МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное
образовательное учреждение высшего образования
«СЕВАСТОПОЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт радиоэлектроники и информационной безопасности

Кафедра информационной безопасности

М.А. Артеменко, А.В. Лебеденко, А.О. Сафьяник

ОСНОВЫ ЗАЩИТЫ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ВВЕДЕНИЕ В PYTHON

Учебное пособие

Севастополь
СевГУ
2018

УДК 004.432
А86

Р е ц е н з е н т:
А.В. Колесников - кандидат технических наук, доцент

Ответственный за выпуск - М.И. Ожиганова, кандидат технических наук,
заведующий кафедрой информационной безопасности

Артеменко М.А.

А86 Основы защиты программного обеспечения. Введение в Python: учеб. пособие / М.А. Артеменко, А.В. Лебеденко, А.О. Сафьяник. – Севастополь: СевГУ, 2018. – 99 с.

Учебное пособие знакомит читателя с основами программирования на языке Python версии 3+. Пособие затрагивает простые структуры данных, такие как строки, списки, словари, кортежи, функции, организацию циклов и ветвлений, списковые сборки, а также касается методов объектно-ориентированного программирования.

Пособие ориентировано на студентов, знакомых с программированием, но ранее не изучавшим язык Python. Пособие предлагается как основной материал по изучению дисциплины «Основы защиты программного обеспечения».

УДК 004.432

Учебное пособие утверждено на заседании кафедры информационной безопасности, протокол № 2 от 2 сентября 2015 г.

.

ОГЛАВЛЕНИЕ

1. ОПИСАНИЕ

Python — мощный и простой для изучения язык программирования. В нем предоставлены проработанные высокоуровневые структуры данных и простой, но эффективный подход к объектно-ориентированному программированию. Сочетание изящного синтаксиса и динамической типизации, совмещенных с интерпретируемой сущностью, делает Python идеальным языком для написания сценариев и ускоренной разработки приложений в различных сферах и на большинстве платформ.

Интерпретатор Python и разрастающаяся стандартная библиотека находятся в свободном доступе в виде исходников и бинарных файлов для всех основных платформ на официальном сайте Python http://www.python.org и могут распространяться без ограничений. Кроме этого на сайте содержатся дистрибутивы и ссылки на многочисленные модули третьих сторон для языка Python, различные программы и инструменты, а также дополнительная документация.

Интерпретатор Python может быть легко расширен с помощью новых функций и типов данных, написанных на C/C++ (или других языков, к которым можно получить доступ из C). Также Python можно применять как язык расширений для настраиваемых приложений.

Это учебное пособие неформально представляет читателю основные концепции и возможности языка и системы Python. Полезно держать интерпретатор Python под рукой для получения практического опыта, но при этом все примеры самодостаточны, так что пособие вполне возможно читать вне сети.

Описание стандартных объектов и модулей вы можете найти в справочнике по библиотеке Python. Руководство по языку дает более формальное описание языка. Перед написанием расширений на C/C++ ознакомьтесь с руководством по расширению и встраиванию в интерпретатор и справочником по Python/C API. Существует также несколько книг, в которых подробно рассмотрен язык Python.

Это пособие не претендует на звание всеобъемлющего и не описывает каждую особенность Python: он даже не описывает *всех* его часто использующихся особенностей. Вместо этого он знакомит читателя с наиболее заслуживающими внимания из них и дает представление о стиле и *привкусе* языка. После прочтения учебного пособия вы сможете писать и читать программы и модули, написанные на Python, и будете готовы узнать больше о различных модулях библиотеки Python, описанных в справочнике по библиотеке Python.

Кроме того, будет нелишним полистать Глоссарий.

2. РАЗЖИГАЯ ВАШ АППЕТИТ

Если вы много работаете за компьютером — то периодически обнаруживаете задачи, которые хотели бы автоматизировать. Например, вы были бы не против ускорить операцию поиска и замены большого количества текстовых файлов, или как-то своеобразно переименовать и отсортировать набор фотографий. Возможно, вам хотелось бы написать небольшую базу данных на заказ, специализированное GUI-приложение или простую игру.

Если вы профессиональный разработчик программных продуктов — вероятно, вы привыкли работать с несколькими библиотеками на C/C++/Java, но находите обычный цикл написания/компиляции/тестирования/перекомпиляции кода чересчур медленным. Возможно, вы пишете набор тестов для такой библиотеки, и процесс написания тестирующего кода воспринимается утомительным. Или же вы написали программу, которая должна использовать специальный язык для расширений и не хотите проектировать и разрабатывать полностью новый язык для вашего приложения.

Python — язык, который вам подойдет.

Вы можете написать шелл-сценарий для Unix или использовать пакетные файлы Windows для некоторых из этих задач, но шелл-сценарии хороши лишь для перемещения файлов и замены текстовых данных — они вряд ли подойдут для написания приложений, снабженных графическим интерфейсом, или игр. Вы можете написать программу на C/C++/Java, но разработка может занять довольно много времени — даже на то, чтобы получить первый рабочий набросок, его требуется немало. Python — проще в использовании, доступен на операционных системах Windows, Mac OS X и Unix, и поможет сделать работу намного быстрее.

Даже учитывая легкость использования, Python — полноценный язык программирования, предлагающий намного больше возможностей для структурирования и поддержки крупных программ, чем могут позволить себе шелл-сценарии или пакетные файлы. С другой стороны, Python также предоставляет намного больше информации для отладки ошибок чем C и, будучи *сверхвысокоуровневым языком*, имеет встроенные высокоуровневые типы данных — такие, как гибкие массивы и словари. Благодаря наличию обобщенных типов данных Python применим для более широкого круга приложений чем Awk или даже Perl, и при этом очень многие вещи остаются в языке Python как минимум настолько же простыми, насколько просты в этих языках.

Python позволяет вам разделить вашу программу на модули, которые могут быть заново использованы в других программах на Python. Он поставляется вместе с внушительной коллекцией стандартных модулей, которые вы можете использовать в качестве фундамента ваших программ; или в качестве примеров для того, чтобы начать изучение Python. Многие

из этих модулей предоставляют различную полезную функциональность: например, ввод-вывод для файлов, системные вызовы, сокеты, и даже инструменты для создания графического пользовательского интерфейса — такие, как Tk.

Python — интерпретируемый язык, и это может сохранить вам немало времени при разработке — в компиляции и связывании нет необходимости. Интерпретатор может использоваться в интерактивном режиме, что позволяет легко экспериментировать с возможностями языка, писать программы-однодневки или проверять функции во время разработки программ методом снизу-вверх. И, кроме всего прочего — это удобный настольный калькулятор.

Python дает возможность писать компактные и читабельные программы. Программы, написанные на Python отличаются большей краткостью чем эквиваленты на C, C++ или Java, по нескольким причинам:

− высокоуровневые типы данных позволяют вам выражать сложные операции в одной инструкции;

− группировка инструкций выполняется отступами, а не операторными скобками;

− нет необходимости в описании переменных или аргументов.

Python *расширяем*: если вы знаете, как программировать на C, то вам легко будет добавить к интерпретатору новую встроенную функцию или модуль, выполнить критические операции на максимальной скорости или связать программы на Python с библиотеками, которые могут быть доступны только в бинарной форме (например, зависящие от поставщика графические библиотеки). Если вы действительно увлечены — вы можете привязать интерпретатор Python к приложению, написанному на C — чтобы использовать его как язык расширений или командный язык этого приложения.

Кстати, язык назван в честь шоу «Летающий цирк Монти Пайтона» на BBC и не имеет ничего общего с рептилиями. Ссылки на скетчи Монти Пайтона в документации не только позволены, но и поощряются!

Теперь, когда ваш интерес к Python полностью пробужден, вам наверняка захочется изучить его более детально. В связи с тем, что лучший способ выучить язык — использовать его — автор приглашает вас использовать интерпретатор Python в процессе чтения.

В следующей главе описана механика использования интерпретатора. Это довольно приземленная информация, но она необходима для работы с примерами, которые будут представлены позже.

Остальная часть учебника описывает различные возможности языка и системы Python за счет примеров — начиная с простых выражений, операторов и типов данных, рассматривая функции и модули, и заканчивая прикосновением к таким продвинутым концепциям как исключения и определенные пользователем классы.

3. ИСПОЛЬЗОВАНИЕ ИНТЕРПРЕТАТОРА PYTHON

3.1. Запуск интерпретатора

Интерпретатор Python после установки располагается, обычно, по пути `/usr/local/bin/python3.1` — на тех компьютерах, где этот путь доступен. Добавление каталога `/usr/local/bin` к пути поиска Unix-шелла (переменная `PATH`) позволит запустить интерпретатор набором команды `python3.1`прямо из шелла. Поскольку выбор каталога, в котором будет обитать интерпретатор, осуществляется при его установке, то возможны и другие варианты — посоветуйтесь с вашим Python-гуру или системным администратором. (Например, путь `/usr/local/python` тоже популярен в качестве альтернативного расположения.)

На машинах с ОС Windows, инсталляция Python обычно осуществляется в каталог `C:\Python31`, но и он может быть изменен во время установки. Чтобы добавить этот каталог к вашему пути поиска, вы можете набрать в окне DOS следующую команду, в ответ на приглашение:

```
set path=%path%; C:\python31
```

При наборе символа конца файла (`Ctrl-D` в Unix, `Ctrl-Z` в Windows) в ответ на основное приглашение интерпретатора, последний будет вынужден закончить работу с нулевым статусом выхода. Если это не сработает — вы можете выйти из интерпретатора путем ввода следующих команд:

```
import sys; sys.exit()
```

Особенности редактирования строк в интерпретаторе не оказываются обычно чересчур сложными. Те, кто установил интерпретатор на машину Unix, потенциально имеют поддержку библиотеки `GNU readline`, обеспечивающей усовершенствованное интерактивное редактирование и сохранение истории. Самый быстрый, наверное, способ проверить, поддерживается ли расширенное редактирование командной строки, заключается в нажатии `Ctrl-P` в ответ на первое полученное приглашение Python. Если вы услышите сигнал — значит вам доступно редактирование командной строки — тогда обратитесь к *Приложению об Интерактивном редактировании входных данных* за описанием клавиш. Если на ваш взгляд ничего не произошло или отобразился символ `^P` — редактирование командной строки недоступно — удалять символы из текущей строки возможно будет лишь использованием клавиши `Backspace`.

Интерпретатор ведет себя сходно шеллу Unix: если он вызван, когда *стандартный ввод* привязан к устройству `tty` — он считывает и выполняет команды в режиме диалога; будучи вызванным с именем файла в качестве параметра или с файлом, назначенным на *стандартный ввод* — он читает и выполняет сценарий из этого файла.

Другой способ запустить интерпретатор — директива **python -c***команда* [arg] ... — при ее использовании поочередно выполняются инструкции(-ция) из *команды* (как при использовании опции **-c** Unix-шелла). В связи с тем, что инструкции Python часто содержат пробелы или другие специальные для шелла символы, рекомендуется заключать *команды* полностью в одинарные кавычки.

Некоторые модули Python оказываются полезными при использовании их в качестве сценариев. Они могут быть запущены в этом виде командой **python -m** *модуль* [arg] ... — таким образом исполняется исходный файл модуля (как произошло бы, если бы вы ввели его полное имя в командной строке).

Обратите внимание на различие между командами `python file` и `python <file`. В последнем случае запросы на ввод из программы, такие как вызов `sys.stdin.read()`, удовлетворяются из самого файла. Поскольку файл уже был прочитан до конца еще до начала выполнения программы — символ конца файла будет обнаружен программой незамедлительно. В большинстве случаев (это, чаще всего, и есть те ситуации, которых вы ожидали) вызовы удовлетворяются независимо от того, файл или устройство привязаны к стандартному вводу интерпретатора Python.

При использовании файла сценария иногда полезно иметь возможность запустить сценарий и затем войти в интерактивный режим. Это может быть сделано через указание параметра **-i** перед именем сценария. (Этот способ не сработает, если сценарий считывается со стандартного ввода — по той самой причине, которая описана в предыдущем абзаце).

3.1.1. Передача параметров

В случае, если интерпретатору известны имя сценария и дополнительные параметры, с которыми он вызван, все они передаются сценарию в переменной `sys.argv`, представляющей собой список (`list`) строк. Длина (`length`) списка — минимум, единица; если не переданы ни имя сценария, ни аргументы — то `sys.argv[0]` содержит пустую строку. Когда в качестве имени сценария передан `'-'` (означает *стандартный ввод*), `sys.argv[0]` устанавливается в `'-'`. Если используется директива **-c** *команда* — `sys.argv[0]` содержит `'-c'`. В случае, если используется директива **-m** *модуль* — то `sys.argv[0]` устанавливается равным полному имени модуля по расположению. Опции, обнаруженные после сочетаний **-c** *команда* или **-m** *модуль* не обрабатываются интерпретатором Python, но остаются в переменной `sys.argv`, дабы обеспечить возможность отслеживания в самой команде или в модуле.

3.1.2. Интерактивный режим

Если команды считываются с `tty` — говорят, что интерпретатор находится в *интерактивном режиме* (режиме диалога). В этом режиме он

приглашает к вводу следующей команды, отобразив *основное приглашение* (обычно это три знака «больше-чем» — >>>); в то же время, для *продолжающих строк* выводится *вспомогательное приглашение* (по умолчанию — три точки — `...`). Перед выводом первого приглашения интерпретатор отображает приветственное сообщение, содержащее номер его версии и пометку о правах копирования:

```
$ python3.1
Python 3.1a1 (py3k, Sep 12 2007, 12:21:02)
[GCC 3.4.6 20060404 (Red Hat 3.4.6-8)] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>>
```

Продолжающие строки используются в случаях, когда необходимо ввести многострочную конструкцию. Взгляните, например, на следующий оператор `if`:

```
>>> the_world_is_flat = 1
>>> if the_world_is_flat:
... print("Be careful not to fall off!")
...
Be careful not to fall off!
```

3.2. Интерпретатор и его окружение

3.2.1. Обработка ошибок

В случае появления ошибки интерпретатор выводит сообщение об ошибке, завершая его стеком вызовов. В интерактивном режиме он снова возвращается в состояние приглашения для ввода команд; если ввод происходит из файла — интерпретатор выходит с ненулевым статусом, сразу после распечатки стека вызовов. (Исключения, обрабатываемые в блоке `except` оператора `try` в этом контексте не считаются ошибками.) Некоторые ошибки исключительно фатальны и вызывают собой принудительное завершение работы с ненулевым статусом — это применимо к внутренним противоречиям языка и к некоторым случаям нехватки памяти. Все сообщения об ошибках выводятся в *стандартный поток ошибок* (`error stream`). Обычный вывод исполняемых команд направляется в *стандартный вывод*.

Нажатие клавиш прерывания процесса (обычно `Ctrl-C` или `DEL`), в ответ на приглашение в основном или вспомогательном режиме, отменяет ввод и возвращает вас к основному приглашению. Символ прерывания, набранный во время выполнения какой-либо команды порождает исключение `KeyboardInterrupt`, которое, в свою очередь, может быть перехвачено оператором `try`.

3.2.2. Исполняемые сценарии на Python

На Unix-системах семейства BSD сценарии на Python могут быть сделаны исполняемыми, также как и шелл-сценарии, путем добавления следующей строки

```
#! /usr/bin/env python3.1
```

(предполагается, что интерпретатор может быть найден по одному из путей, указанных в пользовательской переменной `PATH`) в начало сценария и установки файла в исполняемый режим. Символы `#!` должны быть первыми символами в файле. На некоторых платформах строка должна оканчиваться символом конца строки в стиле Unix (`'\n'`), а не в стиле Windows (`'\r\n'`). Заметьте, что символ решетки `'#'` используется в Python для указания начала комментария.

Исполняемый режим (или разрешение на исполнение) может быть установлен сценарию использованием команды **chmod**:

```
$ chmod +x myscript.py
```

У систем с операционной системой Windows нет такого понятия, как исполняемый режим. Установщик Python автоматически связывает файлы `.py` с файлом `python.exe`, таким образом двойной клик на файле Python запустит его в виде сценария. Расширение может быть и `.pyw` в случае, если окно консоли (которое, обычно, отображается) при запуске сценария подавляется.

3.2.3. Кодировка исходных файлов

По умолчанию, исходники Python считаются созданными в кодировке `UTF-8`. В этой кодировке в строковых литералах, идентификаторах и комментариях могут быть использованы символы большинства языков мира — учитывая то, что стандартная библиотека Python использует символы `ASCII` для именования идентификаторов — и этому соглашению должен следовать любой переносимый код. Для корректного отображения всех этих символов, ваш редактор должен опознавать файл как закодированный в `UTF-8` и должен использовать шрифт, который содержит все символы, используемые в файле.

Также, есть возможность указать другую кодировку для исходных файлов. Для этого нужно добавить специальный комментарий следом за строкой `#!`, дабы описать кодировку исходного файла:

```
# -*- coding: encoding -*-
```

Если используется это описание — все, что находится в этом файле будет опознаваться как имеющее соответствующую кодировку *`encoding`*, а не `UTF-8`. Список возможных кодировок представлен в *Справочнике по библиотеке* — в разделе, описывающем модуль *`codecs`*.

Например, если выбранный вами редактор не поддерживает файлы, закодированные `UTF-8`, и требует применения какой-либо другой кодировки, допустим `Windows-1252`, вы можете написать:

```
# -*- coding: cp-1252 -*-
```

И, с этого момента, использовать в исходных файлах только символы из таблицы символов `Windows-1252`. Устанавливающий (отличную от установленной по умолчанию) кодировку специальный комментарий должен являться первой или второй строкой файла.

3.2.4. Интерактивный файл запуска

Если вы используете Python интерактивно — часто бывает удобным выполнить некоторые стандартные команды перед запуском интерпретатора. Вы можете сделать это, установив переменную окружения с именем `PYTHONSTARTUP` равной имени файла, содержащего ваши команды запуска. Способ схож с использованием файла `.profile` в Unix-шелле.

Этот файл читается только в интерактивных сессиях, но не в случае считывания команд из сценария, и не в случае, если `/dev/tty` назначен как независимый источник команд (который в других случаях ведет себя сходно интерактивным сессиям). Файл исполняется в том же пространстве имен, что и исполняемые команды - поэтому объекты и импортированные модули, которые он определяет, могут свободно использоваться в интерактивной сессии. Также в этом файле вы можете изменить приглашения: `sys.ps1` и `sys.ps2`.

Если вы хотите прочитать дополнительный файл запуска из текущего каталога — вы можете использовать код вроде:

```
If os.path.isfile('.pythonrc.py'): ex-
ec(open('.pythonrc.py').read())
```

Если вы хотите использовать файл запуска в сценарии — вам нужно будет указать это явно:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
exec(open(filename).read())
```

4. НЕФОРМАЛЬНОЕ ВВЕДЕНИЕ В PYTHON

В приведенных далее примерах, ввод и вывод различаются присутствием и отсутствием приглашений соответственно (приглашениями являются >>> и `...`): чтобы воспроизвести пример — вам нужно ввести все, что следует за приглашением, после его появления; строки, не начинающиеся с приглашений, являются выводом интерпретатора. Обратите внимание, что строка, в которой содержится лишь вспомогательное приглашение ("`...` ") означает, что вам нужно ввести пустую строку — этот способ используется для завершения многострочных команд.

Большинство примеров в этом руководстве — даже те, которые вводятся в интерактивном режиме — содержат комментарии. Комментарии в

Python начинаются с символа решетки # (hash) — и продолжаются до физического конца строки. Комментарии могут находиться как в начале строки, так и следовать за пробельными символами или кодом — но не содержаться внутри строки. Символ решетки в строке остается лишь символом решетки. Поскольку комментарии предназначены для того, чтобы сделать код более понятным, и не интерпретируются Python - при вводе примеров они могут быть опущены.

Несколько примеров:

```
# это первый комментарий
SPAM = 1                 # а это второй комментарий
# ... и наконец третий!
STRING = "# Это не комментарий."
```

4.1. Использование Python в качестве калькулятора

Давайте опробуем несколько простых команд Python. Запустите интерпретатор и дождитесь появления основного приглашения — >>>. (Это не должно занять много времени.)

4.1.1. Числа

Поведение интерпретатора сходно поведению калькулятора: вы вводите выражение, а в ответ он выводит значение. Синтаксис выражений привычен: операции +, -, * и / работают также как и в большинстве других языков (например, Паскале или C); для группировки можно использовать скобки. Например:

```
>>> 2+2
4
>>> # Это комментарий
... 2+2
4
>>> 2+2  # а вот комментарий на одной строке с кодом
4
>>> (50-5*6)/4
5.0
>>> 8/5 # При делении целых чисел дробные части не
теряются
1.6000000000000001
```

Заметьте: Вы можете получить результат, несколько отличный от представленного: результаты деления с плавающей запятой могут различаться на разных системах. Позже мы расскажем о том, как контролировать вывод чисел с плавающей запятой. Здесь использован наиболее ин-

формативный вариант вывода этого значения, а не более легко читаемый, какой возможен:

```
>>> print(8/5)
1.6
```

Чтобы пособие читалось легче, мы будем показывать упрощенный вывод чисел с плавающей точкой, и объясним позже, почему эти два способа отображения чисел с плавающей точкой стали различными. Обратитесь к приложению Арифметика с плавающей точкой: Проблемы и ограничения, чтобы ознакомиться с подробным обсуждением.

Для получения целого результата при делении целых чисел, отсекая дробные результаты, предназначена другая операция: `//`:

```
>>> # Деление целых чисел возвращает округленное к
минимальному значение:
... 7//3
2
>>> 7//-3
-3
```

Знак равенства (`'='`) используется для присваивания переменной какого-либо значения. После этого действия в интерактивном режиме ничего не выводится:

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

Значение может быть присвоено нескольким переменным одновременно:

```
>>> x = y = z = 0  # Нулевые x, y и z
>>> x
0
>>> y
0
>>> z
0
```

Переменные должны быть *определены* (`defined`) (должны иметь присвоенное значение) перед использованием, иначе будет сгенерирована ошибка:

```
>>> # попытка получить доступ к неопределенной пере-
менной
... n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Присутствует полная поддержка операций с плавающей точкой; операции над операндами смешанного типа конвертируют целочисленный операнд в число с плавающей запятой:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Поддерживаются и комплексные числа, добавлением к мнимым частям суффикса `j` или `J`. Комплексные числа с ненулевым вещественным компонентом записываются в виде (*<вещественная_часть>*+*<мнимая_часть>*`j`), или могут быть созданы с помощью функции `complex(`*<вещественная_часть>*`, `*<мнимая_часть>*`)`.

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0, 1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Комплексные числа всегда представлены в виде двух чисел с плавающей запятой — вещественной и мнимой частями. Для получения этих частей из комплексного числа `z` используется `z.real` и `z.imag` соответственно.

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

Функции конвертации (приведения) к вещественным и целым числам (`float()`, `int()`) не работают с комплексными числами, так как нет единственно правильного способа сконвертировать комплексное число в вещественное. Используйте функцию `abs(z)` чтобы получить *модуль* числа (в виде числа с плавающей точкой) или `z.real` чтобы получить его вещественную часть:

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: can`t convert complex to float; use
```

```
abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)   # sqrt(a.real**2 + a.imag**2)
5.0
>>>
```

В интерактивном режиме последнее выведенное выражение сохраняется в переменной _. Это значит, что если вы используете Python в качестве настольного калькулятора — всегда есть способ продолжить вычисления с меньшими усилиями, например:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
>>>
```

Эта переменная для пользователя должна иметь статус *только для чтения*. Не навязывайте ей значение собственноручно — вы создадите независимую переменную с таким же именем, скрывающую встроенную переменную вместе с ее магическими свойствами.

4.1.2. Строки

Помимо чисел, Python может работать со строками, которые, в свою очередь, могут быть описаны различными способами. Строки могут быть заключены в одинарные или двойные кавычки:

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

Интерпретатор выводит результаты операций над строками тем же способом, каким они были введены: обрамляя в кавычки, и, кроме того, экранируя обратными слэшами внутренние кавычки и другие забавные символы — для того, чтобы отобразить точное значение. Строка заключается в двойные кавычки, если строка содержит одинарные кавычки и ни одной двойной, иначе она заключается в одинарные кавычки. Повторимся, функция `print()` предоставляет более читаемый вывод.

Строковые литералы могут быть разнесены на несколько строк различными способами. Могут быть использованы *продолжающие строки*, с обратным слэшем в качестве последнего символа строки, сообщающим о том, что следующая строка есть продолжение текущей:

```
hello = "This is a rather long string containing\n\
 several lines of text just as you would do in C.\n\
Note that whitespace at the beginning of the line
is\
significant."
print(hello)
```

Обратите внимание, что новые строки все еще нужно подключать в строку через `\n`; новая строка, за которой следует обратный слэш — не обрабатывается. Запуск примера выведет следующее:

```
This is a rather long string containing
several lines of text just as you would do in C.
Note that whitespace at the beginning of the line is
significant.
```

Если мы объявим строковой литерал *сырым* (`raw`) — символы `\n` не будут конвертированы в новые строки, но и обратный слэш в конце строки, и символ новой строки в исходном коде — будут добавлены в строку в виде данных. Следовательно, код из примера:

```
hello = r"This is a rather long string containing\n\
several lines of text much as you would do in C."
print(hello)
```

выведет:

```
This is a rather long string containing\n\ several
lines of text much as you would do in C.
```

Или строки могут быть обрамлены совпадающей парой тройных кавычек: `"""` или `'''`. Окончания строк не нужно завершать тройными кавычками - при этом будут включены в строку.

```
print("""
Usage: thingy [OPTIONS]
-h                        Display this usage message
-H hostname               Hostname to connect to
""")
```

выводит в результате следующее:

```
Usage: thingy [OPTIONS]
-h                        Display this usage message
-H hostname               Hostname to connect to
```

Строки могут конкатенироваться (склеиваться вместе) операцией + и быть повторенными операцией *:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Два строковых литерала, расположенные друг за другом, автоматически конкатенируются; первая строка в предыдущем примере также могла быть записана как `word = 'Help' 'A'`; это работает только с двумя литералами — не с произвольными выражениями, содержащими строки.

```
>>> 'str' 'ing'                   #  <-  Так — верно
'string'
>>> 'str'.strip() + 'ing'   #  <-  Так — верно
'string'
>>> 'str'.strip() 'ing'     #  <-  Так — не верно
File "<stdin>", line 1, in ?
'str'.strip() 'ing'
^
SyntaxError: invalidsyntax
```

Строки могут быть проиндексированы; также как и в C, первый символ строки имеет индекс 0. Отсутствует отдельный тип для символов; символ является строкой с единичной длиной. Как и в языке программирования Icon, подстроки могут определены через нотацию срезов (`slice`): два индекса, разделенных двоеточием.

```
>>>word[4]
'A'
>>>word[0:2]
'He'
>>>word[2:4]
'lp'
```

Индексы срезов имеют полезные значения по умолчанию; опущенный первый индекс заменяется нулем, опущенный второй индекс подменяется размером срезаемой строки.

```
>>> word[:2]    # Первые два символа
'He'
>>> word[2:]    # Все, исключая первые два символа
'lpA'
```

В отличие от строк в C, строки Python не могут быть изменены. Присваивание по позиции индекса строки вызывает ошибку:

```
>>> word[0] = 'x'
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: 'str' object doesn't support item assign-
ment
>>> word[:1] = 'Splat'
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: 'str' object doesn't support slice as-
signment
```

Тем не менее, создание новой строки со смешанным содержимым — процесс легкий и очевидный:

```
>>> 'x' + word[1:]
'xelpA'
>>> 'Splat' + word[4]
'SplatA'
```

Полезный инвариант операции среза: `s[:i] + s[i:]` эквивалентно `s`.

```
>>> word[:2] + word[2:]
'HelpA'
>>> word[:3] + word[3:]
'HelpA'
```

Вырождения индексов срезов обрабатываются элегантно: чересчур большой индекс заменяется на размер строки, а верхняя граница, меньшая нижней, возвращает пустую строку.

```
>>> word[1:100]
'elpA'
>>> word[10:]
''
>>> word[2:1]
''
```

Индексы могут быть отрицательными числами, обозначая при этом отсчет справа налево. Например:

```
>>> word[-1]     # Последний символ
'A'
>>> word[-2]     # Предпоследний символ
'p'
>>> word[-2:]    # Последние два символа
'pA'
>>> word[:-2]    # Все, кроме последних двух символов
'Hel'
```

Но обратите внимание, что `-0` действительно эквивалентен `0` — это не отсчет справа.

```
>>> word[-0]     # (поскольку -0 равен 0)
'H'
```

Отрицательные индексы вне диапазона обрезаются, но не советуем делать это с одно-элементными индексами (*не-срезами*):

```
>>> word[-100:]
'HelpA'
>>> word[-10]    # ошибка
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

Один из способов запомнить, как работают срезы — думать о них, как об указателях на места между символами, где левый край первого символа установлен в ноль, а правый край последнего символа строки из `n` символов имеет индекс `n`, например:

```
+---+---+---+---+---+
 | H | e | l | p | A |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

Первый ряд чисел дает позицию индексов строки от 0 до 5; второй ряд описывает соответствующие отрицательные индексы. Срез от i до j состоит из всех символов между правым и левым краями, отмеченными, соответственно, через `i` и `j`.

Для всех индексов, больших или равных нулю, длина среза — разность между индексами, при условии что оба индекса находятся в диапазоне. Например, длина `word[1:3]` — 2.

Встроенная функция `len()` возвращает длину строки:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Примечание:

Перечисляемые типы

Строки — вид перечисляемых типов и они поддерживают привычные для этих типов операции.

Строковые методы

Строки поддерживают большое количество методов для поиска и простых трансформаций.

Форматирование строк

Здесь описано форматирование строк с применением функции `str.format()`

Операции форматирования строк в старом стиле

Операции форматирования, вызывающиеся тогда, когда обычные строки или строки в Unicode оказываются левым операндом относительно операции `%`, более детально рассмотрены здесь.

4.1.3. О Unicode

Начиная с Python версии 3.0, строковый тип поддерживает только Unicode (см. http://www.unicode.org/).

Преимущество набора Unicode состоит в том, что он предоставляет порядковый номер для любого символа из любой письменности, использовавшейся в современных или древнейших текстах. До этих пор для символов в сценарии было доступно лишь 256 номеров. Тексты обычно привязывались к кодовой странице, которая устанавливала в соответствие порядковые номера и символы сценария. Это приводило к серьезной путанице, особенно в том, что касалось интернационализации программного продукта. Unicode решает эти проблемы, определяя единую кодовую страницу для всех письменностей.

Для вставки в строку специального символа можно использовать Unicode-экранирование (`Python Unicode-Escape encoding`). Следующий пример все пояснит:

```
>>> 'Hello\u0020World !'
'HelloWorld !'
```

Экранированная последовательность `\u0020` задает символ Unicode с порядковым номером `0x0020` (символ пробела).

Другие символы интерпретируются с использованием соответствующих им порядковых значений тем же способом, что и порядковые номера Unicode. Первые 128 символов кодировки Unicode полностью совпадают с 128 символами кодировки `Latin-1`, использующейся во многих западных странах.

Помимо стандартных способов кодирования, Python предоставляет целый набор различных способов создания Unicode-строк, основываясь на известной кодировке.

Для конвертирования Unicode-строки в последовательность байтов с использованием желаемой кодировки, строковые объекты предоставляют метод `encode()`, принимающий единственный параметр — название кодировки. Предпочитаются названия кодировок, записанные в нижнем регистре.

```
>>>"Äpfel".encode('utf-8')
b'\xc3\x84pfel'
```

4.1.4. Списки

В языке Python доступно некоторое количество *составных* типов данных, использующихся для группировки прочих значений вместе. Наиболее гибкий из них — список (`list`). Его можно выразить в тексте программы

через разделенные запятыми значения (*элементы*), заключенные в квадратные скобки. Элементы списка могут быть разных типов.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Подобно индексам в строках, индексы списков начинаются с нуля, списки могут быть срезаны, объединены (*конкатенированы*) и так далее:

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam',
'eggs', 100, 'Boo!']
```

В отличие от строк, являющихся неизменяемыми, изменить индивидуальные элементы списка вполне возможно:

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Присваивание срезу также возможно, и это действие может даже изменить размер списка или полностью его очистить:

```
>>> # Заменим некоторые элементы:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Удалим немного:
... a[0:2] = []
>>>a
[123, 1234]
>>> # Вставимпару:
... a[1:1] = ['bletch', 'xyzzy']
>>>a
[123, 'bletch', 'xyzzy', 1234]
>>> # Вставим (копию) самого себя в начало
```

```
>>> a[:0] = a
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch',
'xyzzy', 1234]
>>> # Очистка списка: замена всех значений пустым
списком
>>> a[:] = []
>>> a
[]
```

Встроенная функция `len()` также применима к спискам:

```
>>> a = ['a', 'b', 'c', 'd']
>>> len(a)
4
```

Вы можете встраивать списки (создавать списки, содержащие другие списки), например так:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
```

Вы можете добавить что-нибудь в конец списка.

```
>>> p[1].append('xtra')
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Обратите внимание, что в последнем примере `p[1]` и `q` на самом деле ссылаются на один и тот же объект! Мы вернемся к *семантике объектов* позже.

4.2. Первые шаги к программированию

Безусловно, Python можно использовать для более сложных задач, чем сложение двух чисел. Например, мы можем вывести начало последовательности чисел Фибоначчи таким образом:

```
>>> # Ряд Фибоначчи:
... # сумма двух элементов определяет следущий эле-
мент
... a, b = 0, 1
>>> while b < 10:
```

```
...     print(b)
...     a, b = b, a+b
...
1
1
2
3
5
8
```

Этот пример показывает нам некоторые новые возможности.

• Первая строка содержит *множественное присваивание* (`multiple assignment`): переменные `a` и `b` параллельно получают новые значения — `0` и `1`. В последней строке этот метод используется снова, демонстрируя тот факт, что выражения по правую сторону [от оператора присваивания] всегда вычисляются раньше каких бы то ни было присваиваний. Сами же разделенные запятыми выражения вычисляются слева направо.

• Цикл `while` (пока) исполняется до тех пор, пока условие (здесь: `b < 10`) остается истиной. В Python, также как и в C, любое ненулевое значение является истиной (`True`); ноль является ложью (`False`). Условием может быть строка, список или вообще любая последовательность; все, что имеет ненулевую длину, играет роль истины, пустые последовательности — лжи. Использованная в примере проверка — простое условие. Стандартные операции сравнения записываются так же, как и в C: < (меньше чем), > (больше чем), == (равно), <= (меньше или равно), >= (больше или равно) и != (не равно).

• *Тело* цикла выделено *отступом* (`indented`). Отступы — это средство группировки операторов в Python. Интерактивный режим Python (пока!) не имеет какого-либо разумного и удобного средства для редактирования строк ввода, поэтому необходимо использовать табуляции или пробелы для отступа в каждой строке. На практике более сложный текст на Python готовится в текстовом редакторе, а большинство из них имеют функцию авто-отступа. По окончанию ввода составного выражения в интерактивном режиме, необходимо закончить его пустой строкой — признаком завершения (поскольку интерпретатор не может угадать, когда вами была введена последняя строка). Обратите внимание, что размер отступа в каждой строке основного блока должен быть одним и тем же.

• Функция `print()` выводит значения переданных ей выражений. Поведение этой функции отличается от обычного вывода выражения (как происходило выше в примерах с калькулятором) тем, каким способом обрабатываются ряды выражений, величины с плавающей точкой и строки. Строки выводятся без кавычек и между элементами вставляются пробелы, благодаря чему форматирование вывода улучшается — как, например, здесь:

```
>>> i = 256*256
>>> print('Значением i является', i)
Значением i является 65536
```

Для отключения перевода строки после вывода или завершения вывода другой строкой используется именованный параметр `end`:

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print(b, end=' ')
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

5. БОЛЬШЕ СРЕДСТВ ДЛЯ УПРАВЛЕНИЯ ПОТОКОМ КОМАНД

Помимо описанного выше оператора `while`, в Python доступны привычные операторы управления потоком из других языков, но с некоторыми особенностями.

5.1. Оператор `if`

Возможно, наиболее широко известный тип оператора — оператор `if` (если). Пример:

```
>>> x = int(input("Введите, пожалуйста, целое число:
"))
Введите, пожалуйста, целое число: 42
>>> if x < 0:
... x = 0
...print('Отрицательное значение, изменено на ноль')
...elifx == 0:
...print('Ноль')
...elifx == 1:
...print('Один')
...else:
...print('Больше')
...
```

Блока `elif` может не быть вообще, он может быть один или их может быть несколько, а блок `else` (иначе) необязателен. Ключевое слово `elif` — краткая запись `else if` (иначе если) — позволяет избавиться от чрезмерного количества отступов. Оператор `if...elif...elif...` — аналог оператора выбора `switch` или `case`, которые можно встретить в других языках программирования.

5.2.Оператор `for`

Оператор `for` в Python немного отличается от того, какой вы, возможно, использовали в C или Pascal. Вместо неизменного прохождения по арифметической прогрессии из чисел (как в Pascal) или предоставления пользователю возможности указать шаг итерации и условие остановки (как в C), оператор `for` в Python проходит по всем элементам любой последовательности (списка или строки) в том порядке, в котором они в ней располагаются. Например (игра слов не подразумевалась):

```
>>> # Измеримнесколькострок:
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
... print(x, len(x))
...
cat 3
window 6
defenestrate 12
```

Изменять содержимое последовательности, по которой проходит цикл, небезопасно (это в принципе-то возможно только для изменяемых типов, таких как списки). Если необходимо модифицировать список, использующийся для организации цикла, (например, для того чтобы продублировать отдельные элементы) нужно передать циклу его копию. Нотация срезов делает это практически безболезненным:

```
>>> for x in a[:]:  # создать срез-копию всего спи-
ска
...if len(x) > 6: a.insert(0, x)
...
>>> a
['defenestrate', 'cat', 'window', 'defenestrate']
```

5.3. Функция `range()`

Если вам нужно перебрать последовательность чисел, встроенная функция `range()` придет на помощь. Она генерирует арифметические прогрессии:

```
>>> for i in range(5):
... print(i)
...
0
1
2
3
4
```

Указанный конец интервала никогда не включается в сгенерированный список; вызов `range(10)` генерирует десять значений, которые являются подходящими индексами для элементов последовательности длины 10. Можно указать другое начало интервала и другую, даже отрицательную, величину шага.

```
range(5, 10)
от 5 до 9
range(0, 10, 3)
0, 3, 6, 9
range(-10, -100, -30)
-10, -40, -70
```

Чтобы пройти по всем индексам какой-либо последовательности, скомбинируйте вызовы `range()` и `len()` следующим образом:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

В большинстве таких случаев удобно использовать функцию `enumerate()`, обратитесь к Организации циклов.

Странные вещи начинают происходить при попытке вывода последовательности:

```
>>> print(range(10))
range(0, 10)
```

Во многих случаях объект, возвращаемый функцией `range()`, ведет себя как список, но фактически им не является. Этот объект возвращает по очереди элементы желаемой последовательности, когда вы проходите по нему в цикле, но на самом деле не создает списка, сохраняя таким образом пространство в памяти.

Мы называем такие объекты *итерируемыми* (`iterable`), и это все объекты, которые предназначаются для функций и конструкций, ожидающих от них поочередного предоставления элементов до тех пор, пока источник не иссякнет. Мы видели, что оператор `for` является таким *итератором*`iterator`. Функция `list()` тоже из их числа — она создает списки из *итерируемых* объектов:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

Позже мы познакомимся и с другими функциями, которые возвращают и принимают *итерируемые* объекты в качестве параметров.

5.4. Операторы `break` и `continue`, а также условие `else` в циклах

Оператор `break` прерывает выполнение самого ближайшего вложенного цикла `for` или `while` (по аналогии с языком C).

Оператор `continue`, также заимствованный из C, продолжает выполнение цикла со следующей итерации.

Операторы циклов могут иметь ветвь `else`. Она исполняется, когда цикл выполнил перебор до конца (в случае `for`) или когда условие становится ложным (в случае `while`), но не в тех случаях, когда цикл прерывается по `break`. Это поведение иллюстрируется следующим примером, в котором производится поиск простых чисел:

```
>>>for n in range(2, 10):
...for x in range(2, int(n ** 0.5) + 1):
...if n % x == 0:
... print(n, 'равно', x, '*', n//x)
... break
... else:
... print(n, '- простоечисло')
...
2 - простое число
3 - простое число
4 равно 2 * 2
5 - простое число
6 равно 2 * 3
7 - простое число
8 равно 2 * 4
9 равно 3 * 3
```

5.5. Оператор `pass`

Оператор `pass` не делает ничего. Он может использоваться когда синтаксически требуется присутствие оператора, но от программы не требуется действий. Например:

```
>>> while True:
...pass # Ожидание прерывания с клавиатуры (Ctrl+C)
в режиме занятости
...
```

Этот оператор часто используется для создания минималистичных классов, к примеру *исключений* (`exceptions`), или для игнорирования нежелательных исключений:

```
>>> class ParserError(Exception):
...pass
...
>>> try:
... import audioop
... except ImportError:
... pass
...
```

Другой вариант: `pass` может применяться в качестве заглушки для тела функции или условия при создании нового кода, позволяя вам сохранить абстрактный взгляд на вещи. С другой стороны, оператор `pass` игнорируется без каких-либо сигналов и лучшим выбором было бы породить исключение `NotImplementedError`:

```
>>> def initlog(*args):
...  raise NotImplementedError    # Открыть файл для
логгинга, если он еще не открыт
... if not logfp:
...  raise NotImplementedError   # Настроить заглушку
для логгинга
...  raise NotImplementedError('Обработчик инициали-
зации логавызовов')
...
```

Если бы здесь использовались операторы `pass`, а позже вы бы запускали тесты, они могли бы упасть без указания причины. Использование `NotImplementedError` принуждает этот код породить исключение, сообщая вам конкретное место, где присутствует незавершенный код. Обратите внимание на два способа порождения исключений. Первый способ, без сообщения, но сопровождаемый комментарием, позволяет вам оставить комментарий когда вы будете подменять выброс исключения рабочим кодом, который, в свою очередь, в идеале, будет хорошим описанием блока кода, для которого исключение предназначалось заглушкой. Однако, передача сообщения вместе с исключением, как в третьем примере, обуславливает более насыщенный информацией вывод при отслеживании ошибки.

5.6. Определение функций

Мы можем создать функцию, которая выводит числа Фибоначчи до некоторого предела:

```
>>> def fib(n):    # вывести числа Фибоначчи меньшие
(вплоть до) n
... """Выводит ряд Фибоначчи, ограниченный n."""
... a, b = 0, 1
... while b < n:
... print(b, end=' ')
... a, b = b, a+b
...
>>> # Теперь вызовем определенную нами функцию:
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Зарезервированное слово `def` предваряет *определение* функции. За ним должны следовать имя функции и заключенный в скобки список формальных параметров. Выражения, формирующие тело функции, начинаются со следующей строки и должны иметь отступ.

Первым выражением в теле функции может быть строковой литерал — этот литерал является строкой документации функции, или *док-строкой* (`docstring`). (Больше информации о *док-строках* вы найдете в разделе Строки документации) Существуют инструменты, которые используют *док-строки* для того, чтобы сгенерировать печатную или онлайн-документацию или чтобы позволить пользователю перемещаться по коду интерактивно; добавление строк документации в ваш код - это хорошая практика, постарайтесь к ней привыкнуть.

Исполнение функции приводит к созданию новой таблицы символов, использующейся для хранения локальных переменных функции. Если быть более точными, все присваивания переменных в функции сохраняют значение в локальной таблице символов; при обнаружении ссылки на переменную, в первую очередь просматривается локальная таблица символов, затем локальная таблица символов для окружающих функций, затем глобальная таблица символов и, наконец, таблица встроенных имен. Таким образом, глобальным переменным невозможно прямо присвоить значения внутри функций (если они конечно не упомянуты в операторе `global`) несмотря на то, что ссылки на них могут использоваться.

Фактические параметры при вызове функции помещаются в локальную таблицу символов вызванной функции; в результате аргументы передаются через *вызов по значению* (`call by value`) (где значение — это всегда *ссылка*(`reference`) на объект, а не значение его самого). Если одна функция вызывает другую — то для этого вызова создается новая локальная таблица символов.

При определении функции ее имя также помещается в текущую таблицу символов. Тип значения, связанного с именем функции, распознается интерпретатором как функция, определенная пользователем (`user-`

defined function). Само значение может быть присвоено другому имени, которое затем может также использоваться в качестве функции. Эта система работает в виде основного механизма переименования:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

Если вы использовали в работе другие языки программирования, вы можете возразить, что fib — это не функция, а процедура, поскольку не возвращает никакого значения. На самом деле, даже функции без ключевого слова return возвращают значение, хотя и более скучное. Такое значение именуется None (это встроенное имя). Вывод значения None обычно подавляется в интерактивном режиме интерпретатора, если оно оказывается единственным значением, которое нужно вывести. Вы можете проследить за этим процессом, если действительно хотите, используя функцию print():

```
>>> fib(0)
>>> print(fib(0))
None
```

Довольно легко написать функцию, которая возвращает список чисел из ряда Фибоначчи, вместо того, чтобы выводить их:

```
>>> def fib2(n): # вернуть числа Фибоначчи меньшие
(вплоть до) n
... """Возвращает список чисел ряда Фибоначчи, огра-
ниченный n."""
... result = []
... a, b = 0, 1
... while b < n:
... result.append(b)    # см. ниже
... a, b = b, a+b
... returnresult
...
>>> f100 = fib2(100)    # вызываем
>>> f100                # выводим результат
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

И на этот раз пример демонстрирует некоторые новые возможности Python:

- Оператор return завершает выполнение функции, возвращая некоторое значение. Оператор return без аргумента возвращает None. Достижение конца функции также возвращает None.

• Выражение `result.append(b)` вызывает метод `append` объекта-списка `result`. Метод — это функция, которая "принадлежит" объекту и указывается через выражение вида `obj.methodname`, где `obj` — некоторый объект (может быть выражением), а `methodname` — имя метода, присущий объекту данного типа. Различные типы определяют различные методы. Методы разных типов могут иметь одинаковые имена, не вызывая неопределенностей. (Позже в этом учебнике будет рассмотрено как определять собственные типы объектов и методы, используя классы.) Метод `append()`, показанный в примере, определен для объектов типа список. Он добавляет в конец списка новый элемент. В данном примере это действие эквивалентно выражению `result = result + [b]`, но более эффективно.

5.7. Подробнее об определении функций

Также есть возможность определять функции с переменным количеством параметров. Для этого существует три формы, которые также можно использовать совместно.

5.7.1. Значения аргументов по умолчанию

Наиболее полезной формой является задание значений по умолчанию для одного или более параметров. Таким образом создается функция, которая может быть вызвана с меньшим количеством параметров, чем в ее определении: при этом неуказанные при вызове параметры примут данные в определении функции значения. Например:

```
def ask_ok(prompt, retries=4, complaint='Yes or no,
please!'):
while True:
ok = input(prompt)
if ok in ('y', 'yeah', 'yes', 'yep'): return True
if ok in ('n', 'no', 'nop', 'nope'): return False
retries = retries - 1
if retries < 0:
raise IOError('refusenik user')
print(complaint)
```

Эта функция может быть вызвана, например, так: `ask_ok('Do you really want to quit?')` или так: `ask_ok('OK to overwrite the file?', 2)`.

Этот пример также знакомит вас с зарезервированным словом `in`. Посредством его можно проверить, содержит ли последовательность определенное значение или нет.

Значения по умолчанию вычисляются в месте определения функции, в *определяющей* области, поэтому следующий код выведет 5.

```
i = 5

def f(arg=i):
print(arg)
i = 6
f()
```

Важное предупреждение! Значение по умолчанию вычисляется лишь единожды. Это особенно важно помнить, когда значением по умолчанию является изменяемый объект, такой как список, словарь (dictionary) или экземпляры большинства классов. Например, следующая функция накапливает переданные ей параметры:

```
def f(a, L=[]):
L.append(a)
return L
print(f(1))
print(f(2))
print(f(3))
```

Она выведет

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

Если вы не хотите, чтобы значение по умолчанию распределялось между последовательными вызовами, вместо предыдущего варианта вы можете использовать такую идиому:

```
def f(a, L=None):
if L is None:
L = []
L.append(a)
return L
```

5.7.2. Именованные параметры

Функции также могут быть вызваны с использованием именованных параметров (keyword arguments) в форме "*имя* = *значение*". Например, ниже приведенная функция:

```
def parrot(voltage, state='a stiff', action='voom',
type='Norwegian Blue'):
print("-- This parrot wouldn't", action, end=' ')
print("if you put", voltage, "volts through it.")
print("-- Lovely plumage, the", type)
print("-- It's", state, "!")
```

могла бы быть вызвана любым из следующих способов:

```
parrot(1000)
```

```
parrot(action='VOOOOOM', voltage=1000000)
parrot('a thousand', state='pushing up the daisies')
parrot('a million', 'bereft of life', 'jump')
```
а эти случаи оказались бы неверными:
```
parrot()                          # пропущен требуемый
аргумент
parrot(voltage=5.0, 'dead')   # позиционный параметр
вслед за именованным
parrot(110, voltage=220)        # повторное значение
параметра
parrot(actor='John Cleese')   # неизвестное имя пара-
метра
```

В общем случае список параметров должен содержать любое количество позиционных (`positional`) параметров, за которыми может следовать любое количество именованных, и при этом имена аргументов выбираются из формальных параметров. Неважно, имеет формальный параметр значение по умолчанию или нет. Ни один из аргументов не может получать значение более чем один раз — имена формальных параметров, совпадающие с именами позиционных параметров, не могут использоваться в качестве именующих в одном и том же вызове. Вот пример, завершающийся неудачей по причине описанного ограничения:

```
>>> def function(a):
... pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function() got multiple values for key-
word argument 'a'
```

Если в определении функции присутствует завершающий параметр в виде ***имя*, он получит в качестве значения словарь (подробнее в разделе Справочника — Типы-отображения — словари), содержащий все именованные параметры и их значения, исключая те, которые соответствуют формальным параметрам. Можно совместить эту особенность с поддержкой формального параметра в формате **имя* (описывается в следующем подразделе), который получает кортеж (`tuple`), содержащий все позиционные параметры, следующие за списком формальных параметров. (параметр в формате **имя* должен описываться перед параметром в формате ***имя*.) Например, если мы определим такую функцию:

```
def cheeseshop(kind, *arguments, **keywords):
print("-- Do you have any", kind, "?")
print("-- I'm sorry, we're all out of", kind)
```

```
for arg in arguments: print(arg)
print("-" * 40)
keys = sorted(keywords.keys())
for kw in keys: print(kw, ":", keywords[kw])
```

то ее можно будет вызвать так:

```
cheeseshop('Limburger', "It's very runny, sir.",
"It's really very, VERY runny, sir.",
client="John Cleese",
shopkeeper="Michael Palin",
sketch="Cheese Shop Sketch")
```

и она, конечно же, выведет:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
client: John Cleese
shopkeeper: Michael Palin
sketch: CheeseShopSketch
```

Обратите внимание, что список имен (ключей) именованных параметров (`keys`) создается посредством сортировки содержимого списка ключей `keys()` словаря `keywords`; если бы этого не было сделано, порядок вывода параметров был бы произволен.

5.7.3. Списки параметров произвольной длины

Наконец, наиболее редко используется возможность указания того, что функция может быть вызвана с произвольным числом аргументов. При этом сами параметры будут обернуты в кортеж (см. раздел Кортежи). Переменное количество параметров могут предварять ноль или более обычных.

```
def write_multiple_items(file, separator, *args):
file.write(separator.join(args))
```

Обычно параметры неизвестного заранее количества (`variadic`) указываются последними в списке формальных параметров, поскольку включают в себя все остальные переданные в функцию параметры. Все формальные параметры, которые следуют за параметром `*args`, должны быть только именованными, то есть, они могут быть заданы только по имени (в отличие от позиционных параметров).

```
>>> def concat(*args, sep="/"):
... return sep.join(args)
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

5.7.4. Распаковка списков параметров

Обратная ситуация возникает когда параметры уже содержатся в списке или в кортеже, но должны быть распакованы для вызова функции, требующей отдельных позиционных параметров. Например, встроенная функция `range()` ожидает отдельные параметры *start* и *stop* соответственно. Если они не доступны раздельно, для распаковки аргументов из списка или кортежа в вызове функции используйте *-синтаксис:

```
>>> list(range(3, 6))                # обычный вызов с
отдельными параметрами
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))               # вызов с распако-
ванными из списка параметрами
[3, 4, 5]
```

Схожим способом, словари могут получать именованные параметры через **-синтаксис:

```
>>> def parrot(voltage, state='a stiff', ac-
tion='voom'):
... print("-- This parrot wouldn't", action, end='
')
... print("if you put", voltage, "volts through
it.", end=' ')
... print("It's", state, "!")
...
>>> d = {"voltage": "four million", "state":
"bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million
volts through it. It's bleedin' demised !
```

5.7.5. Модель `lambda`

В связи с неустанными просьбами, в Python были добавлены несколько возможностей, которые были привычны для функциональных языков программирования, таких как Lisp. Используя зарезервированное слово `lambda`, вы можете создать небольшую безымянную функцию. Например, функцию, которая возвращает сумму двух своих аргументов, можно записать так: `lambda a, b: a+b`. Формы `lambda` могут быть использованы в любом месте где требуется объект функции. При этом они синтаксически ограничены одним выражением. Семантически, они лишь «синтаксический сахар» для обычного определения функции. Как и определения вложенных функций, `lambda`-формы могут ссылаться на переменные из содержащей их области видимости:

```
>>> def make_incrementor(n):
...  return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

5.7.6. Строки документации

Перечислим некоторые существующие соглашения по содержимому строк документации и их форматированию.

По поводу форматирования строк документации и их содержимого постоянно появляются все новые соглашения.

Первая строка всегда должна быть сжатой, лаконичной сводкой о назначении объекта. Для краткости, в ней не обязательно присутствие имени типа или объекта, поскольку они доступны другими способами (исключая случай, когда имя функции оказывается глаголом, описывающим суть операции). Эта строка должна начинаться с прописной буквы и оканчиваться точкой.

Если строке документации (литералу, объекту строки) требуется больше строк (физических), вторая строка должна быть пустой, визуально отделяя сводку от остального описания. Следующие строки могут быть одним или более абзацем, описывающим соглашения по вызову объекта, сторонние эффекты, и т.д.

Парсер Python не обрабатывает отступы в многостроковых литералах, поэтому инструментам, которые работают над документацией, предлагается, по желанию, делать это самим. Производится это по следующему соглашению. Первая непустая строка после первой строки литерала определяет величину отступа всего литерала документации. (Мы не можем использовать первую строку, поскольку она обычно выравнивается по открывающим кавычкам и ее отступ в литерале не явен). Пробельный „эквивалент“ этого отступа затем отрезается от начала всех строк литерала. Строк с меньшим отступом не должно обнаруживаться, но если они встретились, весь их начальный отступ должен быть обрезан. Эквивалентность пробельных замен может быть протестирована развертыванием табуляции (обычно, к 8-ми пробелам).

Вот пример многострочной документации (*док-строки*):

```
>>> def my_function():
...  """Не делаем ничего, но документируем.
...
...  Нет, правда, эта функция ничего не делает.
```

```
... """
... pass
...
>>>print(my_function.__doc__)
Не делаем ничего, но документируем.
    Нет, правда, эта функция ничего не делает.
```

5.8. Интермеццо: Стиль написания кода

Теперь, когда вам предстоит писать более объемные и сложные блоки кода на Python, настало время поговорить о *стиле написания кода* (`coding style`). Код на большинстве языков программирования может быть записан (или, точнее говоря, *отформатирован* (`formatted`)) различными способами; некоторые из них более читабельны, некоторые — нет. Стремление к написанию легкого для прочтения другими кода всегда считалось хорошим тоном, и выбор правильного стиля для кода крайне ему способствует.

В случае языка Python, в качестве руководства по стилю было создано предложение PEP8, которого придерживаются создатели большинства проектов. В нем учреждается чрезвычайно читабельный и приятный для глаза стиль написания кода. В некоторый момент с ним должен ознакомиться каждый разработчик на Python. Приведем здесь избранные, наиболее важные, пункты:

- Используйте отступ в 4 пробела, не используйте табуляцию.

Четыре пробела легко опознаются и в случае небольших отступов (хватает места для глубоких вложений) и в случае больших отступов (приятнее читается). Табуляция вносит путаницу и лучше от нее воздержаться.

- Разделяйте строки так, чтобы их длина не превышала 79-и символов

Это поможет пользователям с небольшими экранами, а пользователям с большими экранами позволит уложить несколько файлов с исходным кодом рядом.

- Используйте пустые строки для отделения функций, классов, и крупных блоков внутри функций.
- При возможности располагайте комментарий на отдельной строке.
- Используйте строки документации (*док-строки*).
- Применяйте пробелы вокруг символов операций и после запятых, но не добавляйте их в конструкции со скобками: `a = f(1, 2) + g(3, 4)`
- Называйте ваши классы и функции единообразно; соглашение следующее: используйте `CamelCase` для именования классов и `нижний_регистр_с_подчеркиваниями` для функций и методов. (обращайтесь к разделу Первый взгляд на классы за дополнительной информации о классах и методах).

• Не используйте в вашем коде изощренных кодировок, если он рассчитан на использование в интернациональной среде. Стандартный набор ASCII всегда работает на ура.

6. СТРУКТУРЫ ДАННЫХ

Эта глава описывает подробнее некоторые вещи, которые вы уже изучили, а также раскрывает некоторые новые темы.

6.1. Подробнее о списках

У типа данных список также имеются не описанные ранее методы. Ниже приведены все методы объекта типа список:

```
list.append(x)
```

Добавить элемент к концу списка; эквивалент `list[len(list):] = [x]`

```
list.extend(L)
```

Расширить список за счет добавления всех элементов переданного списка; эквивалентно `list[len(list):] = L`.

```
list.insert(i, x)
```

Вставить элемент в указанную позицию. Первый аргумент — это индекс того элемента, перед которым требуется выполнить операцию вставки, поэтому вызов `list.insert(0, x)` вставляет элемент в начало списка, а `list.insert(len(list), x)` эквивалентно `list.append(x)`.

```
list.remove(x)
```

Удалить первый найденный элемент из списка, значение которого — `x`. Если элемент не найден, генерируется ошибка.

```
list.pop([i])
```

Удалить элемент, находящийся на указанной позиции в списке, и вернуть его. Если индекс не указан, `list.pop()` удаляет и возвращает последний элемент списка. (Квадратные скобки вокруг `i` в сигнатуре метода означают, что параметр необязателен, а не необходимость набора квадратных скобок в этой позиции. Вы часто будете встречать такую нотацию в Справочнике по библиотеке.)

```
list.index(x)
```

Вернуть индекс первого найденного в списке элемента, значение которого равно `x`. Если элемент не найден, генерируется ошибка.

```
list.count(x)
```

Вернуть значение сколько раз, `x` встречается в списке.

```
list.sort()
```

Сортировать элементы списка, на месте.

```
list.reverse()
```

Обратить порядок элементов списка, на месте.

Пример, использующий большинство методов списка:

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>>        print(a.count(333),        a.count(66.25),
a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>>a
[-1, 1, 66.25, 333, 333, 1234.5]
```

Использование списка в качестве стека

Методы списков позволяют легко использовать список в качестве стека, где последний добавленный элемент становится первым полученным („первый вошел - последний вышел“). Чтобы положить элемент на вершину стека, используйте метод `append()`. Для получения элемента с вершины стека - метод `pop()` без указания явного индекса. Например:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
```

```
>>> stack
[3, 4]
```

Использование списка в качестве очереди

Вы можете без труда использовать список также и в качестве очереди, где первый добавленный элемент оказывается первым полученным („первый вошел - первый вышел“). Чтобы добавить элемент в конец очереди, используйте метод `append()`, а чтобы получить элемент из начала очереди - метод `pop()` с нулем в качестве индекса. Например:

```
>>> queue = ["Eric", "John", "Michael"]
>>> queue.append("Terry")           # Прибыл Terry
>>> queue.append("Graham")          # Прибыл Graham
>>> queue.pop(0)
'Eric'
>>> queue.pop(0)
'John'
>>> queue
['Michael', 'Terry', 'Graham']
```

Генераторы списков

Использование метода списковой сборки — легкий способ создать список на основе последовательности. В большинстве случаев он применяется для создания списков, в которых каждый элемент является результатом некой операции, произведенной над каждым членом последовательности, или для создания выборок из тех элементов, которые удовлетворяют определенному условию.

Любая списковая сборка состоит из выражения, за которым следует оператор `for`, а затем ноль или более операторов `for` или `if`. Результатом станет список, получившийся через вычисление выражения в контексте следующих за ним операторов `for` и/или `if`. Если в результате вычисления выражения строится кортеж, его нужно явно обернуть в скобки.

В следующем примере на основе списка чисел создается список, где каждое число утроено:

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
```

Применим фантазию:

```
>>> [[x, x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
```

Здесь мы вызываем метод по очереди с каждым элементом последовательности:

```
>>> freshfruit = ['   banana', '   loganberry ',
```

```
'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
```

Используя оператор `if`, мы можем отфильтровать поток:

```
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
```

Кортежи могут быть созданы без использования скобок, но не в этом случае:

```
>>> [x, x**2 for x in vec]   # ошибка - для кортежей
необходимы скобки
File "<stdin>", line 1, in ?
 [x, x**2 for x in vec]
^
SyntaxError: invalid syntax
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
```

Вот несколько вложенных циклов `for` и еще кое-какое забавное поведение:

```
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

Списковые сборки могут применяться в сложных выражениях и вложенных функциях:

```
>>> [str(round(355/113, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

Вложенные списковые сборки

Если вы в состоянии это переварить: списковые сборки могут быть вложенными. Но как любой мощный инструмент, их следует использовать с осторожностью.

Представьте нижеследующий пример матрицы 3x3 в виде списка, содержащего три других списка, по одному в ряд:

```
>>> mat = [
... [1, 2, 3],
... [4, 5, 6],
```

```
... [7, 8, 9],
...]
```

Чтобы поменять строки и столбцы местами, можно использовать такую списковую сборку:

```
>>> print([[row[i] for row in mat] for i in [0, 1, 2]])
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Применяя вложенные списковые сборки, необходимо помнить о важной вещи:

Во избежание недоразумений при конструировании вложенных списковых сборок, читайте их справа налево.

Более многословная, с использованием операторов, версия примера может служить иллюстрацией:

```
for i in [0, 1, 2]:
for row in mat:
print(row[i], end="")
print()
```

В реальных случаях лучше использовать встроенные функции вместо сложных выражений. В нашем случае поможет функция `zip()`:

```
>>> list(zip(*mat))
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

В разделе Распаковка списков параметров описано предназначение звездочки в этих строках.

Другой пример использования вложенных списковых сборок - произведение матрицы a на матрицу b:

```
c=[[sum(x*y for x,y in zip(i,j)) for j in zip(*b)]
for i in a]
```

Это можно сделать эффективнее и прозрачнее, но на данном примере видна мощь инструмента.

Получение единичной матрицы порядка n:

```
a=[[0]*i+[1]+[0]*(n-i-1) for i in range(n)]
```

6.2. Оператор `del`

Существует способ удалить элемент, указывая его индекс, а не его значение: оператор `del`. В отличие от метода `pop()`, он не возвращает значения. Оператор `del` может также использоваться для удаления срезов из списка или полной очистки списка (что мы делали ранее через присваивание пустого списка срезу). Например:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
```

```
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>>a
[]
```

`del` может быть также использован для удаления переменных полностью:

```
>>> del a
```

Ссылка на имя `a` в дальнейшем вызовет ошибку (по крайней мере до тех пор, пока с ним не будет связано другое значение). Позже мы с вами узнаем другие способы использования `del`.

6.3. Кортежи и последовательности

Мы видели, что списки и строки поддерживают много привычных свойств, таких как индексирование и операция получения срезов. Существует два подвида типов данных *последовательность* (`sequence`) (см. Справочник по библиотеке — Последовательности), и поскольку Python — развивающийся язык, со временем могут быть добавлены другие последовательные типы данных. Итак, существует также и другой, достойный рассмотрения, стандартный последовательный тип данных: *кортеж*.

Кортеж состоит из некоторого числа значений разделенных запятыми, например:

```
>>>t = 12345, 54321, 'hello!'
>>>t[0]
12345
>>>t
(12345, 54321, 'hello!')
>>> # Кортежи могут быть вложенными:
... u = t, (1, 2, 3, 4, 5)
>>>u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

Как видите, кортежи на выводе всегда заключены в скобки, таким образом вложенные кортежи интерпретируются корректно; они могут быть введены и с обрамляющими скобками и без, тем не менее в любом случае скобки чаще всего необходимы (если кортеж — часть более крупного выражения).

Кортежи можно использовать в различных целях. Например: `(x, y)` пары координат, записи о рабочих из базы данных, и так далее. Кортежи, как и строки, неизменяемы: невозможно присвоить что-либо индивидуаль-

ным элементам кортежа (однако, вы можете симулировать большинство схожих эффектов за счет операций срезов и конкатенации). Также можно создать кортежи, содержащие изменяемые объекты, такие как списки.

Определенная проблема состоит в конструировании кортежей, состоящих из нуля или одного элемента: в синтаксисе языка есть дополнительные хитрости, позволяющие достигнуть этого. Пустые кортежи формируются за счет пустой пары скобок; кортеж с одним элементом конструируется за счет запятой, следующей за числом (единственное значение не обязательно заключать в скобки). Необычно, но эффективно. Например:

```
>>> empty = ()
>>> singleton = 'hello',    # <-- обратите внимание
на замыкающую запятую
>>>len(empty)
0
>>>len(singleton)
1
>>>singleton
('hello',)
```

Выражение `t = 12345, 54321, 'hello!'` является примером *упаковки кортежей* (`tuple packing`): значения `12345`, `54321` и `'hello!'` упаковываются в кортеж вместе. Обратная операция также возможна:

```
>>> x, y, z = t
```

Такое действие называется, довольно удачно, *распаковкой последовательности* (`sequence unpacking`). Для распаковки на левой стороне требуется список переменных с количеством элементов равным длине последовательности. Обратите внимание, что множественное присваивание на самом деле является лишь комбинацией упаковки кортежа и распаковки последовательности.

Здесь есть некоторая асимметрия: упаковка нескольких значений всегда создает кортеж, а распаковка работает для любой последовательности.

6.4. Множества

Python имеет также тип данных *множество*. Множество — это неупорядоченная коллекция без дублирующихся элементов. Основные способы использования — проверка на вхождение и устранение дублирующихся элементов. Объекты этого типа поддерживают обычные математические операции над множествами, такие как объединение, пересечение, разность и симметрическая разность.

Для создания множеств могут быть использованы фигурные скобки или функция `set()`. *Заметьте:* Для создания пустого множества нужно

использовать `set()`, а не `{}`: в последнем случае создается пустой *словарь* (`dictionary`) — тип данных, который мы обсудим в следующем разделе.

Продемонстрируем работу с множествами на небольшом примере:

```
>>> basket = {'apple', 'orange', 'apple', 'pear',
'orange', 'banana'}
>>> print(basket)
{'orange', 'banana', 'pear', 'apple'}
>>>  fruit_list  =  ['apple',  'orange',  'apple',
'pear', 'orange', 'banana']
>>> fruit = set(fruit_list)                # создать
множество на основе данных из списка (заметьте ис-
чезновение дубликатов -перев.)
>>> fruit
{'orange', 'pear', 'apple', 'banana'}
>>> fruit = {'orange', 'apple'}        # синтаксис {}
эквивалентен [] усписков
>>> fruit
{'orange', 'apple'}
>>> 'orange' in fruit                  # быстрая про-
верка на вхождение
True
>>> 'crabgrass' in fruit
False
>>> # Демонстрация операций со множествами на приме-
ре букв из двух слов
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                    # уникальные
буквы в a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                                 # буквы в a
но не в b
{'r', 'd', 'b'}
>>> a | b                                 # все буквы,
которые встречаются в a или в b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                                 # буквы, ко-
торые есть и в a и в b
{'a', 'c'}
>>> a ^ b                                  # буквы в a
```

```
или в b, но не в обоих
{'r', 'd', 'b', 'm', 'z', 'l'}
```
Как и у списков, у множеств существует синтаксис сборок:
```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

6.5. Словари

Другой полезный встроенный в Python тип данных — это *словарь* (`dictionary`) (см. Справочник по библиотеке — Связывающие типы). Словари иногда встречаются в других языках в виде «ассоциативных записей» или «ассоциативных массивов». В отличие от последовательностей, которые индексируются по диапазону чисел, словари индексируются по *ключам* (`keys`), которые, в свою очередь, могут быть любого неизменяемого типа; строки и числа всегда могут быть ключами. Кортежи могут быть ключами только если они составлены из строк, чисел или кортежей; если кортеж содержит какой-либо изменяемый объект, явно или неявно, то он не может быть использован в качестве ключа. Вы не можете использовать списки в роли ключей, поскольку списки могут быть изменены на месте присваиванием по индексу, присваиванием по срезу или такими методами как `append()` и `extend()`.

Лучше всего воспринимать словарь как неупорядоченный набор пар *ключ: значение* с требованием, чтобы ключи были уникальны (в пределах одного словаря). Пара скобок создает пустой словарь: `{}`. Указывая разделенный запятыми список пар *ключ: значение* внутри скобок, вы задаете содержимое словаря; в таком же формате словарь можно вывести.

Главные операции над словарем — это сохранение значения с каким-либо ключом и извлечение значения по указанному ключу. Также возможно удалить пару *ключ: значение* используя оператор `del`. Если вы сохраняете значение используя ключ, который уже встречается в словаре — старое значение, ассоциированное с этим ключом, стирается. Извлечение значения по несуществующему ключу вызывает ошибку.

Выполнение конструкции `list(d.keys())` с объектом словаря возвращает список всех ключей, использующихся в словаре, в случайном порядке (если вы хотите отсортировать его, к списку ключей можно применить функцию `sorted()`). Чтобы проверить, содержит ли словарь определенный ключ, используйте ключевое слово `in`.

Вот небольшой пример использования словарей:
```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
```

```
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> list(tel.keys())
['irv', 'guido', 'jack']
>>> sorted(tel.keys())
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

Конструктор `dict()` строит словарь непосредственно на основе пар ключей и значений, где каждая пара представлена в виде кортежа. Когда пары могут быть сформированы шаблоном, *списковые сборки* помогут описать список пар более компактно.

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack',
4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

В дополнение ко всему этому, для создания словарей из произвольных выражений для ключей и значений, могут быть использованы словарные сборки:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Позже в учебнике мы изучим выражения-генераторы (`Generator Expressions`), которые даже лучше подходят для снабжения конструктора `dict()` парами ключ-значение.

Если ключи являются простыми строками, иногда легче описать пары используя именованные параметры:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

6.6.Организация циклов

При организации перебора элементов из словаря ключ и соответствующее ему значение могут быть получены одновременно посредством метода `items()`.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the
brave'}
```

```
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robinthebrave
```

Функция `enumerate()` поможет пронумеровать элементы перебираемой в цикле последовательности:

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

Для того чтобы организовать цикл параллельно для двух или более последовательностей, элементы можно предварительно сгруппировать функцией `zip()`.

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
... print('What is your {0}?  It is {1}.'.format(q,
a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

Изменить порядок следования последовательности на обратный поможет функция `reversed()`.

```
>>> for i in reversed(range(1, 10, 2)):
... print(i)
...
9
7
5
3
1
```

Для организации цикла по отсортированной последовательности можно применить функцию `sorted()`, которая возвращает отсортированный список, оставляя исходный без изменений.

```
>>> basket = ['apple', 'orange', 'apple', 'pear',
'orange', 'banana']
>>> for f in sorted(set(basket)):
... print(f)
```

```
...
apple
banana
orange
pear
```

6.7. Подробнее об условиях

Условия в операторах `if` и `while` могут содержать любые операции, а не только операции сравнения.

Операции сравнения `in` и `not in` проверяют, встречается значение в последовательности или нет. Операции `is` и `is not` проверяют, не являются ли два объекта на самом деле одним и тем же (это имеет смысл лишь для изменяемых объектов, таких как списки). Все операции сравнения имеют один и тот же приоритет, меньший чем у любых операций над числами.

Сравнения можно объединять в цепочки. Например, `a < b == c` проверяет, меньше ли `a` чем `b` и, сверх того, равны ли `b` и `c`.

Сравнения могут быть скомбинированы с использованием булевых операций `and` и `or`, а результат сравнения (или любого другого булева выражения) можно отрицать используя `not`. Эти операции имеют меньший приоритет, чем у операций сравнения; среди них у `not` высший приоритет, а у `or` — низший, поэтому `A and not B or C` эквивалентно `(A and (not B)) or C`. Как всегда, явно заданные скобки помогут выразить желаемый порядок выполнения операций.

Булевы операции `and` и `or` — это так называемые *коротящие операции* (`short-circuit operators`): их операнды вычисляются слева направо и вычисление заканчивается как только результат становится определен (очевиден). Например, если `A` и `C` истинны, а `B` — ложно, в условии `A and B and C` выражение `C` не вычисляется. *Коротящая операция* возвращает последний элемент, который был вычислен, что может быть применено не только в целях задания логики.

Можно присвоить результат сравнения, или другого булева выражения, переменной. Например,

```
>>> string1, string2, string3 = '', 'Trondheim',
'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Заметьте, что в Python (в отличие от C) присваивание не может использоваться внутри выражений. Программисты на C могут возмутиться по этому поводу, но на самом деле это позволяет избежать ряда проблем,

обычного для программ на C: указания оператора присваивания (=) в выражении, вместо предполагавшегося сравнения (==).

6.8. Сравнение последовательностей и других типов

Объекты последовательностей можно сравнивать с другими объектами с тем же типом последовательности. Сравнение использует лексикографический порядок: сравниваются первые два элемента, и если они различны — результат сравнения определен; если они равны, сравниваются следующие два элемента и так далее до тех пор, пока одна из последовательностей не будет исчерпана. Если сравниваемые два элемента сами являются последовательностями одного типа, лексикографическое сравнение происходит в них рекурсивно. Если все элементы обеих последовательностей оказались равны, последовательности считаются равными. Если одна последовательность оказывается стартовой последовательностью другой, более короткая последовательность считается меньшей. Лексикографическое упорядочивание строк использует порядок в таблице Unicode для индивидуальных символов. Несколько примеров сравнений между однотипными последовательностями:

```
(1, 2, 3)                < (1, 2, 4)
[1, 2, 3]                < [1, 2, 4]
"Пайтон" < "Паскаль" < "Си" < "Си++"
(1, 2, 3, 4)             < (1, 2, 4)
(1, 2)                   < (1, 2, -1)
(1, 2, 3)               == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'), 4)  < (1, 2, ('abc', 'a'))
```

Обратите внимание, что сравнение объектов различных типов операциями < или > позволено, если объекты имеют соответствующие методы сравнения. Например, смешанные числовые типы сравниваются в соответствии с их численными значениями, так что `0` равен `0.0` и т.д. В противном случае интерпретатор, прервав процесс сортировки, возбудит исключение `TypeError`.

7. МОДУЛИ

Если вы выйдете из интерпретатора и зайдете в него снова, то все определенные вами имена (функции и переменные) будут потеряны. По этой причине, если вы захотите написать несколько более длинную программу, вам лучше использовать текстовый редактор для подготовки ввода для интерпретатора и запускать последний в режиме файлового ввода. Это называется созданием сценария. Если ваша программа становится обширнее, вы можете предпочесть разделить ее на несколько файлов для удобства эксплуатации. Также вы можете захотеть использовать сразу в нескольких

программах некоторую полезную функцию, написанную вами, не копируя ее определение каждый раз.

В языке Python можно поместить требуемые определения в файл и использовать их в сценариях или в интерактивном режиме интерпретатора. Такой файл называется *модулем* (`module`). Определения из модуля могут быть импортированы в других модулях, либо в главном модуле (собрание переменных, к которым есть доступ в сценарии, который непосредственно запускается, и в интерактивном режиме).

Модуль — это файл, содержащий определения и операторы Python. Именем файла является имя модуля с добавленным суффиксом `.py`. Внутри модуля, имя модуля (в качестве строки) доступно в виде значения глобальной переменной с именем `__name__`. Например, используя ваш любимый текстовый редактор, создайте в текущем каталоге файл с именем `fibo.py` со следующим содержимым:

```
"""Модуль вычисления чисел Фибоначчи"""
def fib(n):    # вывести числа Фибоначчи вплоть до n
a, b = 0, 1
while b < n:
print(b, end=' ')
a, b = b, a+b
print()
def fib2(n): # вернуть числа Фибоначчи вплоть до n
result = []
a, b = 0, 1
while b < n:
result.append(b)
a, b = b, a+b
return result
```

Теперь можно войти в интерпретатор Python и импортировать этот модуль следующей командой:

```
>>> import fibo
```

Это действие не переводит имена определенных в модуле функций в текущую таблицу символов, а лишь имя модуля `fibo`. Используя имя модуля, вы можете получить доступ к функциям:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Если вы собираетесь использовать функцию часто, можно присвоить ее локальному имени:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

7.1. Подробнее о модулях

Помимо определений функций модуль может содержать исполняемые операторы. Назначение этих операторов — инициализация модуля: они выполняются при первом импортировании модуля где-либо.

Каждый модуль имеет свою собственную *таблицу символов*, которая используется в качестве глобальной всеми определенными в модуле функциями. Таким образом, автор модуля может использовать глобальные символы в модуле, не опасаясь неожиданных совпадений с глобальными переменными пользователя. С другой стороны, если вы знаете, что делаете, можно сослаться на глобальные переменные модуля, пользуясь той же нотацией, которая применялась для ссылок на его функции: *<имя_модуля>.<имя_элемента>*.

Модули могут импортировать другие модули. Не требуется указывать все операторы `import` в начале модуля (или сценария, с той же целью), но обычно так и делается. Имена из импортированного модуля добавляются в *глобальную таблицу символов* модуля его импортирующего.

Есть вариант оператора `import`, который переносит имена из модуля прямо в таблицу символов импортирующего модуля. Например:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

При этом имя самого модуля, из которого переносятся имена элементов, не добавляется в *локальную таблицу символов* (так, в этом примере, имя `fibo` не определено).

И есть даже способ импортировать все имена, которые определяет данный модуль:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Импортируются все имена, кроме тех, которые начинаются на подчеркивание (_). В большинстве случаев программисты на Python не используют эту возможность, поскольку она внедряет в интерпретатор целый набор новых неизвестных имен и может скрыть некоторые объекты, которые вы уже определили.

Для повышения эффективности, каждый модуль импортируется лишь единожды за сеанс работы с интерпретатором. Поэтому, если вы изменили ваши модули, вам придется перезапустить интерпретатор. Или, если вам

нужно перезагрузить конкретный модуль, можно использовать `imp.reload()` таким образом:

```
import imp; imp.reload(<имя модуля>)
```

7.1.1. Выполнение модулей в качестве сценариев

Когда вы запускаете модуль Python в виде

```
python fibo.py <параметры>
```

то код в этом модуле будет исполнен в момент его импортирования, но значением `__name__` будет строка `"__main__"`. Это значит, что добавляя следующий код в конец сценария:

```
if __name__ == "__main__":
import sys
fib(int(sys.argv[1]))
```

вы можете сделать возможным запуск файла и в качестве сценария, и в качестве импортируемого модуля. Это возможно, поскольку разбирающий командную строку код выполняется только при исполнении модуля как *основного* (`main`) файла:

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

Если модуль импортируется, код не будет выполнен:

```
>>> import fibo
>>>
```

Такой прием часто используется, чтобы предоставить удобный пользовательский интерфейс к модулю или для тестирования (выполнение модуля в качестве сценария запускает набор тестов).

7.1.2. Путь поиска модулей

Если импортируется модуль с именем `spam`, интерпретатор ищет файл с именем `spam.py` в текущем каталоге, а затем в каталогах, указанных в переменной окружения `PYTHONPATH`. У нее такой же синтаксис, как и у переменной шелла `PATH`, которая, в свою очередь, является перечислением каталогов. Когда переменная `PYTHONPATH` не установлена, или файл не найден в описанных в ней местах, поиск продолжается по пути по умолчанию, зависящему от указанного при установке; на Unix это обычно `.:/usr/local/lib/python`.

В действительности поиск модулей производится в списке каталогов в переменной `sys.path`, которая обычно содержит: каталог, в котором находится сценарий на входе (или текущий каталог), `PYTHONPATH` и умолчанием для каталога, указанного при установке. Это позволяет программам на Python (если программист знает, что делает) изменять или подменять путь поиска модулей. Заметьте: поскольку каталог, содержащий запускаемый вами сценарий, также находится в пути поиска, важно, чтобы в нем не

было сценариев с именем стандартного модуля. Иначе, когда этот модуль будет импортироваться, Python будет пытаться загрузить в виде модуля сам сценарий, что в большинстве случаев вызовет ошибку. Для более подробной информации обратитесь к разделу Стандартные модули.

7.1.3. «Скомпилированные» файлы Python

Интерпретатор Python применяет один важный прием для ускорения запуска программы: если в каталоге, где располагается файл с некоторым модулем `spam.py`, находится также файл `spam.pyc`, предполагается, что это уже скомпилированная в байт-код („`byte-compiled`“) версия модуля `spam`. В файле `spam.pyc` зафиксировано время изменения файла `spam.py` версии, использовавшейся для создания `spam.pyc`. Если версии не совпадают — файл `.pyc` игнорируется.

В обычном случае, вам не нужно ничего делать для создания файла `spam.pyc`. Каждый раз, когда `spam.py` успешно компилируется, предпринимается попытка записать скомпилированную версию в `spam.pyc`. Не считается ошибкой, если попытка неудачна: если по какой-либо причине файл не записан полностью, результирующий файл `spam.pyc` будет считаться некорректным и по этой причине в дальнейшем игнорироваться. Содержимое файла `spam.pyc` платформо-независимо, благодаря чему каталог модулей Python может использоваться параллельно машинами с различной архитектурой.

Несколько советов экспертам:

- Когда интерпретатор Python запускается с флагом **`-O`**, в файлах `.pyo` сохраняется сгенерированный оптимизированный код. На данный момент оптимизатор помогает не сильно — он лишь удаляет операторы `assert`. В случае использования **`-O`** оптимизируется весь *байт-код* (`bytecode`); файлы `.pyc` игнорируются, а файлы `.py` компилируются в оптимизированный байт-код.
- Передача двух флагов **`-O`** интерпретатору Python (**`-OO`**) принуждает компилятор байт-кода выполнять оптимизации, в редких случаях результат выполнения которых оказывается некачественно функционирующей программой. На данный момент из байт-кода удаляются только строки `__doc__`, в результате получаются более компактные файлы `.pyo`. Поскольку некоторые программы могут рассчитывать на их (строк) доступность, следует использовать эту возможность только в том случае, если вы знаете что делаете.
- Программа сама по себе не работает хоть сколь-нибудь быстрее, будучи прочитанной из файла `.pyc` или `.pyo`, чем если бы она была прочитана из файла `.py`. Единственный процесс, оказывающийся более быстрым при использовании файлов `.pyc` или `.pyo` — это скорость их подгрузки.

• Если сценарий запущен из командной строки, его байт-код никогда не будет записан в файл `.pyc` или `.pyo`. Таким образом, время запуска сценария может быть уменьшено за счет перемещения большей части его кода в модули или использования небольшого загрузочного сценария, импортирующего этот модуль. Кроме того, можно указывать файл `.pyc` или `.pyo` прямо в командной строке.

• Можно иметь в наличии файл с именем `spam.pyc` (или `spam.pyo`, когда используется **`-O`**), не имея файла `spam.py` для того же модуля. Таким образом можно распространять библиотеки кода Python в том виде, из которого трудно восстановить исходный код.

• Модуль `compileall` может создать файлы `.pyc` (или файлы `.pyo`, когда используется **`-O`**) для всех модулей в каталоге.

7.2. Стандартные модули

Python поставляется с библиотекой стандартных модулей, описанной в отдельном документе, Справочнике по библиотеке Python (далее — «Справочнику по библиотеке»). Некоторые модули встроены в интерпретатор. Они обеспечивают доступ к операциям, не входящим в ядро языка, и встроены для большей эффективности и предоставления доступа к основным средствам операционной системы, таким как *системные вызовы* (`system calls`). Набор таких модулей — выбор настройки, зависимый от используемой платформы. Например, модуль `winreg` предоставляется только на системах с Windows. Один конкретный модуль заслуживает большего внимания: модуль `sys`, встроенный в каждую версию интерпретатора Python. Переменные `sys.ps1` и `sys.ps2` определяют строки, использующиеся в качестве основного и вспомогательного приглашений:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = "Вводите: "
Вводите: print('Ox!')
Ox!
Вводите:
```

Эти две переменные определены только для интерактивного режима интерпретатора.

Переменная `sys.path` представляет из себя список строк, определяющий путь поиска модулей интерпретатора. Она инициализируется значением пути по умолчанию, взятым из переменной окружения `PYTHONPATH`, или встроенным значением по умолчанию, если

PYTHONPATH не установлен. Вы можете изменить ее значение, используя стандартные операции со списками:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

7.3. Функция dir()

Встроенная функция dir() используется для получения имен, определенных в модуле. Она возвращает отсортированный список строк:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__',   '__doc__',   '__excepthook__',
'__name__', '__stderr__',
 '__stdin__',       '__stdout__',       '_getframe',
'api_version', 'argv',
 'builtin_module_names',  'byteorder',  'callstats',
'copyright',
 'displayhook', 'exc_info', 'excepthook',
 'exec_prefix',        'executable',        'exit',
'getdefaultencoding', 'getdlopenflags',
 'getrecursionlimit',  'getrefcount',  'hexversion',
'maxint', 'maxunicode',
 'meta_path',   'modules',   'path',   'path_hooks',
'path_importer_cache',
 'platform',       'prefix',       'ps1',       'ps2',
'setcheckinterval', 'setdlopenflags',
 'setprofile',    'setrecursionlimit',    'settrace',
'stderr', 'stdin', 'stdout',
'version', 'version_info', 'warnoptions']
```

Будучи использованной без аргументов, функция dir() возвращает список имен, определенных в данный момент.

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__doc__', '__file__', '__name__',
'a', 'fib', 'fibo', 'sys']
```

Обратите внимание, что список состоит из имен всех типов: переменных, модулей, функций и т.д.

В возвращаемом функцией `dir()` списке не содержится встроенных функций и переменных. Если вы хотите получить *их* список, то они определены в стандартном модуле `builtins`:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError',                'AssertionError',
'AttributeError', 'BaseException', 'Buffer
Error',       'DeprecationWarning',       'EOFError',
'Ellipsis', 'EnvironmentError', 'Excep
tion',         'False',         'FloatingPointError',
'FutureWarning', 'GeneratorExit', 'IOError
',           'ImportError',           'ImportWarning',
'IndentationError', 'IndexError', 'KeyError',
 'KeyboardInterrupt',  'LookupError',  'MemoryError',
'NameError', 'None', 'NotImp
lemented',     'NotImplementedError',     'OSError',
'OverflowError', 'PendingDeprecatio
nWarning',     'ReferenceError',     'RuntimeError',
'RuntimeWarning', 'StopIteration',
'SyntaxError',    'SyntaxWarning',    'SystemError',
'SystemExit', 'TabError', 'True',
 'TypeError',                   'UnboundLocalError',
'UnicodeDecodeError', 'UnicodeEncodeError', '
UnicodeError',              'UnicodeTranslateError',
'UnicodeWarning', 'UserWarning', 'ValueE
rror',        'Warning',        'ZeroDivisionError',
'__build_class__', '__debug__', '__doc__'
,  '__import__',  '__name__',  'abs',  'all',  'any',
'basestring', 'bin', 'bool', 'b
uffer',   'bytes',   'chr',   'chr8',   'classmethod',
'cmp', 'compile', 'complex', 'cop
yright',   'credits',   'delattr',   'dict',   'dir',
'divmod', 'enumerate', 'eval', 'ex
ec',   'exit',   'filter',   'float',   'frozenset',
'getattr', 'globals', 'hasattr', 'h
ash',   'help',   'hex',   'id',   'input',   'int',
'isinstance', 'issubclass', 'iter', '
len',  'license',  'list',  'locals',  'map',  'max',
'memoryview', 'min', 'next', 'o
bject',   'oct',   'open',   'ord',   'pow',   'print',
'property', 'quit', 'range', 'repr
',  'reversed',  'round',  'set',  'setattr',  'slice',
```

```
'sorted', 'staticmethod', 'st
r', 'str8', 'sum', 'super', 'trunc', 'tuple',
'type', 'vars', 'zip']
```

7.4. Пакеты

Пакеты — способ структурирования *пространств имен* (namespaces) модулей Python за счет использования имен модулей, разделенных точками („dotted module names“). Например, имя модуля A.B означает — подмодуль с именем B в пакете с именем A. Также как использование модулей позволяет авторам различных модулей не заботиться о пересекающихся именах среди глобальных переменных, использование именования через точку позволяет авторам многомодульных пакетов (таких как NumPy или Python Imaging Library) не заботиться о конфликтах имен модулей.

Допустим, вы собираетесь разработать набор модулей (*пакет*, package) для унифицированной работы со звуковыми файлами и звуковыми данными. Существует множество форматов звуковых файлов (обычно их можно распознать по расширению, например: .wav, .aiff, .au). Таким образом, вам может понадобиться создать и поддерживать разрастающуюся коллекцию модулей для конвертирования между различными форматами файлов. Также вам наверняка захочется иметь побольше операций для обработки звуковых данных (таких как смешивание, добавление эха, применение функции эквалайзера, создание искусственного стереоэффекта), поэтому в дополнение к этому вы будете писать нескончаемый поток модулей для исполнения этих операций. Вот возможная структура вашего пакета (выраженная в терминологии иерархической файловой системы):

```
sound/                          Пакет верхнего уровня
__init__.py                Инициализация пакета рабо-
ты со звуком (sound)
formats/                   Подпакет для конвертирова-
ния форматов файлов
__init__.py
wavread.py         (чтение wav)
wavwrite.py        (запись wav)
aiffread.py        (чтение aiff)
aiffwrite.py       (запись aiff)
auread.py          (чтение au)
auwrite.py         (записьau)
...
effects/                    Подпакет для звуковых эф-
```

```
фектов
__init__.py
echo.py             ( эхо )
surround.py         ( окружение )
reverse.py          ( обращение )
...
filters/                    Подпакет для фильтров
__init__.py
equalizer.py        ( эквалайзер )
vocoder.py          ( вокодер )
karaoke.py          ( караоке )
...
```

При импорте пакета Python ищет подкаталог пакета в каталогах, перечисленных в `sys.path`.

Файлы `__init__.py` необходимы для того, чтобы Python трактовал эти каталоги как содержащие пакеты. Это сделано во избежание нечаянного сокрытия правомерных модулей, встречающихся в дальнейшем по пути поиска, каталогами с часто используемыми именами, таким как "`string`". В наипростейшем случае файл `__init__.py` может быть пустым, но в более сложных может содержать код инициализации пакета или устанавливать значение описанной ниже переменной `__all__`.

Пользователи пакета могут импортировать из него конкретные модули, например:

```
import sound.effects.echo
```

Таким образом подгружается подмодуль `sound.effects.echo`. Ссылаться на него нужно используя его полное имя:

```
sound.effects.echo.echofilter(input,   output,   de-
lay=0.7, atten=4)
```

Другой способ импортирования подмодуля:

```
 fromsound.effectsimportecho
```

Так тоже подгружается подмодуль `echo`, но теперь он доступен без префикса пакета, поэтому может использоваться следующим образом:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

И еще один вариант — прямое импортирование желаемой функции или переменной:

```
from sound.effects.echo import echofilter
```

Опять же, таким образом подгружается подмодуль `echo`, но теперь его функция `echofilter()` может быть вызвана непосредственно:

```
echofilter(input, output, delay=0.7, atten=4)
```

Заметьте, что при использовании выражения `from` *`пакет`* `import` *`элемент`*, *элементом* может быть подмодуль (или подпакет) пакета или

любое другое имя, определенное в пакете — например, функция, класс или переменная. Оператор `import` сначала проверяет, определен ли элемент в пакете; если нет — он трактует его как модуль и пытается загрузить. Если не удается его найти, порождается исключение `ImportError`.

Напротив, при использовании синтаксиса в стиле `import` *`элемент.подэлемент.подэлемент`*, все элементы кроме последнего должны быть пакетами; последний элемент может быть модулем или пакетом, но не может быть классом, функцией или переменной, определенными в предыдущем элементе.

*7.4.1. Импорт * из пакета*

Что происходит, когда пользователь пишет `from sound.effects import *`? В идеале, мы бы надеялись, что таким образом код выходит в файловую систему и находит какие подмодули существуют в пакете, импортируя их все. К сожалению, такой метод не очень хорошо работает на платформах Windows, поскольку у файловой системы не всегда есть корректная информация о регистре имен файлов. На этих платформах нет гарантированного способа узнать, нужно ли импортировать файл `ECHO.PY` в качестве модуля `echo`, `Echo` или `ECHO`. (Например, у Windows 95 есть назойливая привычка показывать имена всех файлов с заглавной буквы.) Ограничение DOS на имя файла в формате 8+3 добавляет забавную проблему, связанную с длинными именами модулей.

Единственный выход для автора пакета — предоставить его подробное содержание. Оператор `import` использует следующее соглашение: если в коде файла `__init__.py` текущего пакета определен список `__all__`, то он полагается списком имен модулей, которые нужно импортировать в случае `from` *`пакет`* `import *`. На совести автора поддержка этого списка в соответствующем состоянии в каждой новой версии пакета. Впрочем, авторы пакета могут его не поддерживать вообще, если не видят смысла в импортировании `*` из их пакета. Например, файл `sounds/effects/__init__.py` может содержать следующий код:

```
__all__ = ["echo", "surround", "reverse"]
```

Это будет значить, что выражение `from sound.effects import *` импортирует три именованных подмодуля из пакета `sound`.

Если список `__all__` не определен, оператор `from Sound.Effects import *` не импортирует все подмодули пакета `sound.effects` в текущее пространство имен: он лишь убеждается, что импортирован пакет `sound.effects` (возможно, выполняя код инициализации из `__init__.py`), а затем импортирует все определенные в пакете имена. В этот список попадают любые имена, определенные (и загруженные явно подмодулями) в `__init__.py`. В него также попадают все

явно загруженные предшествующими операторами import подмодули. Рассмотрим следующий код:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

В этом примере модули echo и surround импортируются в текущее пространство имен, поскольку они определены в пакете sound.effects на тот момент, когда исполняется оператор from ... import. (И это также работает, если определен __all__.)

Обратите внимание, что в общем случае импортирование * из модуля не приветствуется, поскольку в результате часто получается плохочитаемый код. Однако вполне нормально использовать его в интерактивных сессиях, чтобы меньше печатать, а определенные модули разработаны для экспорта только тех имен, которые следуют определенным шаблонам.

Помните: в использовании from *пакет* import *определенный_подмодуль* нет ничего плохого. На самом деле — это рекомендованная запись, до тех пор пока при импортировании модуля не нужно использовать подмодули с одинаковым именем из разных пакетов.

7.4.2. Ссылки внутри пакета

Когда пакеты структурированы в подпакеты (например, в случае пакета sound), для того, чтобы сослаться на пакеты-потомки вы можете использовать абсолютное импортирование (absolute imports). Например, если модуль sound.filters.vocoder нуждается в модуле echo из пакета sound.effects, он должен использовать from sound.effects import echo.

Вы можете также использовать *относительное импортирование* (relative imports), применяя следующую форму оператора import: from *модуль* import *имя*. При таком способе импортирования для описания текущего и родительского пакетов используется символ точки. Например, для модуля surround вы можете написать:

```
from . import echo
from .. import formats
from ..filtersimportequalizer
```

Обратите внимание, что относительное импортирование основано на имени текущего модуля. Поскольку имя главного модуля всегда „__main__“, модули, предназначенные для использования в качестве главных модулей приложения на Python, должны всегда использовать *абсолютное импортирование* (absolute imports).

7.4.3. Пакеты в нескольких каталогах

Пакеты поддерживают еще один специальный аттрибут: __path__. Перед исполнением файла __init__.py этого пакета, он инициализиру-

ется списком, содержащим имя каталога, в котором этот файл находится. Изменив переменную, можно повлиять на ход поиска модулей и подпакетов, содержащихся в пакете.

Хотя эта возможность нужна не так часто, она может быть использована для расширения набора модулей, находящихся в пакете.

8. ВВОД И ВЫВОД

Ознакомить пользователя с выводом программы можно различными способами — данные могут быть выведены в читаемом виде или записаны в файл для последующего использования. Часть возможностей будет обсуждена в этой главе.

8.1. Удобное форматирование вывода

На данный момент мы выяснили два способа вывода значений: *операторные выражения* (`expression statements`) и функция `print()`. (Третий способ — использование метода `write()` объектов файлов; на файл стандартного вывода можно сослаться как на `sys.stdout`. Более подробную информацию по этому пункту смотрите в Справочнике по библиотеке.)

Часто возникает желание иметь больший контроль над форматированием вывода, чем обычная печать значений разделенных пробелами. Есть два способа форматирования вашего вывода. Первый способ — выполнять самостоятельно всю работу над строками: используя срезы строк и конкатенацию вы можете создать любой шаблон, какой пожелаете. Стандартный модуль `string` содержит много полезных операций для выравнивания строк по определенной ширине колонки (скоро мы их кратко рассмотрим). Второй способ — использование метода `str.format()`.

Модуль `string` содержит класс `Template`, который предоставляет еще один способ подстановки значений в строки.

Остается, конечно, один вопрос: каким образом конвертировать значения в строки? К счастью, в Python есть два способа для преобразования любого значения в строку — это функции `repr()` и `str()`.

Предназначение функции `str()` — возврат значений в довольно-таки читабельной форме; в отличие от `repr()`, чье назначение — генерирование форм, которые могут быть прочитаны интерпретатором (или вызовут ошибку `SyntaxError`, если эквивалентного синтаксиса не существует). Для тех объектов, у которых нет формы для человеческого прочтения функция `str()` возвратит такое же значение, как и `repr()`. У многих значений, таких как числа или структуры, вроде списков и словарей, оди-

наковая форма для обеих функций. Строки и числа с плавающей точкой, в частности, имеют по две разных формы.

Несколько примеров:

```
>>> s = 'Привет, мир.'
>>>str(s)
'Привет, мир.'
>>>repr(s)
"'Привет, мир.'"
>>>str(0.1)
'0.1'
>>> repr(0.1)
'0.10000000000000001'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'Значение x - ' + repr(x) + ', a y - ' +
repr(y) + '...'
>>>print(s)
Значение x - 32.5, a y - 40000...
>>> # Фунция repr(), примененная к строке, добавляет
кавычки и обратные слэши:
... hello = 'привет, мир\n'
>>> hellos = repr(hello)
>>>print(hellos)
'привет, мир\n'
>>> # Параметром функции repr() может быть объект
Python:
... repr((x, y, ('фарш', 'яйца')))
"(32.5, 40000, ('фарш', 'яйца'))"
```

Вот два способа вывести таблицу квадратов и кубов:

```
>>> for x in range(1, 11):
...      print(repr(x).rjust(2), repr(x*x).rjust(3),
end=' ')
...      # Обратите внимание на использование end в
предыдущей строке
...     print(repr(x*x*x).rjust(4))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
```

```
 8  64  512
 9  81  729
10 100 1000

>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x,
x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

(Обратите внимание, что в первом примере единичные пробелы между колонками добавлены функцией `print()`: она всегда вставляет пробелы между своими параметрами)

Этот пример демонстрирует работу метода строковых объектов `rjust()`, выравнивающего строку по правому краю в поле переданной ширины, отступая пробелами слева. Имеются также похожие методы `ljust()` и `center()`. Эти методы не выводят ничего, они лишь возвращают новую строку. Если строка на входе чересчур длинная, то они не усекают ее, что обычно является меньшим из зол. (Для усечения можно добавить операцию среза, например: `x.ljust(n)[:n]`.)

Есть другой метод — `zfill()`, который заполняет нулями пространство слева от числовой строки. Он распознает знаки плюс и минус:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

Основной способ применения метода `str.format()` выглядит так:

```
>>> print('Мы — те {0}, что говорят
"{1}!"'.format('рыцари', 'Ни'))
Мы — те рыцари, что говорят "Ни!"
```

Скобки с символами внутри (их называют *полями форматирования* (`format fields`)) заменяются на объекты, переданные методу `format`.

Номер в скобках обозначает позицию объекта в списке параметров, переданных методу format.

```
>>> print('{0} и {1}'.format('фарш', 'яйца'))
фарш и яйца
>>> print('{1} и {0}'.format('фарш', 'яйца'))
яйца и фарш
```

Если в методе format используются именованные параметры, можно ссылаться на их значения, используя имя соответствующего аргумента.

```
>>> print('Этот {food} — {adjective}.'.format(
... food='фарш', adjective='непередаваемоужасен'))
Этот фарш — непередаваемо ужасен.
```

Позиционные и именованные параметры можно произвольно совмещать:

```
>>> print('История о {0}е, {1}е, и
{other}е.'.format('Билл', 'Манфред',
other='Георг'))
История о Билле, Манфреде, и Георге.
```

После имени поля может следовать необязательный спецификатор формата ‘:’. С его помощью можно управлять форматированием значения. Следующий пример оставляет у числа Пи только три цифры после десятичного разделителя.

```
>>> import math
>>> print('ЗначениеПИ — примерно
{0:.3f}.'.format(math.pi))
Значение ПИ — примерно 3.142.
```

После спецификатора ‘:’ можно указать число — минимальную ширину поля, выраженную в количестве символов. Это удобно использовать для создания красивых таблиц:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab':
7678}
>>> for name, phone in table.items():
... print('{0:10} ==> {1:10d}'.format(name, phone))
...
Jack       ==>       4098
Dcab       ==> 7678
Sjoerd     ==>       4127
```

Если ваша строка с форматами очень длинна, а вы не хотите разбивать ее на подстроки, было бы неплохо если бы вы могли ссылаться на переменные, предназначенные для форматирования, не по позиции, а по имени. Это можно сделать, просто передав словарь и используя квадратные скобки ‘[]’ для доступа к ключам.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab':
```

```
8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d};
'
'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Тоже самое можно сделать, передав словарь именованных параметров, используя нотацию „**“:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab':
8637678}
>>>print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab:
{Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

В частности, такой прием удобно использовать в сочетании со встроенной функцией `vars()`, которая возвращает словарь с локальными переменными.

Подробное описание форматирования строк с применением метода `str.format()` описано в разделе Синтаксис строк форматирования.

8.1.1. Форматирование строк в старом стиле

Для форматирования строк можно использовать и операцию `%`. Она интерпретирует левый операнд как строку форматирования в стиле `sprintf`, которую следует применить к правому операнду, и возвращает строку, получившуюся в результате этого преобразования. Например:

```
>>> import math
>>> print 'Значение ПИ — примерно %5.3f.' % math.pi
Значение ПИ — примерно 3.142.
```

Поскольку метод `str.format()` довольно нов, большая часть исходных кодов Python все еще использует операцию `%`. Однако, со временем, форматирование строк будет удалено из языка, поэтому в большинстве случаев следует использовать `str.format()`.

Больше информации можно найти в разделе Операции форматирования строк в старом стиле.

8.2. Запись и чтение файлов

Функция `open()` возвращает объект файла и в большинстве случаев используется с двумя аргументами: `open(`*`имя_файла, режим`*`)`.

```
>>> f = open('/tmp/workfile', 'w')
```

Первый параметр — строка, содержащая имя файла. Второй — другая строка, содержащая несколько символов, описывающих способ использования файла. Значение параметра *режим* может быть символом `'r'`, если файл будет открыт только для чтения, `'w'` — открыт только для записи (существующий файл с таким же именем будет стерт) и `'a'` — файл от-

крыт для добавления: любые данные, записанные в файл автоматически добавляются в конец. `'r+'` открывает файл и для чтения, и для записи. Параметр *режим* необязателен: если он опущен — предполагается, что он равен `'r'`.

В обычном случае файлы открываются в *текстовом режиме* (`text mode`) — это значит что вы читаете из файла и записываете в файл строки в определенной кодировке (по умолчанию используется `UTF-8`). Если добавить к режиму файла символ 'b', файл открывается в *двоичном режиме* (`binary mode`): теперь данные считываются и записываются в виде двоичных объектов. Этот режим следует использовать для всех файлов, которые не содержат текст.

При использовании текстового режима, все окончания строк, по умолчанию, специфичные для платформы (`\n` в Unix, `\r\n` в Windows) усекаются до символа `\n`, при чтении из файла, и конвертируются обратно из `\n` в вид, специфичный для платформы, при записи в файл. Эти закулисные изменения в файловых данных корректно работают в случае текстовых файлов, но испортят двоичные данные в файлах вроде `JPEG` или `EXE`. Внимательно следите за тем, чтобы использовать двоичный режим при чтении и записи таких файлов.

8.2.1. Методы объектов-файлов

В примерах ниже подразумевается, что заранее создан файловый объект с именем `f`.

Чтобы прочитать содержимое файла, вызовите `f.read(`*`размер`*`)` — функция читает некоторое количество данных и возвращает их в виде строки или байтового объекта. *`размер`* — необязательный числовой параметр. Если *`размер`* опущен или отрицателен, будет прочитано и возвращено все содержимое файла; если файл по величине в два раза больше оперативной памяти вашего компьютера, то решение этой проблемы остается на вашей совести. В противном случае, будет прочитано и возвращено максимум *`размер`* байт. Если был достигнут конец файла, `f.read()` вернет пустую строку (*''*).

```
>>> f.read()
'Это все содержимое файла.\n'
>>> f.read()
''
```

`f.readline()` читает одну строку из файла; символ новой строки (`\n`) остается в конце прочитанной строки и отсутствует при чтении последней строки файла только если файл не оканчивается пустой строкой. За счет этого возвращаемое значение становится недвусмысленным: если `f.readline()` возвращает пустую строку — достигнут конец файла, в

то же время незаполненная строка, представленная посредством '\n', содержит лишь символ новой строки.

```
>>> f.readline()
'Это первая строка файла.\n'
>>> f.readline()
'Вторая строка файла\n'
>>> f.readline()
''
```

f.readlines() возвращает список, содержащий все строки с данными, обнаруженные в файле. Если передан необязательный параметр подсказка_размера, функция читает из файла указанное количество байт, плюс некоторое количество байт сверх того, достаточное для завершения строки, и формирует список строк из результата. Функция часто используется для более эффективного (файл не загружается в память полностью) построчного чтения больших файлов. Возвращены будут только полные (завершенные) строки.

```
>>> f.readlines()
['Это первая строка файла.\n', 'Вторая строка фай-
ла\n']
```

Альтернативный способ построчного чтения - организация цикла по файловому объекту. Он быстр, рационально использует память и имеет простой код в результате:

```
>>> for line in f:
print(line, end='')
Это первая строка файла.
Вторая строка файла
```

Альтернативный способ проще, но не предоставляет тонкого контроля над происходящим. Поскольку оба этих способа работают с буферизацией строк по-разному, их не следует смешивать.

f.write(*строка*) записывает содержимое *строки* в файл и возвращает количество записанных байтов.

```
>>> f.write('This is a test\n')
15
```

Чтобы записать в файл нечто отличное от строки, предварительно это нечто нужно в строку сконвертировать:

```
>>> value = ('ответ', 42)
>>> s = str(value)
>>>f.write(s)
18
```

f.tell() возвращает целое, представляющее собой текущую позицию в файле f, измеренную в байтах от начала файла. Чтобы изменить позицию объекта-файла, используйте f.seek(*смещение, откуда*). По-

зиция вычисляется прибавлением смещения к точке отсчета; точка отсчета выбирается из параметра *откуда*. Значение 0 параметра *откуда* отмеряет смещение от начала файла, значение 1 применяет текущую позицию в файле, а значение 2 в качестве точки отсчета использует конец файла. Параметр *откуда* может быть опущен и по умолчанию устанавливается в 0, используя начало файла в качестве точки отсчета.

```
>>> f = open('/tmp/workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>>f.seek(5)      # Перейтикшестомубайтувфайле
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2) # Перейти к третьему байту с конца
13
>>> f.read(1)
b'd'
```

При работе с текстовыми файлами (открытыми без символа b в строке режима), выполнять позиционирование (seek) позволяется только от начала файла (за исключением прокрутки в конец файла с использованием seek(0, 2)).

Когда вы закончили все действия над файлом, вызовите f.close() чтобы закрыть его и освободить все системные ресурсы, использованные при открытии этого файла. Все попытки использовать объект-файл после вызова f.close() приведут к возникновению исключения.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

Считается хорошей манерой использовать ключевое слово with при работе с файловыми объектами. Преимущество этого способа в том, что файл всегда корректно закрывается после выполнения блока, либо если при выполнении было порождено исключение. Кроме того, получающийся код намного короче, чем эквивалентная форма с блоками try-finally:

```
>>> with open('/tmp/workfile', 'r') as f:
...     read_data = f.read()
>>>f.closed
True
```

У объектов-файлов есть еще несколько дополнительных методов, таких как isatty() и truncate(), которые используются не так часто;

обратитесь к Справочнику по библиотеке для более полного обзора по файловым объектам.

8.2.2. Модуль `pickle`

Строки могут с легкостью быть записаны в файл и прочитаны из файла. В случае чисел нужно применить несколько больше усилий: метод `read()` возвращает только строки, которые придется передать функции вроде `int()`, которая принимает строку вида `'123'` и возвращает ее числовое значение: `123`. Однако если вы намереваетесь сохранить более сложные типы данных, такие как списки, словари или экземпляры классов, все становится несколько запутаннее.

Вместо того чтобы принуждать программиста постоянно писать и отлаживать код для замысловатых типов данных, Python предоставляет стандартный модуль под названием `pickle`. Это замечательный модуль, который может принять любой объект Python (даже некоторые формы кода на Python!) и конвертировать его в строковое представление: этот процесс называется *консервацией* (`pickling`). Восстановление объекта из его строкового представления называется *расконсервацией* (`unpickling`): строка, описывающая объект, может быть сохранена в файл, добавлена к некоторым данным, или отослана через соединение по сети на удаленный компьютер.

Если у вас есть некоторый объект `x` и объект файла `f`, открытый на запись в двоичном режиме (binary mode, с параметром 'wb'), простейший способ *законсервировать* объект требует одной-единственной строки кода:

```
pickle.dump(x, f)
```

Чтобы снова расконсервировать объект, при условии что `f` — объект файла, открытого для чтения (так же в двоичном режиме, с параметром 'rb'):

```
x = pickle.load(f)
```

(Существуют варианты выполнения этих операций, применяемые при *расконсервации* нескольких объектов или когда вам требуется записать *консервированные* данные в файл; обратитесь к документации по модулю pickle из Справочника по библиотеке.)

Pickle — стандартный способ для создания объектов Python, которые могут быть повторно использованы другими программами или будущими версиями этой же программы; для них есть технический термин — *устойчивый объект* (`persistent object`). Поскольку pickle используется часто, многие авторы расширений для Python заботятся о том, чтобы новые типы данных, такие как матрицы, могли быть корректно *законсервированы* и *расконсервированы*.

9. ОШИБКИ И ИСКЛЮЧЕНИЯ

До этого момента сообщения об ошибках лишь упоминались, но если вы пробовали примеры на практике — возможно, вы уже видели некоторые. Существует (как минимум) два различимых вида ошибок: *синтаксические ошибки* (`syntax errors`) и *исключения* (`exceptions`).

9.1. Синтаксические ошибки

Синтаксические ошибки, также известные как ошибки разбора кода (*парсинга*, `parsing`) — вероятно, наиболее привычный вид жалоб компилятора, попадающихся вам при изучении Python:

```
>>> while True print('Hello world')
File "<stdin>", line 1, in ?
while True print('Hello world')
^
SyntaxError: invalid syntax
```

Парсер повторно выводит ошибочную строку и отображает небольшую «стрелку», указывающую на самую первую позицию в строке, где была обнаружена ошибка. Причина ошибки (или по крайней мере место обнаружения) находится в символе, предшествующем указанному: в приведенном примере ошибка обнаружена на месте вызова функции `print()`, поскольку перед ним пропущено двоеточие (`':'`). Также здесь выводятся имя файла и номер строки, благодаря этому вы знаете в каком месте искать, если ввод был сделан из сценария.

9.2. Исключения

Даже если выражение или оператор синтаксически верны, они могут вызвать ошибку при попытке их исполнения. Ошибки, обнаруженные при исполнении, называются *исключениями* (`exceptions`). Они не фатальны: позже вы научитесь перехватывать их в программах на Python. Большинство исключений, правда, как правило, не обрабатываются программами и приводят к сообщениям об ошибке, таким как следующие:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: int division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: coercing to Unicode: need string or buff-
er, int found
```

Последняя строка сообщения об ошибке описывает произошедшее. Исключения представлены различными типами и тип исключения выводится в качестве части сообщения: в примере это типы `ZeroDivisionError`, `NameError` и `TypeError`. Часть строки, описывающая тип исключения — это имя произошедшего встроенного исключения. Такое утверждение верно для всех встроенных исключений, но не обязано быть истинным для исключений, определенных пользователем (однако, само соглашение — довольно полезное). Имена стандартных исключений — это встроенные идентификаторы (не ключевые слова).

Оставшаяся часть строки описывает детали произошедшего на основе типа исключения, которое было его причиной.

Предшествующая часть сообщения об ошибке показывает контекст, где произошло исключение, в форме стека вызовов. В общем случае она содержит стек, состоящий из списка строк исходного кода; тем не менее, в нее не войдут строки, прочитанные из стандартного ввода.

В разделе Встроенные исключенияСправочника по библиотеке вы найдете список встроенных исключений и их значений.

9.3. Обработка исключений

Существует возможность написать код, который будет перехватывать избранные исключения. Посмотрите на представленный пример, в котором пользователю предлагают вводить число до тех пор, пока оно не окажется корректным целым. Тем не менее, пользователь может прервать программу (используя сочетание клавиш `Control-C` или какое-либо другое, поддерживаемое операционной системой); заметьте — о вызванном пользователем прерывании сигнализирует исключение `KeyboardInterrupt`.

```
>>> while True:
... try:
... x = int(input("Введите, пожалуйста, число: "))
... break
... except ValueError:
... print("Ой!  Это некорректное число.  Попробуйте
еще раз...")
...
```

Оператор `try` работает следующим образом:

- В начале исполняется *блок `try`* (операторы между ключевыми словами `try` и `except`).

- Если при этом не появляется исключений, *блок* `except` не выполняется и оператор `try` заканчивает работу.
- Если во время выполнения блока `try` было возбуждено какое-либо исключение, оставшаяся часть блока не выполняется. Затем, если тип этого исключения совпадает с исключением, указанным после ключевого слова `except`, выполняется блок `except`, а по его завершению выполнение продолжается сразу после оператора `try-except`.
- Если порождается исключение, не совпадающее по типу с указанным в блоке `except` — оно передается внешним операторам `try`; если ни одного обработчика не найдено, исключение считается *необработанным* (`unhandled exception`), и выполнение полностью останавливается и выводится сообщение, схожее с показанным выше.

Оператор `try` может иметь более одного блока `except` — для описания обработчиков различных исключений. При этом будет выполнен максимум один обработчик. Обработчики ловят только те исключения, которые возникают внутри соответствующего блока `try`, но не те, которые возникают в других обработчиках этого же самого оператора `try-except`. Блок `except` может указывать несколько исключений в виде заключенного в скобки кортежа, например:

```
... except (RuntimeError, TypeError, NameError):
... pass
```

В последнем блоке `except` можно не указывать имени (или имен) исключений. Тогда он будет действовать как обработчик группы исключений. Используйте эту возможность с особой осторожностью, поскольку таким образом он может с легкостью перехватить и фактическую ошибку программиста! Также такой обработчик может быть использован для вывода сообщения об ошибке и порождения исключения заново (позволяя при этом обработать исключение коду, вызвавшему обработчик):

```
import sys
try:
f = open('myfile.txt')
s = f.readline()
i = int(s.strip())
except IOError as err:
print("I/O error: {0}".format(err))
except ValueError:
print("Не могу преобразовать данные в целое.")
except:
print("Неожиданная ошибка:", sys.exc_info()[0])
raise
```

У оператора try-except есть необязательный *блок else*, который, если присутствует, должен размещаться после всех блоков except. Его полезно использовать при наличии кода, который должен быть выполнен, если блок try не породил исключений. Например:

```
for arg in sys.argv[1:]:
try:
f = open(arg, 'r')
except IOError:
print('немогуоткрыть', arg)
else:
print(arg, 'содержит', len(f.readlines()), 'строк')
f.close()
```

Использование блока else предпочтительнее, чем добавление дополнительного кода к блоку try, поскольку исключает неожиданный перехват исключения, которое появилось не по причине выполнения кода, защищенного оператором try-except.

При появлении исключения, оно может иметь ассоциированное значение, также известное как *аргумент* (argument) исключения. Присутствие и тип аргумента зависят от типа самого исключения.

В блоке except можно указать переменную, следующую за именем исключения. Переменная связывается с экземпляром исключения, аргументы которого хранятся в instance.args. Для удобства, экземпляр исключения определяет метод __str__(), так что вывод аргументов может быть произведен явно, без необходимости отсылки к .args. Таким образом, вы также можете создать/взять экземпляр исключения перед его порождением и добавить к нему атрибуты по желанию.

```
>>> try:
... raise Exception('spam', 'eggs')
... except Exception as inst:
... print(type(inst))  # экземплярисключения
... print(inst.args)   # аргументы хранятся в .args
... print(inst)     # __str__ позволяет вывести args
явно,
...       # но может быть переопределен в подклассах
исключения
...x, y = inst   # распаковка args
...    print 'x =', x
... print 'y =', y
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
```

```
x = spam
y = eggs
```

Если у исключения есть аргументы, они выводится в качестве последней («детальной») части сообщения о необработанном исключении.

Обработчики исключений перехватывают не только исключения, появившиеся прямо в блоке `try`, но также и возбужденные внутри функций, которые были в блоке `try` вызваны (даже неявно). Например:

```
>>> def this_fails():
... x = 1/0
...
>>> try:
... this_fails()
... except ZeroDivisionError as err:
...  print('Перехват ошибки времени исполнения:',
err)
...
Перехват ошибки времени исполнения: integer division
or modulo by zero
```

9.4. Порождение исключений

Оператор `raise` позволяет программисту принудительно породить исключение. Например:

```
>>> raise NameError('ПриветТам')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: ПриветТам
```

Единственный аргумент оператора `raise` определяет исключение, которое нужно возбудить. Им может быть либо экземпляр исключения, либо класс исключения (класс, дочерний к классу `Exception`).

Если вам нужно определить, было ли возбуждено исключение, не перехватывая его — упрощенная форма оператора `raise` позволит возбудить исключение заново:

```
>>> try:
... raise NameError('ПриветТам')
... except NameError:
... print('Исключение пролетело мимо!')
... raise
...
Исключениепролетеломимо!
Traceback (most recent call last):
File "<stdin>", line 2, in ?
```

```
NameError: ПриветТам
```

9.5. Исключения, определенные пользователем

В программах можно определять свои собственные исключения — посредством создания нового класса исключения. В общем случае, исключения должны быть унаследованы от класса `Exception`: явно или неявно. Например:

```
>>> class MyError(Exception):
... def __init__(self, value):
... self.value = value
... def __str__(self):
... return repr(self.value)
...
>>> try:
... raise MyError(2*2)
... except MyError as e:
...  print('Поймано мое исключение со значением:',
e.value)
...
Поймано мое исключение со значением: 4
>>> raise MyError('ой!')
Traceback (most recent call last):
File "<stdin>", line 1, in ?
__main__.MyError: 'ой!'
```

В этом примере был перегружен конструктор по умолчанию `__init__()` класса `Exception`. Новое поведение отличается лишь созданием нового атрибута `value` и заменяет поведение по умолчанию, при котором создается атрибут `args`.

Классы исключений могут определять любые действия, которые могут делать все другие классы. Однако, обычно их внутренняя структура довольно проста, и предоставляет лишь некоторые атрибуты, позволяющие обработчикам исключений выяснить информацию об ошибке подробно. При создании модуля, который может породить различные ошибки, обычной практикой будет создание базового класса для исключений, определенных в этом модуле, и подклассов для различных ошибочных состояний:

```
class Error(Exception):
"""Базовый класс для всех исключений в этом моду-
ле."""
pass
class InputError(Error):
```

```
"""Исключение порождается при ошибках при вводе.
Атрибуты:
expression -- выражение на вводе, в котором обнару-
жена ошибка
message -- описаниеошибки
"""
def __init__(self, expression, message):
self.expression = expression
self.message = message
class TransitionError(Error):
"""Порождается, когда операция пытается выполнить
неразрешенный переход из одного состояния в другое.

Attributes:
previous -- состояние в начале перехода
next -- новое состояние, попытка принять которое бы-
ла принята
message -- описание, по какой причине такой переход
невозможен
"""
def __init__(self, previous, next, message):
self.previous = previous
self.next = next
self.message = message
```

Большинство исключений имеет имя, заканчивающееся на «`Error`», подобно стандартным исключениям.

Много стандартных модулей определяют собственные исключения, сообщающие об ошибках, которые могут появиться в определяющих их модулях.

9.6. Определение действий при подчистке

У оператора `try` есть другой необязательный блок, предназначенный для операций подчистки, которые нужно выполнить независимо от условий:

```
>>> try:
... raise KeyboardInterrupt
... finally:
... print('Прощай, мир!')
...
Прощай, мир!
Traceback (most recent call last):
File "<stdin>", line 2, in ?
```

```
KeyboardInterrupt
```

Блок `finally` исполняется всегда, когда интерпретатор покидает оператор `try`, независимо — были исключения или нет. Если в блоке `try` появилось исключение, которое не было обработано в блоке `except` (или появилось в самих блоках `except` или `else`) — оно порождается заново после выполнения блока `finally`. Также блок `finally` исполняется «по пути наружу», если какой-либо другой блок оператора `try` был покинут за счет одного из операторов: `break`, `continue` или `return`. Более сложный пример:

```
>>> def divide(x, y):
... try:
... result = x / y
... except ZeroDivisionError:
... print("делениенаноль!")
... else:
... print("результат: ", result)
... finally:
... print("выполнениеблокаfinally")
...
>>>divide(2, 1)
результат:  2
выполнение блока finally
>>> divide(2, 0)
деление на ноль!
выполнение блока finally
>>>divide("2", "1")
выполнениеблокаfinally
Traceback (most recent call last):
File "<stdin>", line 1, in ?
File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str'
and 'str'
```

Как видите, блок `finally` выполняется при любом событии. Ошибка `TypeError` порождается при делении двух строк и не перехватывается блоком `except`, и поэтому порождается заново сразу после выполнения блока `finally`.

В приложениях реального мира, блок `finally` применяется для освобождения внешних ресурсов (таких как файлы или сетевые соединения), независимо от того, было ли их использование удачным.

9.7. Предопределенные действия по подчистке

Некоторые объекты определяют стандартные действия при подчистке, применяемые если объект больше не нужен, независимо от того, удачна была операция использования объекта или нет. Посмотрите на следующий пример, в которым мы пытаемся открыть файл и вывести его содержимое на экран.

```
for line in open("myfile.txt"):
print(line)
```

Проблема этого кода в том, что он оставляет файл открытым на неопределенное количество времени после выполнения данной части кода. В простых сценариях это не является проблемой, но может стать ей в больших приложениях. Оператор `with` позволяет использовать объекты (такие как, например, файлы) таким образом, чтобы вы всегда могли быть уверены в том, что ресурсы будут сразу и корректно очищены.

```
with open("myfile.txt") as f:
for line in f:
print(line)
```

После выполнения оператора, файл *f* всегда закрывается, даже если при прочтении строк обнаружилась проблема. В документации к объектам, которые поддерживают предопределенные действия по подчистке, таким как файлы, эта их способность будет явно указана.

10. КЛАССЫ

За счет механизма классов Python в язык с минимальным использованием нового синтаксиса и семантики добавляется возможность создания классов. Это смесь классовых механизмов, заимствованных из C++ и Modula-3. Как и в случае модулей, классы в Python не устанавливают абсолютного барьера между определением и программистом, рассчитывая больше на аккуратность и вежливость последнего — чтобы он не «врывался в определения». Наиболее важные возможности классов, тем не менее, содержат в себе всю возможную мощь: механизм наследования классов поддерживает несколько предков для класса, производный класс может переопределять любые методы своего предка или предков, а любой его метод может вызвать метод предка с таким же именем. Объекты могут содержать произвольное количество закрытых (`private`) данных.

В терминологии C++, члены класса (включая данные-члены), обычно, открыты (`public`) (исключая Приватные переменные, описанные ниже), а все функции-члены — виртуальны. Нет специальных конструкторов и деструкторов. Как в Modula-3, нет краткой ссылки на члены объекта из его методов: функция-метод определяется с явным первым аргументом, опи-

сывающим объект, который неявно передается при вызове. Как в Smalltalk, классы сами по себе являются объектами, хотя и в более широком смысле: в Python все типы данных — объекты. Таким образом обеспечивается семантика для импортирования и переименования. В отличие от C++ и Modula-3 встроенные типы могут использоваться в качестве предков для расширения возможностей пользователем. Кроме того, как в C++, но не как в Modula-3, большинство встроенных операторов со специальным синтаксисом (арифметические операторы, индексирование и т.д.) могут быть переопределены для экземпляров классов.

10.1. Пара слов о терминологии

Обходя стороной поддерживаемую всем миром терминологию, применимую к разговорам о классах, в нашем случае я буду говорить в терминах C++ и Smalltalk. (Предпочел бы использовать термины языка Modula-3, поскольку Python ближе к ней по объектно-ориентированной семантике, чем к C++, но предполагаю, что немногие читатели слышали о нем.)

Объекты обладают индивидуальностью, и с одним объектом может быть связано несколько имен (в нескольких областях видимости). Такая практика в других языках известна как *совмещение имен* (`aliasing`). На первый взгляд, совмещение малозаметно в Python, и его можно без последствий игнорировать при работе с основными неизменяемыми типами (числами, строками, кортежами). Тем не менее, совмещение имен влияет на семантику программного кода Python, работающего с изменяемыми объектами: списками, словарями и большинством типов, описывающих сущности вне программы (файлы, окна и т.п.). Обычно такая практика считается полезной, поскольку псевдонимы работают подобно указателям и вероятно даже превосходят их возможности. Например, передача объекта - дешевая операция, поскольку по реализации передается только указатель. Если функция изменяет переданный в качестве аргумента объект, это будет заметно и в месте вызова. За счет этого пропадает необходимость в двух различных механизмах передачи аргументов.

10.2. Области видимости и пространства имен в Python

Прежде чем заняться классами необходимо получить представление о правилах областей видимости в Python. Определения классов проделывают над пространствами имен некоторые ловкие трюки. Чтобы полностью понимать происходящее, нужно знать о принципах работы областей видимости и пространств имен. Эти знания не помешают любому профессиональному программисту на Python.

Давайте начнем с нескольких определений.

Пространство имен (namespace) — это набор связей имен с объектами. В настоящий момент большинство пространств имен реализованы в виде словарей Python, но не стоит заострять на этом внимание (если только по поводу производительности): возможно, в будущем реализация изменится. Примеры пространств имен: набор встроенных имен (функции вроде abs() и имен встроенных исключений); глобальные имена в модуле; локальные имена при вызове функции. Важная вещь, которую необходимо знать о пространствах имен — это то, что нет абсолютно никакой связи между именами в разных пространствах имен: например, два разных модуля могут без проблем определять функцию «maximize», так как пользователи модулей будут использовать имена модулей в качестве префиксов.

Кстати, слово *атрибут* (attribute) я применяю к любому имени, следующему за точкой. Например, в выражении z.real, real — это атрибут объекта z. Строго говоря, ссылки на имена в модуле являются ссылками на атрибуты: в выражении *имя_модуля.имя_функции* под *имя_модуля* скрывается объект модуля, а под *имя_функции* — его атрибут. В таком случае обнаруживается прямая связь между атрибутами модуля и глобальными именами, определенными в модуле: они разделяют между собой одно и тоже пространство имен.

Запись в атрибуты может быть запрещена (*атрибут только для чтения*, read-only attribute) или разрешена (*перезаписываемый атрибут*, writable attribute). В последнем случае присваивание атрибуту является возможным. Атрибуты модуля перезаписываемы: вы можете написать "modname.the_answer = 42". Перезаписываемые атрибуты могут также быть удалены оператором del. Например, код "del modname.the_answer" удалит атрибут the_answer из объекта с именем modname.

Пространства имен создаются в различные моменты и имеют разное время жизни. Пространство имен, содержащее встроенные имена создается при запуске интерпретатора и не удаляется никогда. Глобальное пространство имен модуля создается при вычитке определения модуля. Обычно, пространства имен модулей также «живут» до выхода из интерпретатора. Выражения, выполняемые верхне-уровневым порождением интерпретатора, прочитанные из файла сценария или интерактивно, рассматриваются как часть модуля под названием __main__, поэтому у них есть свое собственное глобальное пространство имен. (Встроенные имена по факту также живут в модуле, он называется builtins).

Локальное пространство имен функции создается при ее вызове и удаляется когда функция возвращает значение либо порождает исключение, внутри нее не перехваченное. (На самом деле, лучшим способом объ-

яснить, что происходит на самом деле, было бы «забывание»). Конечно же, рекурсивные порождения имеют свои пространства имен каждое.

Облас ть видимости (`scope`) — это текстовая область в программе на Python, из которой прямым образом доступно пространство имен. «Прямым образом доступно» подразумевает, что явная ссылка на имя вынуждает интерпретатор искать это имя в пространстве имен.

Несмотря на то, что области видимости определяются статически, используются они динамически. В любой момент во время выполнения существует как минимум три вложенных области видимости, чьи пространства имен доступны прямым образом: самая внутренняя область видимости (по ней поиск осуществляется в первую очередь) содержит локальные имена; пространства имен всех объемлющих [данный код] функций, поиск по которым начинается с ближайшей объемлющей [код] области видимости; область видимости среднего уровня, по ней следующей проходит поиск и она содержит глобальные имена текущего модуля; и самая внешняя область видимости (заключительный поиск) — это пространство имен, содержащее встроенные имена.

Если имя определено глобально, тогда все ссылки и присваивания уходят прямо в область видимости среднего уровня, содержащую глобальные имена модуля. Чтобы сменить привязку у всех переменных, найденных вне самой внутренней области видимости, можно использовать оператор `nonlocal`; если такая переменная не объявлена как `nonlocal`, то она используется только для чтения (попытка записать значение в такую переменную создаст новую локальную переменную в самой внутренней области видимости, оставляя идентично названную вовне переменную без изменений).

Обычно локальная область видимости ссылается на локальные имена текущей (на уровне текста) функции. Вне функций локальная область видимости ссылается на то же пространство имен, что и глобальная область видимости: пространство имен модуля. Определения классов помещают в локальную область видимости еще одно пространство имен.

Важно осознавать, что области видимости ограничиваются на текстовом уровне: глобальная область видимости функции, определенная в модуле, является пространством имен этого модуля, независимо от того, откуда или по какому псевдониму была эта функция вызвана. С другой стороны, фактический поиск имен осуществляется динамически, во время выполнения. Как бы то ни было, язык развивается в сторону статического разрешения имен (во время компиляции), так что не стоит полагаться на динамическое разрешение имен. (Фактически, локальные переменные уже определены статично.)

Особая хитрость в Python состоит в том, что — при условии, что в данной области не включены операторы `global` или `nonlocal` — при-

сваивания именам всегда уходят в самую внутреннюю область видимости. Присваивания не копируют данных, а лишь связывают имена с объектами. Тоже самое верно и для удалений: оператор "`del x`" удаляет связь `x` из пространства имен, на которое ссылается локальная область видимости. В действительности, все операции, вводящие новые имена, используют локальную область видимости: в частности, операторы импорта и описаний функций связывают имя модуля или функции в локальной области видимости соответственно. (Для того, чтобы указать определенной переменной, что она должна быть расположена в глобальной области видимости, может использоваться оператор `global`.)

Оператор `global` можно использовать для того, чтобы объявить определенные переменные как привязанные к глобальной области видимости и указывает, что их переназначения должны происходить в ней; оператор `nonlocal` помечает переменные как привязанные к окружающей их области видимости и указывает, что их переназначения должны происходить в ней.

10.3. Пример по областям видимости и пространствам имен

Приведем пример, показывающий, каким образом можно ссылаться на разные области видимости и пространства имен и как `global` и `nonlocal` влияют на привязку переменной.

```
def scope_test():
def do_local():
spam = "локальныйспам"
def do_nonlocal():
nonlocal spam
spam = "нелокальныйспам"
def do_global():
globalspam
spam = "глобальныйспам"
spam = "тестовый спам"
do_local()
print("После локального присваивания:", spam)
do_nonlocal()
print("После нелокального присваивания:", spam)
do_global()
print("После глобального присваивания:", spam)
scope_test()
print("В глобальной области видимости:", spam)
```

Вывод кода из примера таков:

```
После локального присваивания: тестовый спам
После нелокального присваивания: нелокальный спам
```

```
После глобального присваивания: нелокальный спам
В глобальной области видимости: глобальный спам
```

Заметьте, что локальное присваивание (работающее по умолчанию) не заменяет глобальную привязку на связывание из `scope_test`. Нелокальное присваивание заменило глобальную привязку на связывание из `scope_test`, а глобальное присваивание заменило привязку на связывание на уровне модуля.

Можно увидеть, что до глобального присваивания у переменной `spam` не было предшествующих связываний.

10.4. Первый взгляд на классы

В описании классов представлено немного нового синтаксиса, три новых типа объектов и некоторое количество новой семантики.

10.4.1. Синтаксис определения класса

Простейшая форма определения класса выглядит так:

```
class ИмяКласса:
<оператор-1>
.
.
.
<оператор-N>
```

Определения классов, как и определения функций (операторы `def`), должны быть исполнены для того, чтобы определить действие. (Вы можете, предположим, поместить определение класса в ветку оператора `if` или внутрь функции.)

На практике, внутри определения класса обычно помещаются определения функций, но позволено использовать и другие операторы — и иногда с пользой — как мы увидим позже. Определения функций внутри класса имеют особенную форму списка аргументов, в связи с соглашениями по вызову методов — опять же, это будет рассмотрено ниже.

При вводе определения класса создается новое пространство имен, которое и используется в качестве локальной области видимости. Таким образом, все присваивания локальным переменным происходят в этом новом пространстве имен. В частности, определения функций связываются здесь с именами новых функций.

При успешном окончании парсинга определения класса (по достижении конца определения), создается *объект-класс* (`class object`). По существу, это обертка вокруг содержимого пространства имен, созданного во время определения класса; подробнее объекты классов мы изучим в следующем разделе. Оригинальная локальная область видимости (та, которая действовала в последний момент перед вводом определения класса)

восстанавливается, а объект-класс тут же связывается в ней с именем класса, указанном в заголовке определения класса (в примере — *ИмяКласса*).

10.4.2.Объекты-классы

Объекты-классы поддерживают два вида операций: ссылки на атрибуты и создание экземпляра.

Ссылки на атрибуты (Attribute references) используют стандартный синтаксис, использующийся для всех ссылок на атрибуты в Python: *объект.имя*. Корректными именами атрибутов являются все имена, которые находились в пространстве имен класса при создании объекта-класса. Таким образом, если определение класса выглядело так:

```
class MyClass:
"""Простой пример класса"""
i = 12345
deff(self):
return 'приветмир'
```

то MyClass.i и MyClass.f являются корректными ссылками на атрибуты, возвращающими целое и *объект-функцию* (function object) соответственно. Атрибутам класса можно присваивать значение, так что вы можете изменить значение MyClass.i через присваивание. __doc__ также является корректным атрибутом, возвращающим строку документации, принадлежащей классу: "Простой пример класса".

Создание экземпляра класса использует синтаксис вызова функции. Просто представьте, что объект-класс — это непараметризированная функция, которая возвращает новый экземпляр класса. Например (предполагая класс, приведенный выше):

```
x = MyClass()
```

создает новый экземпляр класса и присваивает этот объект локальной переменной x.

Операция *создания экземпляра* (instantiation) создает объект данного класса. Большая часть классов предпочитает создавать экземпляры, имеющие определенное начальное состояние. Для этого класс может определять специальный метод под именем __init__(), например так:

```
def __init__(self):
self.data = []
```

Когда в классе определен метод __init__(), при создании экземпляра автоматически вызывается __init__() нового, только что созданного объекта. Так, в этом примере, новый инициализированный экземпляр может быть получен за счет выполнения кода:

```
x = MyClass()
```

Конечно же, для большей гибкости, метод `__init__()` может иметь параметры. В этом случае аргументы, переданные оператору создания экземпляра класса, передаются методу `__init__()`. Например,

```
>>> class Complex:
... def __init__(self, realpart, imagpart):
... self.r = realpart
... self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>>x.r, x.i
(3.0, -4.5)
```

10.4.3. Объекты-экземпляры

Теперь, что же мы можем делать с объектами-экземплярами? Единственные операции, доступные объектам-экземплярам — это ссылки на атрибуты. Есть два типа корректных имен атрибутов — это атрибуты-данные и методы.

Атрибуты-данные (`data attributes`) аналогичны «переменным экземпляров» в Smalltalk и «членам-данным» в C++. Атрибуты-данные не нужно описывать: как и переменные, они начинают существование в момент первого присваивания. Например, если `x` — экземпляр созданного выше `MyClass`, следующий отрывок кода выведет значение `16`, не вызвав ошибок:

```
x.counter = 1
while x.counter < 10:
x.counter = x.counter * 2
print(x.counter)
delx.counter
```

Другой тип ссылок на атрибуты экземпляра — это *метод* (`method`). Метод — это функция, «принадлежащая» объекту. (В Python термин не уникален для экземпляров класса: другие объекты также могут иметь методы. Например, объекты-списки имеют методы `append`, `insert`, `remove`, `sort` и т.п. Тем не менее, ниже под термином «метод» мы будем понимать только методы объектов-экземпляров классов, пока отдельно не будет указано иное.)

Корректные имена методов объектов-экземпляров зависят от их класса. По определению, все атрибуты класса, являющиеся объектами-функциями, описывают соответствующие методы его экземпляров. Так, в нашем примере, `x.f` является корректной ссылкой на метод, а `x.i` ей не является, поскольку не является и `MyClass.i`. Но при этом `x.f` — это не то же самое, что `MyClass.f`: это объект-метод, а не объект-функция.

10.4.4. Объекты-методы

Обычно, метод вызывают сразу после его связывания [с функцией]:

```
x.f()
```

На примере `MyClass` такой код возвратит строку `'привет мир'`. Однако не обязательно вызывать метод так уж сразу: `x.f` — это объект-метод, он может быть отложен и вызван когда-либо позже. Например:

```
xf = x.f
while True:
print(xf())
```

будет печатать `'привет мир'` до конца времен.

Что конкретно происходит при вызове метода? Вы, возможно, заметили, что `x.f()` выше был вызван без аргументов, хотя в описании функции `f` аргумент был указан. Что же случилось с аргументом? Несомненно, Python порождает исключение когда функция, требующая присутствия аргумента, вызвана без единого — даже, если он на самом деле не используется...

Теперь вы, возможно, догадались: отличительная особенность методов состоит в том, что в качестве первого аргумента функции передается объект. В нашем примере вызов `x.f()` полностью эквивалентен вызову `MyClass.f(x)`. В общем случае, вызов метода со списком из *n* аргументов эквивалентен вызову соответствующей функции со списком аргументов, созданным за счет вставки объекта, вызвавшего метод, перед первым аргументом.

Если вы все еще не поняли, как работают методы, взгляд на реализацию возможно прояснит происходящее. Когда атрибут экземпляра ссылается на что-либо, не являющееся атрибутом-данными, производится поиск по классу. Если имя указывает корректный атрибут класса, являющийся объектом-функцией, создается метод: через упаковку (указателя на) объекта-экземпляра и найденного объекта-функции в абстрактный объект, получается объект-метод. Когда объект-метод вызывается со списком аргументов, он снова распаковывается и новый список аргументов конструируется из объекта-экземпляра и оригинального списка аргументов, и затем уже с новым списком аргументов вызывается объект-функция.

10.5. Различные замечания

Атрибуты-данные переопределяют атрибуты-методы с тем же именем; для того, что обезопасить себя от случайных конфликтов имен, которые могут привести к трудно-обнаруживаемым ошибкам в больших программах, разумно использовать какое-нибудь соглашение, которое могло бы уменьшить шансы возникновения конфликтов. Возможные соглашения включают в себя: написание имен методов строчными буквами, предваре-

ние имени атрибутов-данных некоторой короткой уникальной строкой (предположим, лишь символом подчеркивания ("_")), или использование глаголов для именования методов и существительных для именования данных.

Методы могут ссылаться на атрибуты-данные также как и обычные пользователи («клиенты») объекта. Другими словами, классы не подходят для разработки чистых абстрактных типов данных. Фактически же в Python нет ничего, вынуждающего вас скрывать данные: сокрытие основано на соглашении между программистами. (С другой стороны, реализация Python, написанная на C, может полностью скрывать детали разработки и, если нужно, контролировать доступ к объекту, это можно делать в расширениях для Python, написанных на C).

Клиенты должны использовать атрибуты-данные с осторожностью, так как иначе они могут нарушить инварианты, подразумеваемые методами класса при использовании атрибутов-данных. Заметьте, что обычно клиенты могут добавлять собственные атрибуты-данные к объектам-экземплярам, не нарушая работы методов, если не происходит конфликтов имен. Опять же, соглашение об именовании может избавить вас от головной боли и в этих случаях.

(Прим. перев.) Автором здесь не упомянут механизм свойств (`property`). Свойство синтаксически является атрибутом-данным, но за кулисами с ним могут быть связаны отдельные методы для чтения, записи и удаления. Документация к функции-декоратору `property` хорошо проясняет суть дела:

```
>>> print(property.__doc__)
property(fget=None, fset=None, fdel=None, doc=None)
-> property attribute
fget - функция для чтения, fset - для записи, fdel -
для удаления атрибута.
Типичный пример использования для управляемого атри-
бута x:
class C(object):
def getx(self): return self._x
def setx(self, value): self._x = value
def delx(self): del self._x
x = property(getx, setx, delx, "Я - свойство 'x'.")
Декораторы упрощают определение новых свойств и из-
менение существующих:
class C(object):
@property
def x(self): return self._x
@x.setter
```

```
def x(self, value): self._x = value
@x.deleter
def x(self): del self._x
```
(Конец прим.)

У методов нет краткой записи для ссылок изнутри на атрибуты-данные (и другие методы!). Это и вправду повышает читабельность методов: нет шанса спутать локальные переменные и переменные экземпляров при просмотре тела метода.

Обычно, первый аргумент метода называется `self`. Это не более чем соглашение: имя `self` не имеет абсолютно никакого специального смысла для языка Python. (Однако, обратите внимание, что если вы не следуете соглашениям, ваш код может стать менее читабелен для других программистов; и также, потенциально, программа навигации по классам может опираться на такие соглашения.)

Любой объект-функция, являющийся атрибутом класса, определяет метод для экземпляров этого класса. Не так важно, чтобы текст определения функции был заключен в определение класса: присваивание объекта-функции локальной переменной класса также работает неплохо. Например:

```
# Функция, определенная вне класса
def f1(self, x, y):
return min(x, x+y)
class C:
f = f1
def g(self):
return 'приветмир'
h = g
```

Теперь `f`, `g` и `h` — все являются атрибутами класса `C`, ссылающимися на объекты-функции, и следовательно, все они являются методами экземпляров `C` — `h` становится полностью эквивалентен `g`. Заметьте, что такая практика обычно лишь запутывает читателя программы.

Методы могут вызывать другие методы за счет использования атрибутов-методов аргумента `self`:

```
class Bag:
def __init__(self):
self.data = []
def add(self, x):
self.data.append(x)
def addtwice(self, x):
self.add(x)
self.add(x)
```

Методы могут ссылаться на глобальные имена таким же образом, как и обычные функции. Глобальная область видимости, связанная с методом — это модуль, содержащий определение класса. (Сам класс никогда не используется в качестве глобальной области видимости!) В то время, как одни редко находят причины для использования глобальных данных в методах, существует множество разумных причин использовать глобальную область видимости: для примера, функции и модули, импортированные в глобальную область видимости, могут использоваться в методах так же, как в функциях и классах, в ней определенных. Обычно класс, содержащий метод, сам определен в этой глобальной области видимости, и в следующем разделе мы найдем пару хороших причин, почему методу может быть необходимо ссылаться на собственный класс!

10.6. Наследование

Конечно же, не поддерживай «класс» наследование, не стоило бы называть его «классом». Синтаксис производного класса выглядит так:

```
class ИмяПроизводногоКласса(ИмяБазовогоКласса):
<оператор-1>
.
.
.
<оператор-N>
```

Имя *ИмяБазовогоКласса* должно быть определено в области видимости, содержащей определение производного класса. Вместо имени базового класса также позволяется использовать другие выражения. Это может быть полезно, например, когда базовый класс определен в другом модуле:

```
class ИмяПроизводногоКласса(имямодуля.ИмяБазовогоКласса):
```

Использование определения производного класса проходит таким же образом, как и базового. Базовый класс полностью сохраняется по завершению конструирования объекта-класса. Такой метод используется для разрешения ссылок на атрибуты: если запрошенный атрибут не был найден в самом классе, поиск продолжается в базовом классе. Правило применяется рекурсивно, если базовый класс сам является производным от некоторого другого класса.

В создании экземпляров производных классов нет ничего особенного: `ИмяПроизводногоКласса()` создает новый экземпляр класса. Ссылки на методы разрешаются следующим образом: производится поиск соответствующего атрибута класса (спускаясь вниз по цепочке базовых классов,

если необходимо) и ссылка на метод считается корректной, если она порождает объект-функцию.

Производные классы могут перегружать методы своих базовых классов. Поскольку у методов нет особых привилегий при вызове других методов того же объекта, метод базового класса, вызывающий другой метод, определенный в этом же классе, может вызвать перегруженный метод производного класса. (Для программистов на C++: все методы в Python фактически виртуальны.)

При перегрузке метода в производном классе возможна не только замена действия метода базового класса с тем же именем, но и его расширение. Существует простой способ вызвать метод базового класса прямым образом: просто вызовите "`ИмяБазовогоКласса.имяметода(self, аргументы)`". Такой способ будет неожиданно полезным и для клиентов. (Обратите внимание, что он работает только если базовый класс определен и импортирован прямо в глобальную область видимости).

В языке Python есть функции, которые работают с наследованием:

- Используйте `isinstance()` чтобы проверить тип объекта: `isinstance(obj, int)` возвратит `True` только если `obj.__class__` является `int` или некоторым классом, наследованным от `int`.
- Используйте `issubclass()` чтобы проверить наследственность класса: `issubclass(bool, int)` возвратит `True`, поскольку класс `bool` является наследником (`subclass`) `int`. Однако, `issubclass(float, int)` возвратит `False`, поскольку класс `float` не является наследником `int`.

10.6.1. Множественное наследование

Python также поддерживает форму *множественного наследования* (`multiple inheritance`). Определение класса с несколькими базовыми классами будет выглядеть так:

```
class ИмяПроизводногоКласса(Базовый1, Базовый2, Базовый3):
<оператор-1>
.
.
.
<оператор-N>
```

В простейших случаях и для большинства задач, вы можете представлять себе поиск атрибутов, наследованных от родительского класса в виде «сперва вглубь», затем «слева-направо». Таким образом, если атрибут не найден в *`ИмяПроизводногоКласса`*, его поиск выполняется в *`Базо-`*

вом1, затем (рекурсивно) в базовых классах *Базового1* и только если он там не найден, поиск перейдет в *Базовый2* и так далее.

На самом деле все немного сложнее. Порядок разрешения методов (`method resolution order`) меняется динамически, чтобы обеспечить возможность сотрудничающих вызовов `super()`. Этот способ известен в некоторых других языках с поддержкой множественного наследования как "вызов-следующего-метода" („`call-next-method`“) и имеет больше возможностей, чем вызов родительского метода в языках с единичным наследованием.

Динамическое упорядочивание (`dynamic ordering`) имеет важность, поскольку все вариации множественного наследования проявляют в себе эффект ромбовых отношений (когда как минимум один родительский класс может быть доступен различными путями из низшего в иерархии класса). Например, все классы наследуются от `object`, так что множественное наследование в любом виде предоставляет более одного пути для того, чтобы достичь `object`. Чтобы защитить базовые классы от двойных и более запросов, динамический алгоритм «выпрямляет» (`linearizes`) порядок поиска таким образом, что тот сохраняет указанный слева-направо порядок для каждого класса, который вызывает каждый родительский класс только единожды и является монотонным (значит, класс можно сделать наследником, не взаимодействуя с порядком предшествования его родителей). Обобщенные вместе, эти свойства позволяют разрабатывать надежные и расширяемые классы, используя множественное наследование. С подробностями можно ознакомиться по этой ссылке: http://www.python.org/download/releases/2.3/mro/ (перевод).

10.7. Приватные переменные

В Python имеется ограниченная поддержка идентификаторов, приватных для класса. Любой идентификатор в форме `__spam` (как минимум два предшествующих символа подчеркивания, как максимум один завершающий) заменяется дословно на `_classname__spam`, где `classname` — текущее имя класса, лишенное предшествующих символов подчеркивания. Это *искажение* (`mangling`) производится без оглядки на синтаксическую позицию идентификатора, поэтому может использоваться для определения переменных, приватных для класса экземпляров, переменных класса, методов, переменных в глобальной области видимости (`globals`), и даже переменных, использующихся в экземплярах, приватных для этого класса на основе экземпляров *других* классов. Имя может быть обрезано, если его длина превышает 255 символов. Вне классов, или когда имя состоит из одних символов подчеркивания, искажения не происходит.

Предназначение искажения имен состоит в том, чтобы дать классам легкую возможность определить «приватные» переменные экземпляров и методы, не беспокоясь о переменных экземпляров, определенных в производных классах, и о забивании кодом вне класса переменных экземпляров. Обратите внимание, что правила искажения имен разработаны, в основном, чтобы исключить неприятные случайности — решительная душа все еще может получить доступ или изменить переменные, предполагавшиеся приватными. В некотором особом окружении, таком как отладчик, это может оказаться полезным — и это единственная причина, по которой лазейка не закрыта. (Внимание: наследование класса с таким же именем как и у базового делает возможным использование приватных переменных базового класса).

Заметьте, что код, переданный в `exec()` или `eval()`, не предполагает в качестве текущего имени класса имя класса, порождающего вызов — так же, как и в случае эффекта с оператором `global` — эффекта, который также ограничен для всего побайтно-компилирующегося кода. И, такое же ограничение применимо для функций `getattr()`, `setattr()` и `delattr()`, и также для прямой ссылки на `__dict__`.

10.8. Всякая всячина

Иногда бывает полезен тип данных, похожий на `record` из языка Pascal или `struct` из языка C, например, для хранения нескольких поименованных элементов данных. Для этой цели подойдет даже пустое определение класса:

```
class Employee:
pass
john = Employee() # Создать пустую запись о рабочем
# Заполнить поля записи
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

Фрагменту кода на Python, требующему на входе некоторого абстрактного типа данных, можно дать экземпляр, эмулирующий методы этого типа данных. Например, если имеется функция, умеющая форматировать данные из файлового объекта, то можно определить класс с методами `read()` и `readline()` (работающие с данными, скажем, из строкового буфера) и передать ей экземпляр этого класса в качестве аргумента.

Объекты-методы экземпляров также имеют атрибуты: `m.__self__` — исходный объект-экземпляр с методом `m()`, а `m.__func__` — объект-функция, соответствующий методу.

10.9. Исключения — тоже классы

Исключения, определенные пользователем, могут быть также отождествлены с классами. При использовании этого механизма становится возможным создавать расширяемые иерархии исключений.

Оператор `raise` имеет следующие (синтаксически) правильные формы:

```
raise Класс
raise Экземпляр
```

В первой форме, *`Класс`* должен быть экземпляром типа или класса, производного от него. Первая форма является краткой записью следующего кода:

```
raise Class()
```

Класс в блоке `except` является сопоставимым с исключением, если является этим же классом или самим по себе базовым классом (никаких других способов обхода — описанный в блоке `except` производный класс не сопоставим с базовым). Например, следующий код выведет B, C, D в этом порядке:

```
class Employee:
pass
john = Employee() # Создать пустую запись о рабочем
# Заполнить поля записи
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

Обратите внимание, что если бы блоки `except` шли в обратном порядке (начиная с "`except B`"), код вывел бы B, B, B — сработал бы первый совпадающий блок `except`.

При выводе сообщения об ошибке о необработанном исключении, выводится класс исключения, затем двоеточие и пробел, и наконец экземпляр, приведенный к строке за счет встроенной функции `str()`.

10.10. Итераторы

К этому моменту вы, возможно, заметили, что используя оператор `for` можно организовать цикл по большинству объектов-контейнеров:

```
for element in [1, 2, 3]:
print(element)
for element in (1, 2, 3):
print(element)
for key in {'один':1, 'два':2}:
print(key)
for char in "123":
```

```
print(char)
for line in open("myfile.txt"):
print(line)
```

Такой стиль доступа к элементам прост, лаконичен и удобен. Использованием *итераторов* (iterators) пропитан язык Python, и это его выделяет среди других. Негласно, оператор for вызывает метод iter() объекта-контейнера. Функция возвращает объект итератора, который определяет метод __next__(), который по очереди получает доступ к элементам в контейнере, по одному за раз. Если больше не остается элементов, метод __next__() порождает исключение StopIteration, которое сообщает оператору for о необходимости завершения прохода. Вы можете вызывать метод __next__() посредством встроенной функции next(); следующий пример показывает, как это работает:

```
>>> s = 'абв'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> next(it)
'а'
>>> next(it)
'б'
>>> next(it)
'в'
>>> next(it)
Traceback (most recent call last):
File "<stdin>", line 1, in ?
next(it)
StopIteration
```

Ознакомившись с механизмами, скрытыми за протоколом итераторов, легко добавить возможность итерирования к вашим классам. Определите метод __iter__(), который возвращает объект с методом next(). Если класс определяет и метод next(), тогда __iter__() может просто возвращать self.

```
class Reverse:
"Итератор по последовательности в обратном направле-
нии"
def __init__(self, data):
self.data = data
self.index = len(data)
def __iter__(self):
return self
```

```
def __next__(self):
if self.index == 0:
raise StopIteration
self.index = self.index - 1
return self.data[self.index]
>>> for char in Reverse('спам'):
... print(char)
...
м
а
п
с
```

10.11. Генераторы

Генераторы (generators) — простой и мощный инструмент для создания итераторов. Они записываются как обычная функция, но где бы им ни было необходимо вернуть данные, используется оператор `yield`. Каждый раз, когда над ним вызывается `next()`, генератор возвращается к месту, где он был оставлен (он запоминает все значения данных, а также какой оператор был выполнен последним). Пример показывает, что создание генераторов может быть тривиально простым:

```
def reverse(data):
for index in range(len(data)-1, -1, -1):
yield data[index]

>>> for char in reverse('гольф'):
... print(char)
...
ф
ь
л
о
г
```

Все, что можно сделать с использованием генераторов, может быть сделано с использованием основанных на итераторах классов, как описано в предыдущем разделе. Благодаря автоматическому созданию методов `__iter__()` и `__next__()` генераторы так компактны.

Другая важная особенность состоит в том, что между вызовами сохраняются локальные переменные и *состояние выполнения* (execution state). Это позволяет конструкциям функций быть проще, а получению

переменных экземпляров быть намного легче, нежели с использованием `self.index` и `self.data`.

В дополнение к автоматическому созданию методов и сохранению состояния, когда генераторы заканчивают свое действие, они автоматически порождают исключение `StopIteration`. В комбинации, эти особенности позволяют легко создавать итераторы не прилагая усилий больших, чем нужно для написания обычной функции.

10.12. Выражения-генераторы

Некоторые простые генераторы могут быть сжато закодированы в выражении с использованием синтаксиса, схожего со *списковыми сборками*, но с круглыми скобками вместо квадратных. Выражения-генераторы разработаны в основном для случаев, когда генератор тут же используется в качестве аргумента функции. Выражения с генераторами более компактные, но менее гибкие чем полные определения генераторов и обычно используют память экономнее, чем эквивалентные *списковые сборки*.

Примеры:

```
>>> sum(i*i for i in range(10))                          #
суммаквадратов
285
>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>>sum(x*yforx,yinzip(xvec, yvec))          # скаляр-
ноепроизведение
260
>>> from math import sin, radians
>>>  sine_table  =  {x:  sin(radians(x))  for  x  in
range(0, 91)}
>>> unique_words = set(word  for line in page  for
word in line.split())
>>> valedictorian = max((student.gpa, student.name)
for student in graduates)
>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -
1))
['f', 'l', 'o', 'g']
```

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Коэлье Л.П., Ричерт В. Построение систем машинного обучения на языке PythonПеревод с английского. – М.: ДМК Пресс, 2015. – ISBN 978-5-97060-330-7.
2. Маккинли У. Python и анализ данных; пер. с англ. – М.: ДМК Пресс, 2015. — 482 с. — ISBN 978-5-97060-315-4.
3. Марк Саммерфилд. Python на практике. — Перевод с английского. — М.: ДМК Пресс, 2014. — 338 с. — ISBN 978-5-97060-095-5.
4. Марк Лутц. Изучаем Python, 4-е издание. — Перевод с английского. — СПб.: Символ-Плюс, 2010. — 1280 с — ISBN 978-5-93286-159-2
5. Марк Саммерфилд. Программирование на Python 3. Подробное руководство. – Перевод с английского. – СПб.: Символ-Плюс, 2009. – 608 с. – ISBN 978-5-93286-161-5.
6. Ноа Гифт, Джереми М. Джонс. Python в системном администрировании UNIX и Linux. — Перевод с английского. — СПб.: Символ-Плюс, 2009. – 512 с. – ISBN 978-5-93286-149-3.
7. Сузи Р.А. Python. Наиболее полное руководство (+CD). – СПб.: БХВ-Петербург, 2002. – 768 с. – ISBN 5-94157-097-X.
8. Хахаев И.А. Практикум по алгоритмизации и программированию на Python: учебник. – М.: Альт Линукс, 2010. – 126 с. (Библиотека ALT Linux). – ISBN 978-5-905167-02-7.
9. Шапошникова С. Основы программирования на Python: учебник. Вводный курс; версия 2. – 2011. – 44 с.
10. https://docs.python.org/3.1/tutorial/ - Официальные документация по языку Python (дата обращения на сайт 23.04.2015 13:34).

У ч е б н о е и з д а н и е

Артеменко М.А.
Лебеденко А.В.
Сафьяник А.О.

ОСНОВЫ ЗАЩИТЫ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ВВЕДЕНИЕ В PYTHON

Учебное пособие

Редактор - О.В. Дмитриева

Изд. № 127/18. Объем 6,25 п.л.
РИИЦМ ФГАОУ ВО «Севастопольский государственный университет»