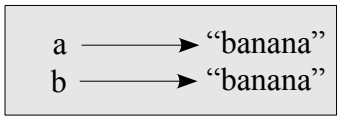
§8.9. Объекты и значения

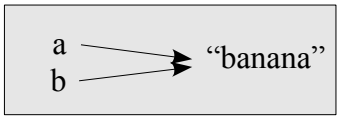
Теперь разберемся с тем, что такое объекты, и чем они отличаются от значений. Для начала попробуем выполнить следующие выражения:

```
>>> a = "banana"
>>> b = "banana"
```

Мы знаем, что `a` и `b` будут указывать на одну и ту же строку `"banana"`, но пока мы не можем сказать, разные ли это строки или это одна и та же строка, на которую указывают две переменные.

Таким образом, пока возможны два варианта:





В первом случае переменные `a` и `b` ссылаются на две разные сущности, содержащие одинаковые последовательности символов. Во втором – эти переменные ссылаются на одну и ту же строку, т.е. на один и тот же *объект*.

Пока не будем давать формальное определение понятию объекта – мы вернемся к нему, когда будем изучать объектно-ориентированное программирование в последующих главах.

Каждый объект имеет уникальный идентификатор, который можно получить с помощью функции `id()`. Так что вызвав эту функцию с параметрами `a` и `b`, мы можем определить ссылаются ли они на один и тот же объект или нет.

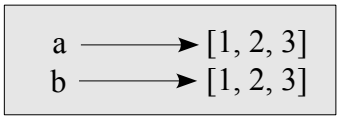
```
>>> id(a)
135044008
>>> id(b)
135044008
```

В данном случае вы получили один и тот же идентификатор дважды. Это означает, что интерпретатор Питона создал в памяти только один строковый объект, и на него ссылаются обе переменные.

Интересно, что в случае со списками интерпретатор ведет себя по-другому. При присваивании двух списков с одинаковым набором элементов мы получим два разных объекта:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> id(a)
135045528
>>> id(b)
135041704
```

Таким образом диаграмма состояния будет выглядеть так:



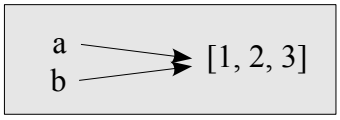
Переменные `a` и `b` ссылаются на одинаковые значения, но на разные объекты. Это связано с тем, что списки, в отличие от строк можно менять, т.е. если бы обе переменные ссылались на один объект, то при изменении значения a менялось бы и значение b, что могло бы внести серьезную неразбериху. Такое поведение нормально, просто интерпретатор Питона, зная, что строки изменять нельзя, экономит таким образом память.

§8.10. Ссылки на объекты

В предыдущем разделе мы выяснили, как переменные ссылаются на различные объекты. Теперь рассмотрим другой случай. Если присвоить переменную, ссылающейся на объект, другой переменной, то мы получим переменные, ссылающиеся на один объект. Т.е. копирования объекта в данном случае не произойдет:

```
>>> a = [1, 2, 3]
>>> b = a
```

Диаграмма состояния после такого присваивания будет выглядеть так:



Т.к. две переменные ссылаются на один и тот же список, иногда говорят, что они являются *псевдонимами* (по-английски, *aliases*). Таким образом изменение значения первой переменной автоматически затронет значение, на которое ссылается вторая:

```
>>> b[0] = 5
>>> print a
[5, 2, 3]
```

Несмотря на то, что такое поведение иногда может быть полезным, оно может привести к семантическим ошибкам, найти которые крайне трудно. Это значит, что лучше стараться избегать клонирования при работе изменяемыми объектами. Разумеется, с неизменяемыми объектами таких проблем не возникает, поэтому Питон создает псевдонимы на одинаковые строки, благодаря чему достигается экономия памяти.

§8.11. Копирование списков

При необходимости изменить список и при этом сохранить его исходный вариант нам не обойтись без возможности скопировать список, а не просто получить ссылку на него. Процесс копирования в данном контексте иногда называют *клонированием*, чтобы избежать двусмысленностей и путаницы.

Самый простой способ копирования – использовать оператор построения среза:

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> print b
[1, 2, 3]
```

При формировании среза списка a в памяти компьютера создается новый список, в который записываются значения элементов среза. В данном случае срез включает в себя весь исходный список.

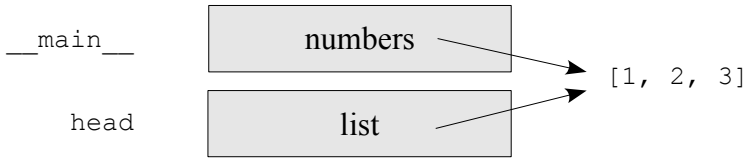
Теперь мы можем изменять список `b`, никак не влияя на список `a`:

```
>>> b[0] = 5
>>> print a
[1, 2, 3]
```

§8.12. Списки-параметры

Списки, как и обычные переменные тоже могут быть аргументами функций. При этом в тело функции передается ссылка на список, а не его копия. Например, функция `head()` принимает в качестве параметра список, на который ссылается переменная `list`:

```
>>> def head(list):
... return list[0]
>>> numbers = [1, 2, 3]
>>> head(numbers)
1
```



Параметр `list` и переменная `numbers` ссылаются на один и тот же объект. На диаграмме состояний это можно отобразить так:

Например,функция `deleteHead()` удаляет из переданного в качестве параметра списка первый элемент:

```
>>> def deleteHead(list):
...     del list[0]
>>> numbers = [1, 2, 3]
>>> deleteHead(numbers)
>>> print numbers [2, 3]
```

Возврат списка из функции так же производится по ссылке. Например, функция tail() возвращает список содержащий все элементы переданного в качестве параметра списка, кроме первого:

```
>>> def tail(list):
...     return list[1:]
>>> numbers = [1, 2, 3]
>>> rest = tail(numbers)
>>> print rest [2, 3]
```

Функция `tail()` возвращает новый список, созданный с помощью оператора построения среза. Значение в переменной `numbers` при этом никак не изменилось.

§8.13. Вложенные списки

Вложенными называются списки, которые являются элементами другого списка. В приведенном ниже примере третий элемент списка `l` является сложенным списком:

```
>>> l = ["hello", 2.0, 5, [10, 20]]
```

Если вывести `list[3]`, то мы получим `[10, 20]`. Чтобы получить элемент сложенного списка, нам придется сделать два шага:

```
>>> elt = l[3]
>>> elt[0]
10
```

Другой более короткий вариант – использовать композицию:

```
>>> l[3][1]
20
```

Операторы индексирования выполняются слева на право. Поэтому приведенный пример сначала получает третий элемент из исходного списка `l`, а затем – первый элемент полученного вложенного списка.

§8.14. Матрицы

TODO: Можно написать немного о матрицах (где и как используются).

Вложенные списки обычно используются для представления матриц. Например, матрицу:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

можно представить так:

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

В переменной matrix хранится список с тремя элементами, в каждом из которых хранится строка матрицы тоже в виде списков. Извлечь строку матрицы мы можем обычным образом – с помощью оператора индексирования:

```
>>> matrix[1]
[4, 5, 6]
```

В свою очередь для доступа к конкретному элементу матрицы мы можем использовать композицию двух операторов индексирования: первый индекс будет определять номер строки, а второй – номер элемента (т.е. номер столбца) в ней.

```
>>> matrix[1][1]
5
```

Данный способ представления матриц является стандартным, но не единственным. Возможен вариант использования первого индекса для нумерации столбцов, а второго – для нумерации строк. Позже в *10-й главе* мы познакомимся с принципиально другим способом представления матриц с помощью словарей.

§8.15. Списки и строки

Так как строки являются неизменяемыми объектами, иногда удобно их конвертировать в списки. Для этого в Питоне есть функция `list()`, которая в качестве параметра принимает

любой тип последовательностей (например, строку как последовательность символов) и создает из элементов этой последовательности список:

```
>>> list("Crunchy Frog")
['C', 'r', 'u', 'n', 'c', 'h', 'y', ' ', 'F', 'r', 'o', 'g']
```

Аналогично функция `str()` принимает в качестве параметра любое значение и возвращает строковое представление этого значения.

```
>>> str(5)
'5'
>>> str(None)
'None'
>>> str(list("nope"))
"['n', 'o', 'p', 'e']"
```

Как видно из последнего примера, функция `str()` может использоваться для объединения списка символов в одну строку. Для этой же цели может использоваться функция `join()` из модуля `string`:

```
>>> import string
>>> char_list = list("Frog")
>>> char_list
['F', 'r', 'o', 'g']
>>> string.join(char_list, '')
'Frog'
```

Еще одна крайне полезная функция `split()` из модуля `string` позволяет получать списки строк. Данная функция разбивает строку на несколько слов. По умолчанию, любое число или пробельный символ интерпретируется как разделитель между словами:

```
>>> import string
>>> song = "The rain in Spain..."
>>> string.split(song)
['The', 'rain', 'in', 'Spain...']
```

Необязательный аргумент функции может использоваться для определения *разделителя* между словами. В следующем примере в качестве разделителя используется строка `"ai"`:

```
>>> string.split(song, 'ai')
['The r', 'n in Sp', 'n...']
```

Обратите внимание, что сам разделитель не фигурирует в строках, из которых составляется результирующий список.

Функция `join()`, по сути, осуществляет преобразование обратное функции `split()`.Она принимает список строк и соединяет его элементы с помощью пробелов:

```
>>> l = ['The', 'rain', 'in', 'Spain...']
>>> string.join(l)
'The rain in Spain...'
```

Подобно функции `split()`, `join()` тоже имеет необязательный аргумент, который задает разделитель между словами. По умолчанию, `join()` в качестве разделителя использует пробел.

```
>>> string.join(list, '_')
'The_rain_in_Spain...'
```

Упражнение. *Разбейте строку из переменной `song` на слова, а затем сцепите их в обратном порядке так, чтобы при выводе на экран слова строки выводились по одному на каждой строке.*
Решений у данной задачи несколько. Самое короткое может быть реализовано с помощью встроенных операций над списками. Поэтому рекомендуем почитать справку: `help(list)`

Глава 9. Кортежи

§9.1. Понятие кортежа

Итак, в предыдущих главах мы уже ознакомились с двумя типами данных, являющихся последовательностями: строками, состоящими из символов, и списками, способными хранить в себе элементы любого типа. Одним из их отличий, о которых мы уже упоминали, является то, что элементы списка менять можно, а символы строки – нет. Другими словами, строки *неизменяемы*, а списки *изменяемы*.

В Питоне есть еще один тип данных, называемый *кортеж*, который полностью аналогичен списку за исключением того, что его элементы изменять не допускается. Синтаксис создания кортежа – перечисление значений через запятую:

```
>>> tuple = 'a', 'b', 'c', 'd', 'e'
```

Принято, но не обязательно, заключать список элементов кортежа в скобки:

```
>>> tuple = ('a', 'b', 'c', 'd', 'e')
```

Для создания одноэлементного кортежа необходимо поставить в конце запятую:

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

Без запятой Питон интерпретирует `('a')` как строку в скобках:

```
>>> t2 = ('a')
>>> type(t2)
<type 'string'>
```

Абстрагируясь от синтаксиса, операции над кортежами такие же, как и над строками. Оператор индексирования выбирает элемент из кортежа.

```
>>> tuple = ('a', 'b', 'c', 'd', 'e')
>>> tuple[0]
'a'
```

Оператор построения среза выбирает диапазон элементов.

```
>>> tuple[1:3]
('b', 'c')
```

Но если мы попробуем изменить значение одного из элементов, мы получим ошибку:

```
>>> tuple[0] = 'A'
TypeError: object doesn't support item assignment
```

Хотя мы не можем менять элементы кортежа, мы можем заменить его другим кортежем:

```
>>> tuple = ('A',) + tuple[1:]
>>> tuple
('A', 'b', 'c', 'd', 'e')
```

§9.2. Применение кортежи

При реализации многих алгоритмов возникает необходимость поменять местами значения двух переменных. С классическими операторами присвоения нам требуется введение дополнительной временной переменной. Например, чтобы поменять местами a и b:

```
>>> temp = a
>>> a = b
>>> b = temp
```

Если это приходится делать часто, то код получается громоздким, что плохо сказывается на его понятности – становится сложнее понять суть выполняемых действий. В Питоне же имеется форма задания кортежа, которая лаконично решает эту задачу:

```
>>> a, b = b, a
```

Слева кортеж переменных, справа кортеж значений. Каждое значение соответствует своей переменной. Все выражения справа вычисляются до присвоения. Такая возможность дает гибкость в задании кортежей.

Обратите внимание, что количество переменных слева и значений справа должно быть одинаковым:

```
>>> a, b, c, d = 1, 2, 3
ValueError: unpack tuple of wrong size
```

§9.3. Кортежи и возвращаемые значения

Кстати, в §5.4 мы уже, сами того не заметив, применяли котрежи, когда писали функцию вычисления корней квадратного уравнения.

```
>>> def PrintRoots(a, b, c):
...     D = b**2 - 4 * a * c
...     if D < 0:
...         return None, None
...     import math
...     x1 = (-b + math.sqrt(D)) / 2 * a
...     x2 = (-b - math.sqrt(D)) / 2 * a
...     return x1, x2
>>>
>>> print PrintRoots(3, 2, 1)
(None, None)
```

Данная функция возвращает ни что иное как кортеж, состоящий из пары значений! Любая в Питоне функция может возвращать кортеж в качестве результата.

Чтобы было понятнее, попробуем написать функцию, которая меняет местами значения двух переменных:

```
>>> a, b = swap(a, b)
```

Конечно, создание функции `swap()` не дает нам никакого преимущества, к тому же, здесь есть «подводные камни»[18], ради которых и был затеян данный эксперимент:

```
>>> def swap(x, y):
...     x, y = y, x
...
>>>
```

Такая функция, работать не будет – она выполняется, не выдавая никакого сообщения об ошибке, но она не выполняет то, что мы от нее хотим. Это пример семантической ошибки.

Упражнение. *Нарисуйте стековую для этой функции и объясните, почему она не работает.*

§9.4. Случайные числа

Большинство программ делают одно и то же при каждом выполнении, поэтому говорят, что такие программы *определенные*. Определенность хорошая вещь до тех пор, пока мы считаем, что одни и те же вычисления должны давать один и тот же результат. Тем не менее, в некоторых программах от компьютера требуется непредсказуемость. Типичным примером являются игры, но есть масса других применений: в частности, моделирование физических процессов или статистические эксперименты.

Заставить программу быть действительно непредсказуемой задача не такая простая, но есть способы заставить ее казаться непредсказуемой. Одним из таких способов является генерирование случайных чисел и использование их в программе.

В Питоне есть встроенная модуль, который позволяет генерировать *псевдослучаные* числа. Они не истинно случайны, с математической точки зрения, но для наших целей вполне подойдут.

Модуль `random` включает в себя функцию `random`, которая возвращает действительное число в диапазоне от `0.0` до `1.0`. Каждый раз при вызове функции `random` вы получите число из длинного ряда. Чтобы посмотреть, как она работает, запустим следующую программу:

```
>>> import random
>>> for i in range(10):
>>>     x = random.random()
>>>     print x
```

Чтобы получить случайное число между `0.0` и верхней границей `high`, просто умножьте `x` на `high`.

18 Программисты чаще используют в таких случаях термин «грабли».

Упражнение. *Получите случайное число между* `low` *(нижняя граница) и* `high` *(верхняя граница). Затем получите случайное целое число между* `low` *(нижняя граница) и* `high` *(верхняя граница), включая граничные значения.*

§9.5. Список случайных величин

Первым шагом является создание списка случайных величин. randomList принимает целое число в качестве параметра и возвращает список случайных чисел заданной длины. Она начинает выполнятся со списком из n нулей. При каждом проходе через цикл она заменяет один из элементов случайным числом. Возвращаемое значение является ссылкой на полный список:

```
def randomList(n):
    s = [0] * n
    for i in range(n):
        s[i] = random.random()
    return s
```

Проверим эту функцию на списке из восьми элементов. Из соображений отладки удобно начать с малого.

```
>>> randomList(8)
0.15156642489
0.498048560109
0.810894847068
0.360371157682
0.275119183077
0.328578797631
0.759199803101
0.800367163582
```

Числа, выдаваемые функцией `random`, распределены равномерно; это значит, что все значения равновероятны.

Если разбить диапазон возможных значений на одинаковые по размеру «ячейки» и посчитать, сколько раз случайное число попадет в каждую из ячеек, то мы увидим, что в каждой ячейке это количество почти одно и то же.

Эту теорию можно проверить, написав программу, которая будет разделять диапазон на ячейки и считать количество значений в каждой из них.

§9.6. Паттерны программирования

Итак, перейдем к анализу выборки случайных величин. Хорошим решением задач вроде этой является ее разбиение на подзадачи.

В нашем случае мы просматриваем список значений и считаем количество значений, попадающих в данный диапазон. Звучит знакомо, не так ли?

В параграфе §7.8 мы написали программу, которая просматривает строку и считает, сколько раз в строке присутствует заданная буква. Поэтому, мы можем начать с копирования старой программы и адаптации ее под текущую задачу. Исходный код программы был такой:

```
count = 0
for char in fruit:
    if char == 'a':
        count += 1
print count
```

Первым шагом будет замена `fruit` на `list` и `char` на `num`. Это не изменит программу, просто приблизит ее к контексту нашей задачи.

Следующим шагом будет изменение критерия проверки. Нас не интересуют буквы. Нам надо видеть, попал ли `num` в заданный диапазон между `low` и `high`.

```
count = 0
for num in list:
    if low < num < high:
        count += 1
print count
```

И наконец, оформим этот код в виде функции под названием `inBucket()`. Параметрами функции являются список значений и границы диапазона `low` и `high`.

```
def inBucket(list, low, high):
    count = 0
    for num in list:
        if low < num < high:
            count += 1
    return count
```

Как видите, копируя и модифицируя существующую программу можно быстро реализовать нужную функциональность и не тратить много времени на разработку алгоритма. Такие заготовки называют *паттернами* (от англ. *pattern* – пример, образец). Если вам придется решать задачу, которую вы уже решали, просто используйте ранее полученное решение.

§9.7. Анализ выборки

Теперь применим полученную функцию для анализа количества случайных величин попадающий в каждую из «ячеек» диапазона. По мере роста количества «ячеек» использование `inBucket()` становится немного громоздким. С двумя ячейками еще неплохо:

```
bucket1 = inBucket(a, 0.0, 0.5)
bucket2 = inBucket(a, 0.5, 1)
```

Но с четырьмя запись становится неуклюжей:

```
bucket1 = inBucket(a, 0.0, 0.25)
bucket2 = inBucket(a, 0.25, 0.5)
bucket3 = inBucket(a, 0.5, 0.75)
bucket4 = inBucket(a, 0.75, 1.0)
```

Встают две задачи. Первая заключается в том, что нам надо создавать новые имена переменных для каждого нового результата, при этом хотелось бы иметь возможность подбирать количество «ячеек» выборок различного размера. Вторая заключается в том, что приходится вычислять диапазон для каждой ячейки.

Сначала решим вторую проблему. Если количество ячеек `numBuckets`, то ширина каждой ячейки равна `1.0/numBuckets`.

Воспользуемся циклом для вычисления диапазона каждой ячейки. Переменная цикла i меняется от `0` до `numBucket - 1`:

```
bucketWidth = 1.0/numBuckets


high = low
for i in range(numBuckets):
    print i
    low = high
    high += bucketWidth
    print low, "to", high
```

Для вычисления нижней границы каждой ячейки умножим переменную цикла на ширину ячейки. Верхняя граница находится на расстоянии `bucketWidth`.

С `numBuckets = 8` вывод следующий:

```
0.0 to 0.125
0.125 to 0.25
0.25 to 0.375
0.375 to 0.5
0.5 to 0.625
0.625 to 0.75
0.75 to 0.875
0.875 to 1.0
```

Легко убедиться, что каждая ячейка имеет одинаковую ширину, что они не перекрывают друг друга и покрывают весь диапазон от `0.0` до `1.0`.

Теперь вернемся к первой задаче. Требуется способ хранения восьми целых чисел с использованием переменной цикла для выбора только одного числа. Наверняка вам уже пришла в голову идея: «Список!»

Необходимо создавать список ячеек вне цикла, поскольку сделать это нужно только один раз. Внутри цикла мы будем вызывать функцию `inBucket()` и обновлять значение `i`-го элемента:

```
numBuckets = 8
buckets = [0] * numBuckets      # Создание списка с нужным
                                # количеством элементов
bucketWidth = 1.0 / numBuckets

for i in range(numBuckets):
    low = high
    high += bucketWidth
    buckets[i] = inBucket(list, low, high)
print buckets
```

Со списком из `1000` значений эта программа будет выдавать примерно такой список:

```
[138, 124, 128, 118, 130, 117, 114, 131]
```

Все эти значения близки к `1000 / 8 = 125`. таким образом можно сделать вывод, что все значения из используемого диапазона более или менее равновероятны.

Упражнение. *Модифицируйте программу так, чтобы она сохраняла в списке пары значений (кортежи с двумя элементами): количество элементов, попавших в диапазон, и отклонение суммы от ожидаемой (`len(list) / numBuckets`).*
Попробуйте изменить размер списка и количество ячеек. Как изменились отклонения? Как вы думаете, почему?

§9.8. Более эффективное решение

Хотя эта программа работает, она не такая эффективная, какой могла бы быть. При каждом вызове `inBucket()` она просматривает весь список. По мере роста количества ячеек и размера списка, возрастает количество проходов, а, следовательно, и затраты на вычисления.

Попробуем найти другое решение, для чего нам нужно устранить главное «тонкое место» в нашем алгоритме: повторяющие проходы по элементам списка для каждой «ячейки». Как избавиться от функции `inBucket()`?

Поскольку `bucketWidth = 1.0 / numBuckets`, деление на `bucketWidth` равнозначно умножению на `numBuckets`. Если умножить число из диапазона от `0.0` до `1.0` на `numBuckets`, то получится число из диапазона от `0.0` до `numBuckets`. Если это число округлить в меньшую сторону, то получится как раз искомая величина – номер ячейки:

```
numBuckets = 8
buckets = [0] * numBuckets
for i in list:
    index = int(i * numBuckets)
    buckets[index] = buckets[index] + 1
```

Для преобразования действительного числа к целому мы применили функцию `int()`,которая отбрасывает дробную часть.

Упражнение. *Проверьте работоспособность этой программ. Может ли она выдать индекс за пределами диапазона (отрицательный или больше, чем* `len(buckets)-1`*)? Объясните свой ответ.*

Списки, содержащие счетчики количества значений отвечающих некоторым условиям (например, попадание в заданный диапазон), называют *гистограммами*.

Упражнение. *Напишите функцию* `histogram()`*, которая принимает в качестве параметров список и количество ячеек и выдает гистограмму с заданным количеством ячеек.*

Глава 10. Словари

Строки, списки и кортежи используют в качестве индексов целые числа. Если же мы попытаемся использовать использовать в качестве индексов значения изменяемых типов данных, то интерпретатор выведет ошибку.

Словари схожи со списками за исключением того, что в них могут использоваться в качестве индекса значение любого неизменяемого типа (например, `str`, `float`, `tuple`).

§10.1. Создание словаря

Для примера мы создадим англо-испанский словарь, индексами в котором будут служить строковые значения.

Один из способов создать словарь – начать с пустого словаря и добавлять элементы. Пустой словарь обозначается фигурными скобками `{}`:

```
>>> eng2sp = {}
>>> eng2sp['one'] = 'uno'
>>> eng2sp['two'] = 'dos'
```

Первый оператор присваивания создает словарь названный `eng2sp`; остальные операторы добавляют новые элементы в словарь. Мы можем распечатать текущее значение словаря обычным способом:

```
>>> print eng2sp
{'one': 'uno', 'two': 'dos'}
```

Элементы словаря выводятся через запятую; каждый элемент содержит индекс и значение, разделенные двоеточием. В словаре индексы называют *ключами*, а элементы называют *парами ключ-значение*.

Другой способ создать словарь – предоставить список пар ключ-значение, используя тот же синтаксис что и в предыдущем выводе.

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

Если мы снова распечатаем значение `eng2sp`, мы получим небольшой сюрприз:

```
>>> print eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

Пары ключ-значение расположены в другом порядке! К счастью нет повода беспокоиться о расположении, так как элементы словаря никогда не индексируются целыми индексами. Вместо этого для поиска соответствующего значения мы используем ключ:

```
>>> print eng2sp['two']
'dos'
```

Ключ `'two'` выдал значение `'dos'` хотя оно появляется в третьей паре ключ-значение.

§10.2. Операции над словарями

Оператор `del` удаляет из словаря пару ключ-значение. Например, следующий словарь содержит названия различных фруктов и количество каждого фрукта на складе:

```
>>> inventory = {'apples': 430, 'bananas': 312, 'oranges': 525,
'pears': 217}
>>> print inventory
{'oranges': 525, 'apples': 430, 'pears': 217, 'bananas': 312}
```

Если кто-нибудь купил все груши, мы можем удалить этот элемент из словаря:

```
>>> del inventory['pears']
>>> print inventory
{'oranges': 525, 'apples': 430, 'bananas': 312}
```

Или если скоро мы ожидаем поступления новых груш, мы можем просто изменить значение связанное с грушами:

```
>>> inventory['pears'] = 0
>>> print inventory
{'oranges': 525, 'apples': 430, 'pears': 0, 'bananas': 312}
```

Функция `len` также работает со словарями, она возвращает число пар ключ-значение:

```
>>> len(inventory)
4
```

§10.3. Методы словарей

[NOTE: возможно, стоит ввести понятие метода (на простом уровне) немного пораньше]

Метод – это обычная функция, принимающая аргументы и возвращающая значения, но отличающаяся синтаксисом вызова. Подробнее с методами мы будем разбираться в главах, посвященных объектно-ориентированному программированию, где станет понятны более глубокие различия между обычными функциями и методами.

Итак, метод `key` получает словарь и возвращает список ключей которые находит, но вместо синтаксиса функции `keys(eng2sp)`, мы используем синтаксис метода `eng2sp.keys()`.

```
>>> eng2sp.keys()
['one', 'three', 'two']
```

Такая форма записи с точкой задает имя функции, `keys`, а так же имя объекта, к которому необходимо применить функцию, `eng2sp`. Пустые круглые скобки показывают, что этот метод не принимает параметров.

Вызов метода называют invocation; в этом случае, мы могли сказать, что мы вызываем `keys()` объекта `eng2sp`.

Похожий метод `values()` – он возвращает список значений в словаре:

```
>>> eng2sp.values()
['uno', 'tres', 'dos']
```

Метод `items` возвращает список кортежей ключ-значение:

```
>>> eng2sp.items()
[('one','uno'), ('three', 'tres'), ('two', 'dos')]
```

Если метод принимает аргументы, он использует тот же синтаксис что и вызов функции. Например, метод `has_key()` принимает ключ и возвращает истину (`1`) если такой ключ присутствует в словаре.

```
>>> eng2sp.has_key('one')
1
>>> eng2sp.has_key('deux')
0
```

Если вы пытаетесь вызвать метод без указания объекта, вы получаете ошибку. В этом случае, сообщение об ошибке не очень информативное:

```
>>> has_key('one')
NameError: has_key
```

§10.4. Использование псевдонимов и копирование

Так как словари могут быть изменены, необходимо знать о псевдонимах. Всякий раз, когда две переменные ссылаются на один и тот же объект, изменения одной воздействуют на другую.

[TODO: стековая диаграмма]

Если вы хотите изменить словарь и сохраняете копию оригинала, используйте метод `copy()`. Например, `opposites` – словарь, который содержит пары антонимов:

```
>>> opposites = {'up': 'down', 'right': 'wrong', 'true': 'false'}
>>> alias = opposites
>>> copy = opposites.copy()
```

Переменные `alias` и `opposites` ссылаются на один и тот же объект; переменная `copy()` ссылается на только что созданную копию того же словаря. Если мы изменим псевдонимы, `opposites` так же изменится.

```
>>> alias['right'] = 'left'
>>> opposites['right']
'left'
```

Если мы изменим `copy`, то словарь, `opposites` останется прежним.

```
>>> copy['right'] = 'privilege'
>>> opposites['right']
'left'
```

§10.5. Разряженные матрицы

В разделе *8.14* для представления матрицы мы использовали список списков. Это хороший выбор для матрицы содержащей в основном не нулевые значения, но рассмотрим разряженную матрицу подобную этой:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

Представление в виде списка содержит множество нулей:

```
>>> matrix1 = [ [0,0,0,1,0],
                [0,0,0,0,0],
                [0,2,0,0,0],
                [0,0,0,0,0],
                [0,0,0,3,0] ]
```

Но есть и другой способ хранения матриц – мы можем использовать словарь: ключами будут кортежи, хранящие номер строки и номер столбца. Вот представление той же матрицы в виде словаря:

```
>>> matrix2 = {(0,3): 1, (2, 1): 2, (4, 3): 3}
```

Нам необходимо только три пары ключ-значение, одной для каждого ненулевого элемента матрицы. Каждый ключ – кортеж и каждое значение целое число.

Для доступа к элементу матрицы мы могли использовать оператор индексирования:

```
>>> matrix1[0][3]
1
>>> matrix2[0,3]
1
```

Обратите внимание синтаксис для представления в виде словаря не такой как для представления в виде вложенных списков. Вместо двух целочисленных индексов мы используем один индекс – кортеж целых чисел.

Но в таком подходе есть одна сложность. Если мы укажем на нулевой элемент, мы получим ошибку, так как в словаре нет элемента с таким ключом:

```
>>> matrix2[1,3]
KeyError: (1, 3)
```

Данную проблему решает метод `get()`:

```
>>> matrix.get((0,3), 0)
1
```

Первый аргумент – ключ, второй аргумент – значение, которое `get()` должен возвратить, если такого ключа в словаре нет:

```
>>> matrix.get((1,3), 0)
0
```

get definitely improves the semantics of accessing a sparse matrix. Shame about the syntax.

Метод get определенно улучшает семантику доступа к разряженной матрице.

§10.6. Подсказки

Если вы забавлялись с рекурсивной функцией поиска чисел Фибоначчи из раздела *5.12*, то, наверное, заметили, что чем большие аргументы вы передаете, тем дольше функция выполняется. Более того, время расчетов увеличивается очень быстро. На одной нашей машине, `fibonacci(20)` завершается мгновенно, `fibonacci(30)` думает около секунды и `fibonacci(40)` будет работать почти бесконечно.

Чтобы понять почему так происходит, рассмотрим диаграмму вызовов для функции `fibonacci()` с `n = 4`:

[TODO: нарисовать в изображение http://www.ibiblio.org/obp/thinkCSpy/illustrations/fibonacci.png]

Диаграмма вызовов функций отображает совокупность прямоугольных блоков, обозначающих функции, с линиями, соединяющими каждый блок с блоками функций которые он вызывает. Вверху диаграммы, `fibonacci()` с `n = 4` вызывает `fibonacci()` с `n = 3` и `n = 2`. В свою очередь, `fibonacci()` с `n = 3` вызывает `fibonacci()` с `n = 2` и `n = 1`. И так далее.

Таким образом, каждая ветка дерева, по сути, отображает изменение стека вызовов, т.е. Поток выполнения в процессе расчетов, должен пройти по каждой ветке этого дерева. Подсчитайте сколько раз были вызваны `fibonacci(0)` и `fibonacci(1)`. Не самое эффективное решение задачи, с точки зрения производительности (хотя оно, бесспорно, красивое).

Хорошее решение – сохранять значения, которые были недавно вычислены, запоминая их в словаре. Предыдущее вычисленное значение, которое запоминается для последующего использования, называют подсказкой. Ниже приведена реализация `fibonacci()`, использующая подсказки:

```
previous = {0:1, 1:1}

def fibonacci(n):
    if previous.has_key(n):
        return previous[n]
    else:
        newValue = fibonacci(n-1) + fibonacci(n-2)
        previous[n] = newValue
        return newValue
```

Словарь названный `previous` хранит числа Фибоначчи, которые мы уже знаем. Мы начинаем только с двух пар: `0` соответствует `1` и `1` соответствует `1`.

Всякий раз, когда вызывается функция `fibonacci()`, она проверяет словарь, чтобы определить содержит ли он результат. Если это так, функция может немедленно возвратить результат, без выполнения дополнительных рекурсивных вызовов. Если нет, он должна рассчитать новое значение. Новое значение добавляется в словарь, перед тем как функция вернёт результат.

Используя эту реализацию, наша машина может вычислить функцию `fibonacci()` при `n = 40` почти мгновенно. Но когда мы пытаемся вычислить `fibonacci(50)`, мы сталкиваемся с другой проблемой:

```
>>> fibonacci(50)
OverflowError: integer addition
```

Ответ, как вы увидите через минуту `20365011074`. Проблема в том, что это число слишком большое чтобы поместиться в тип `int`. Такую ситуацию называют *переполнением*. К счастью эта проблема имеет простое решение. Ему посвящен следующий раздел.

§10.7. Тип «длинное целое число»

Питон предоставляет тип названный `long int`, который может хранить целые числа любого размера. Есть два пути создать значение типа `long int`. Первый – написать целое число с заглавной `L` в конце.

```
>>> type(1L)
<type 'long int'>
```

Другой способ – использовать функцию `long()` чтобы преобразовать значение к типу `long int`. Функция `long()` может принимать любые численные типы и даже строки цифр:

```
>>> long(1)
1L
>>> long(3.9)
3L
>>> long('57')
57L
```

Все математические операции работают с `long int`, значит, нам не придется сильно переделывать нашу функцию `fibonacci()`:

```
>>> previous = {0:1L, 1:1L}
>>> fibonacci(50)
20365011074L
```

Простым изменением начального содержимого словаря, мы изменяем поведение функции. Первые два числа в последовательности `long int`, поэтому все последующие числа в последовательности тоже.

Упражнение. *Измените функцию `factorial()` так, чтобы она возвращала результат типа `long int`. Протестируйте ее.*

§10.8. Подсчет букв

В упражнении *главы 7* мы написали функцию `countLetters`, которая подсчитывала число вхождений буквы в строку. Более общий вариант этой задачи – построение гистограммы букв в строке, то есть вычисление, сколько раз каждая буква появляется в строке. Такие гистограммы могут пригодиться для частотного анализа – одного из метода расшифровки кодов простой замены (например, шифра Цезаря)[19], или для компрессии текстовых файлов. Так как различные буквы появляются с различными частотами, мы можем

19 Кстати, метод частотного анализа использовал Шерлок Холмс для того, чтобы прочесть текст зашифрованный с помощью «пляшущих человечков».

сжать файл, используя короткие коды для распространенных букв и длинные коды для букв, которые появляются менее часто (алгоритм Фано, алгоритм Хемминга и другие).

Словари предоставляют элегантный способ создавать гистограммы:

```
>>> letterCounts = {}
>>> for letter in "Mississippi":
...     letterCounts[letter] = letterCounts.get (letter, 0) + 1
...
>>> letterCounts
{'M': 1, 's': 4, 'p': 2, 'i': 4}
```

Мы начинаем с пустого словаря. Для каждой буквы в строке мы находим текущий счетчик (возможно нулевой) и увеличиваем его на единицу. В конце словарь содержит пары: буквы и их частоты.

Для красоты можно вывести гистограмму в алфавитном порядке с помощью методов `items()` и `sort()`:

```
>>> letterItems = letterCounts.items()
>>> letterItems.sort()
>>> print letterItems
[('M', 1), ('i', 4), ('p', 2), ('s', 4)]
```

Вы встречались с методом `items()` ранее, но метод `sort()` – первый метод который вы применяете к спискам. Есть другие методы списков, включая `append()`, `extend()` и `reverse()`. Еще раз просмотрите справку (функция `help()`) для получения их детального описания.

Глава 11. Файлы и обработка исключений

Пока программа исполняется ее данные хранятся в памяти. Когда программа завершается или компьютер выключают, данные в памяти исчезают. Чтобы хранить данные постоянно, вы должны поместить их в файл. Файлы обычно сохраняют на жестком диске, дискете или компакт-диске.

Когда имеется большое количество файлов, их часто организуют в директории (также называемые «папками»). Каждый файл распознается по уникальному имени или по комбинации имени файла и имени директории.

Читая и записывая файлы, программы могут обмениваться информацией с друг другом и создавать пригодные для печати форматы, подобные PDF.

Работа с файлами во многом подобна работе с книгами. Чтобы прочитать книгу вы должны ее открыть. Когда вы закончите чтение – закрыть. Пока книга открыта, вы можете или ее читать или в нее писать. В обоих случаях вы знаете, в каком месте книги вы находитесь. Большую часть времени вы читаете книгу по порядку, но вы также можете пропустить какую-то часть.

Все это также относиться к файлам. Чтобы открыть файл вы должны указать его имя и определить хотите вы его читать или записать в него какую информацию.

Операция открытия файла создает файловый объект. В приведенном ниже примере мы с помощью переменной `f` ссылаемся на новый файловый объект.

```
>>> f = open("test.dat","w")
>>> print f
<open file 'test.dat', mode 'w' at fe820>
```

Функция `open()` принимает два аргумента. Первый – имя файла, второй – режим его открытия. Режим `"w"` означает, что мы открыли файл для записи (от английского "write").

Если файла с именем `test.dat` нет, он будет создан. Если такой файл уже существует, его содержимое будет заменено тем, что мы в него запишем.

When we print the file object, we see the name of the file, the mode, and the location of the object.

Когда мы распечатываем переменную, содержащую ссылку на файловый объект, мы видим имя файла, режим и адрес файлового объекта.

To put data in the file we invoke the write method on the file object:

Чтобы поместить данные в файл мы вызываем метод файлового объекта write:

>>> f.write("Now is the time")

>>> f.write("to close the file")

Closing the file tells the system that we are done writing and makes the file available for reading:

Операция закрытия файла говорит системе, что мы закончили запись и делаем файл доступным для чтения:

>>> f.close()

Now we can open the file again, this time for reading, and read the contents into a string. This time, the mode argument is "r" for reading:

Сейчас мы можем открыть файл снова, на этот раз для чтения и считать содержимое в строку. Для включения режима чтения передадим аргумент "r":

```
>>> f = open("test.dat","r")
```

If we try to open a file that doesn't exist, we get an error:

Если мы попытаемся открыть файл, который не существует, мы получим ошибку:

```
>>> f = open("test.cat","r")
IOError: [Errno 2] No such file or directory: 'test.cat'
```

Not surprisingly, the read method reads data from the file. With no arguments, it reads the entire contents of the file:

Не удивительно, метод read читает данные из файла. Без аргументов, он считает все содержимое файла:

NOTE: Не понятно, чему не удивляется автор в первом предложении абзаца. Тому, что read читает файлы? Или тому, что выдает ошибку, потому что не может прочитать не существующий файл?

```
>>> text = f.read()
>>> print text
Now is the timeto close the file
```

There is no space between "time" and "to" because we did not write a space between the strings.

Между "time" и "to" нет пробела, потому что мы не записывали пробел между строками.

read can also take an argument that indicates how many characters to read:

Метод read также принимает аргумент, который указывает, сколько символов считать:

```
>>> f = open("test.dat","r")
>>> print f.read(5)
Now i
```

Если не переоткрыть файл после предпоследнего примера, этот пример не сработает.

If not enough characters are left in the file, read returns the remaining characters. When we get to the end of the file, read returns the empty string:

Если в файле осталось меньше символов, чем указано в аргументе метода read, он считает все оставшиеся символы. Когда мы дойдем до конца файла, read вернет пустую строку:

NOTE: Смысл абзаца вроде такой. Здесь ИМХО надо бы сказать чего-нибудь о том, что существует указатель, который при последовательном чтении файла смещается к его концу на количество считанных символов (байт). Каждый последующий read() читает с того символа, на которое в данный момент указывает сей указатель...

```
>>> print f.read(1000006)
s the timeto close the file
>>> print f.read()


>>>
```

The following function copies a file, reading and writing up to fifty characters at a time. The first argument is the name of the original file; the second is the name of the new file:

Функция в следующем примере копирует файл, считывая и записывая до 50 символов за раз. Первый аргумент функции - имя оригинального файла, второй - имя нового файла:

```
def copyFile(oldFile, newFile):
  f1 = open(oldFile, "r")
  f2 = open(newFile, "w")
  while 1:
    text = f1.read(50)
    if text == "":
      break
    f2.write(text)
  f1.close()
  f2.close()
  return
```

The break statement is new. Executing it breaks out of the loop; the flow of execution moves to the first statement after the loop.

В примере есть новый оператор - break. Его выполнение приводит к выходу из цикла, поток исполнения перемещается на первый оператор после цикла.

In this example, the while loop is infinite because the value 1 is always true. The only way to get out of the loop is the execute break, which happens when text is the empty string, which happens when we get to the end of the file.

В этом примере цикл while бесконечный, так как значение 1 всегда эквивалентно истине. Есть только один способ выйти из цикла - выполнить оператор break, что происходит когда text содержит пустую строку, когда мы доходим до конца файла.

§11.1. Текстовые файлы

A text file is a file that contains printable characters and whitespace, organized into lines separated by newline characters. Since Python is specifically designed to process text files, it provides methods that make the job easy.

Текстовые файлы - файлы которые содержат печатные символы и пробелы, сгруппированные в строки, которые отделены друг от друга символами новой строки. Поскольку Питон специально создан для обработки текстовых файлов, он предоставляет методы, которые делают такую работу простой.

NOTE: Ну вообще то текстовыми ИМХО являются любые файлы, которые содержат информацию, представленную в виде любых закодированных символов. Так, например, у меня одни весы пишут в файл результаты взвешиваний в виде чисел в кодах ASCII - такой файл текстовый, другие записывают результаты в формате Excel, где числа представлены не кодами символов цифр, а своим двоичным представлением - такой файл уже не есть текстовый. А уж печатные символы в текстовом файле или не печатные, пробелы там разделители или амперсанды с диезами, разбит ли он на строки или не разбит, уже не имеет ИМХО ни какого значения.

To demonstrate, we'll create a text file with three lines of text separated by newlines:

Для иллюстрации мы создадим текстовый файл с тремя строками текста, разделенными символом новой строки:

```
>>> f = open("test.dat","w")
>>> f.write("line one\nline two\nline three\n")
>>> f.close()
```

The readline method reads all the characters up to and including the next newline character:

Метод readline читает все символы до символа новой строки включительно:

```
>>> f = open("test.dat","r")
>>> print f.readline()
line one

>>>
```

readlines returns all of the remaining lines as a list of strings:

Метод readlines возвращает все оставшиеся в файле строки в виде списка строк:

```
>>> print f.readlines()
['line two\012', 'line three\012']
```

In this case, the output is in list format, which means that the strings appear with quotation marks and the newline character appears as the escape sequence
012.

В этом случае вывод представлен в формате списка, где строки заключены в кавычки, а символ новой строки появляется в виде управляющей последовательности
012.

**NOTE: Честно говоря, я не понял всей глубины сего абзаца. Если readlines возвращает список, то само собой очевидно, что и вывод print'a будет списком, в котором строки будут заключены в кавычки, о чем уже писалось в главе этим самым спискам посвященной. И что означает html тег
 перед кодом символа новой строки?**

At the end of the file, readline returns the empty string and readlines returns the empty list:

Если файл уже считан полностью, то readline вернет пустую строку, а readlines вернет пустой список:

```
>>> print f.readline()

>>> print f.readlines()
[]
```

The following is an example of a line-processing program. filterFile makes a copy of oldFile, omitting any lines that begin with #:

Следующий пример - программа процессор строк. Функция filterFile создает копию файла oldFile пропуская все строки которые начинаются с #:

С символа # обычно начинается однострочковый комментарий в различных *nix ориентированных скриптовых языках. Ну вот хотя бы в Питоне например...

```
def filterFile(oldFile, newFile):
  f1 = open(oldFile, "r")
  f2 = open(newFile, "w")
  while 1:
    text = f1.readline()
    if text == "":
      break
    if text[0] == '#':
      continue
    f2.write(text)
```

```
    f1.close()
    f2.close()
    return
```

The continue statement ends the current iteration of the loop, but continues looping. The flow of execution moves to the top of the loop, checks the condition, and proceeds accordingly.

Оператор continue завершает текущую итерацию цикла, но не завершает цикл. Поток исполнения перемещается в начало цикла, проверяет условие и продолжает вычисления в соответствии с результатом проверки.

Thus, if text is the empty string, the loop exits. If the first character of text is a hash mark, the flow of execution goes to the top of the loop. Only if both conditions fail do we copy text into the new file.

Так, если text содержит пустую строку, цикл завершается. Если первый символ в строке text - диез (# - «решетка»), поток исполнения переходит в начало цикла. Только если не выполняются оба условия, мы копируем text в новый файл.

§11.2. Запись переменных

The argument of write has to be a string, so if we want to put other values in a file, we have to convert them to strings first. The easiest way to do that is with the strfunction:

Аргумент метода write должен быть строкой, так если вы хотите поместить в файл значения других типов, вы должны сначала преобразовать их в строку. Простейший способ воспользоваться функцией str:

```
>>> x = 52
>>> f.write (str(x))
```

An alternative is to use the format operator %. When applied to integers, % is the modulus operator. But when the first operand is a string, % is the format operator.

Альтернатива использовать оператор форматирования %. Когда % применяется к целым числам, он возвращает остаток от деления (оператор деления по модулю). Но когда первый оператор строка, % оператор форматирования.

The first operand is the format string, and the second operand is a tuple of expressions. The result is a string that contains the values of the expressions, formatted according to the format string.

Первый оператор форматирующая строка, второй оператор - кортеж выражений. Результат - строка, содержащая значения выражений, отформатированных в соответствии с форматирующей строкой.

As a simple example, the format sequence "%d" means that the first expression in the tuple should be formatted as an integer. Here the letter d stands for "decimal":

В качестве простого примера - форматирующая последовательность "%d" означает, что первое выражение в кортеже должно быть отформатировано как целое число. Здесь буква d означает «decimal» - десятичный.

NOTE: Судя по всему, должна существовать форматирующая последовательность для чисел в других системах счисления.

```
>>> cars = 52
>>> "%d" % cars
'52'
```

The result is the string '52', which is not to be confused with the integer value 52.

Результат - строка '52', которую не следует путать с целым числом 52.

A format sequence can appear anywhere in the format string, so we can embed a value in a sentence:

Форматирующая последовательность может появляться где угодно в форматирующей строке, так мы можем вставить значение переменной в фразу:

```
>>> cars = 52
>>> "In July we sold %d cars." % cars
'In July we sold 52 cars.'
```

The format sequence "%f" formats the next item in the tuple as a floating-point number, and "%s" formats the next item as a string:

Форматирующая последовательность "%f" форматирует следующий элемент в кортеже как число с плавающей точкой, а "%s" как строку:

```
>>> "In %d days we made %f million %s." % (34, 6.1, 'dollars')
'In 34 days we made 6.100000 million dollars.'
```

By default, the floating-point format prints six decimal places.

По умолчанию, формат чисел с плавающей точкой предоставляет шесть разрядов после запятой.

NOTE: У всего человечества исповедующего демократические ценности в десятичных дробях разделитель - точка. У нас же запятая, поэтому не понятно, что писать «числа с плавающей запятой» или «шесть разрядов после десятичной точки»?

The number of expressions in the tuple has to match the number of format sequences in the string. Also, the types of the expressions have to match the format sequences:

Количество выражений в кортеже должно соответствовать числу форматирующих последовательностей в строке. Также тип выражений должен соответствовать форматирующим последовательностям:

```
>>> "%d %d %d" % (1,2)
TypeError: not enough arguments for format string
>>> "%d" % 'dollars'
TypeError: illegal argument type for built-in operation
```

In the first example, there aren't enough expressions; in the second, the expression is the wrong type.

В первом примере не хватает выражений, во втором - не правильный тип выражения.

For more control over the format of numbers, we can specify the number of digits as part of the format sequence:

Мы можем дополнительно управлять форматом чисел, устанавливая, как часть форматирующей последовательности, количество выводимых цифр:

```
>>> "%6d" % 62
'    62'
>>> "%12f" % 6.1
'    6.100000'
```

The number after the percent sign is the minimum number of spaces the number will take up. If the value provided takes fewer digits, leading spaces are added. If the number of spaces is negative, trailing spaces are added:

Число после знака процента - минимальное количество знакомест, которое будет занято. Если предоставляемое значение занимает только несколько цифр, добавляются

лидирующие пробелы. Если число выводимых цифр отрицательно, добавляются замыкающие пробелы:

```
>>> "%-6d" % 62
'62    '
```

For floating-point numbers, we can also specify the number of digits after the decimal point:

Для чисел с плавающей точкой, мы так же можем определить число цифр после десятичной точки:

```
>>> "%12.2f" % 6.1
'        6.10'
```

In this example, the result takes up twelve spaces and includes two digits after the decimal. This format is useful for printing dollar amounts with the decimal points aligned.

В примере результат занимает двенадцать знакомест и включает две цифры после десятичной точки. Этот формат полезен для печати сумм в долларах, выровненных по десятичной точке.

NOTE: Суммы в рублях тоже не плохо печатаются :) Только вот знакомест формат в примере занимает не 12 а 13 (десятичная точка), впрочем автор говорит о цифрах...

For example, imagine a dictionary that contains student names as keys and hourly wages as values. Here is a function that prints the contents of the dictionary as a formatted report:

Например вообразим словарь, который содержит - имена студентов в качестве ключей и почасовую оплату в качестве значений. Функция report распечатывает содержимое словаря как отформатированный отчет:

```
def report (wages) :
  students = wages.keys()
  students.sort()
  for student in students :
    print "%-20s %12.02f" % (student, wages[student])
```

To test this the function, we'll create a small dictionary and print the contents:

Для проверки этой функции создадим маленький словарь и распечатаем содержимое:

```
>>> wages = {'mary': 6.23, 'joe': 5.45, 'joshua': 4.25}
>>> report (wages)
joe                  5.45
joshua               4.25
mary                 6.23
```

By controlling the width of each value, we guarantee that the columns will line up, as long as the names contain fewer than twenty-one characters and the wages are less than one billion dollars an hour.

Задавая ширину каждого значения, мы гарантируем, что столбцы будут выровнены до тех пор, пока имя содержит менее двадцати одного символа и почасовая оплата менее одного миллиарда долларов.

Выводимая в примере таблица будет выглядеть выровненной при условии использования моноширного шрифта, например Courier New или Terminal.

§11.3. Директории

When you create a new file by opening it and writing, the new file goes in the current directory (wherever you were when you ran the program). Similarly, when you open a file for reading, Python looks for it in the current directory.

Когда вы создаете новый файл, открывая и записывая его, новый файл направляется в текущую директорию (ту, где вы находились, когда запускали программу). Аналогично, когда вы открываете файл для чтения, Питон ищет его в текущей директории.

If you want to open a file somewhere else, you have to specify the path to the file, which is the name of the directory (or folder) where the file is located:

Если вы желаете открыть файл где-либо в другом месте, вы должны указать путь к файлу, который содержит имя директории (или папки) в которой располагается файл:

```
>>> f = open("/usr/share/dict/words","r")
>>> print f.readline()
Aarhus
```

This example opens a file named words that resides in a directory named dict, which resides in share, which resides in usr, which resides in the top-level directory of the system, called /.

В примере открывается файл с именем words, который находится в директории с именем dict, которая находится в share, которая в свою очередь находится в usr, usr находится в корневой директории системы, названной / - root, корень.

You cannot use / as part of a filename; it is reserved as a delimiter between directory and filenames.

Вы не должны использовать / как часть имени файла, этот символ зарезервирован в качестве разделителя между именами директорий и именами файлов.

The file /usr/share/dict/words contains a list of words in alphabetical order, of which the first is the name of a Danish university.

Файл /usr/share/dict/words содержит список слов в алфавитном порядке, первое - имя Датского университета.

Действительно, по указанному пути существует файл words, ссылка на файл linux.words, первые три слова Aarhus, Aaron, Ababa...

§11.4. Pickling

In order to put values into a file, you have to convert them to strings. You have already seen how to do that with str:

Чтобы записать значение в файл, вы должны преобразовать его в строку. Вы совсем недавно видели, как это сделать с функцией str:

```
>>> f.write (str(12.3))
>>> f.write (str([1,2,3]))
```

The problem is that when you read the value back, you get a string. The original type information has been lost. In fact, you can't even tell where one value ends and the next begins:

Когда вы считываете записанные значения обратно, возникает проблема, вы получаете строку. Начальная информация о типе значения теряется. Фактически, вы даже не можете сказать, где заканчивается одно значение и начинается другое:

```
>>> f.readline()
'12.3[1, 2, 3]'
```

The solution is pickling, so called because it "preserves" data structures. The pickle module contains the necessary commands. To use it, import pickle and then open the file in the usual way:

Решение проблемы - засолка, называется так, потому что «сохраняет» структуру данных. Модуль pickle содержит необходимые команды. Чтобы его использовать, импортируйте pickle и затем как обычно откройте файл:

Слово pickling имеет следующие значения:1) квашение, засол, маринование; 2) протравливание, травление, декапирование.

```
>>> import pickle
>>> f = open("test.pck","w")
```

To store a data structure, use the dump method and then close the file in the usual way:

Чтобы сохранить структуру данных, используйте метод dump после чего закройте файл как обычно.

```
>>> pickle.dump(12.3, f)
>>> pickle.dump([1,2,3], f)
>>> f.close()
```

Then we can open the file for reading and load the data structures we dumped:

Позже вы можете открыть файл для чтения и загрузить данные, которые мы сохранили:

```
>>> f = open("test.pck","r")
>>> x = pickle.load(f)
>>> x
12.3
>>> type(x)
<type 'float'>
>>> y = pickle.load(f)
>>> y
[1, 2, 3]
>>> type(y)
<type 'list'>
```

Each time we invoke load, we get a single value from the file, complete with its original type.

Каждый раз, когда мы вызываем метод load, мы получаем из файла одно значение, соответствующее своему оригинальному типу.

§11.5. Исключения

Whenever a runtime error occurs, it creates an exception. Usually, the program stops and Python prints an error message.

Всякий раз, когда во время исполнения программы появляется ошибка, она порождает исключение. Обычно программа останавливается, и Питон выдает сообщение об ошибке.

For example, dividing by zero creates an exception:

Например деление на ноль порождает исключение:

```
>>> print 55/0
ZeroDivisionError: integer division or modulo
```

So does accessing a nonexistent list item:

Тоже происходит при попытке обратиться к несуществующему элементу списка:

```
>>> a = []
>>> print a[5]
IndexError: list index out of range
```

Or accessing a key that isn't in the dictionary:

Или при попытке получить значение по не существующему ключу словаря:

```
>>> b = {}
>>> print b['what']
KeyError: what
```

In each case, the error message has two parts: the type of error before the colon, and specifics about the error after the colon. Normally Python also prints a traceback of where the program was, but we have omitted that from the examples.

В каждом случае сообщение об ошибке состоит из двух разделенных двоеточием частей: типа ошибки и дополнительной информации. Обычно Питон также выводит стек вызовов функций на момент возникновения исключения, но мы убрали его из примеров.

Sometimes we want to execute an operation that could cause an exception, but we don't want the program to stop. We can handle the exception using the try and except statements.

Иногда мы хотим выполнить операции, которые могут вызвать исключения, но мы не хотим, чтобы программа остановилась в случае появления ошибки. Мы можем обработать исключение, используя операторы try и except.

For example, we might prompt the user for the name of a file and then try to open it. If the file doesn't exist, we don't want the program to crash; we want to handle the exception:

Например, мы можем запросить у пользователя имя файла и затем попытаться открыть этот файл. Мы не хотим, чтобы программа обрушилась, если такой файл не существует, мы хотим обработать исключение:

```
filename = raw_input('Enter a file name: ')
try:
  f = open (filename, "r")
except:
  print 'There is no file named', filename
```

The try statement executes the statements in the first block. If no exceptions occur, it ignores the except statement. If any exception occurs, it executes the statements in the except branch and then continues.

Оператор try выполняет команды первого блока. Если исключение не произошло, он пропускает блок операторов после except. Если произошло исключение, try выполняет операторы в блоке except и затем передает управление следующему оператору.

We can encapsulate this capability in a function: exists takes a filename and returns true if the file exists, false if it doesn't:

Мы можем заключить эту возможность в функцию: exists принимает имя файла и возвращает true если файл с таким именем существует, false если нет:

```
def exists(filename):
  try:
    f = open(filename)
    f.close()
```

```
    return 1
  except:
    return 0
```

You can use multiple except blocks to handle different kinds of exceptions. The Python Reference Manual has the details.

Вы можете использовать несколько блоков except, чтобы обрабатывать ошибки различных типов. Подробности приведены в Python Reference Manual.

If your program detects an error condition, you can make it raise an exception. Here is an example that gets input from the user and checks for the value 17. Assuming that 17 is not valid input for some reason, we raise an exception.

Если ваша программа способна определить условия, при котрых возникает ошибка, вы можете вызвать (возбудить) собственное исключение. Вот пример функции, которая принимает ввод пользователя и проверяет равно ли введенное значение 17. Предположив, что 17 по какой-то причине не является корректным вводом, мы вызываем исключение:

```
def inputNumber () :
  x = input ('Pick a number: ')
  if x == 17 :
    raise 'BadNumberError', '17 is a bad number'
  return x
```

The raise statement takes two arguments: the exception type and specific information about the error. BadNumberErroris a new kind of exception we invented for this application.

Оператор raise принимает два аргумента: тип исключения и дополнительную информацию об ошибке. BadNumberErroris новый тип исключения, который мы придумали для своего приложения.

If the function that called inputNumber handles the error, then the program can continue; otherwise, Python prints the error message and exits:

Если функция, которая вызвала inputNumber, обработает ошибку, выполнение программы может продолжиться, в противном случае, Питон выдаст сообщение об ошибке и завершит программу:

```
>>> inputNumber ()
Pick a number: 17
BadNumberError: 17 is a bad number
```

The error message includes the exception type and the additional information you provided.

Сообщение об ошибке включает тип ошибки и ту дополнительную информацию, которую вы предоставили.

As an exercise, write a function that uses inputNumber to input a number from the keyboard and that handles the BadNumberError exception.

Упражнение: напишите функцию, которая использует inputNumber, для ввода чисел с клавиатуры и обрабатывает исключение BadNumberError.

Глава 12. Классы и объекты

Глава 13. Классы и функции

Глава 14. Методы

Глава 15. Наборы объектов

Глава 16. Наследование

Глава 17. Связные списки

Глава 18. Стеки

Глава 19. Очереди и очереди с приоритетами

Глава 20. Деревья

Глава 21. Функциональное программирование

Заключение. С высоты птичьего полета

«Ты можешь подняться выше, Джонатан, потому что ты учился. Ты окончил одну школу, теперь настало время начать другую»

Р. Бах, «Чайка по имени Джонатан Ливингстон».

Приложение А. Советы по отладке программ

Приложение B. Создание и использование модулей

Приложение C. Создание типов данных

Приложение D. Написание программ с графическим интерфейсом

Приложение Е. Методологии командной разработки

Приложение F. Методические указания преподавателям