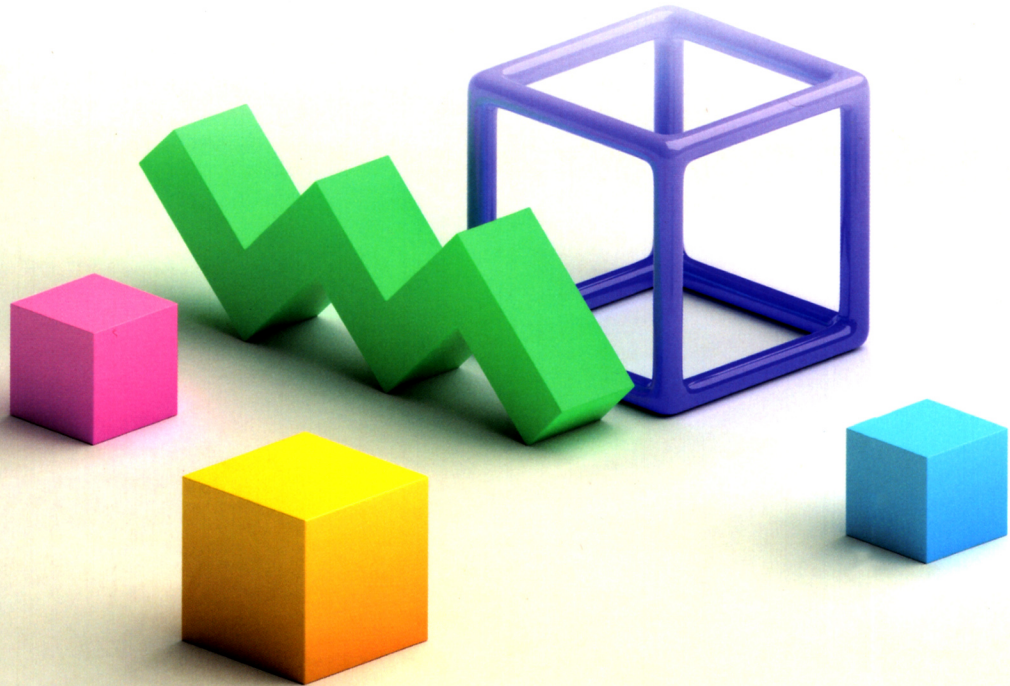
ДЖОШ КОГЛИАТИ

PYTHON

ДЛЯ НЕПРОГРАММИСТОВ
САМОУЧИТЕЛЬ
В ПРИМЕРАХ



ДЖОШ КОГЛИАТИ

PYTHON

ДЛЯ НЕПРОГРАММИСТОВ
САМОУЧИТЕЛЬ
В ПРИМЕРАХ

МОСКВА
ИЗДАТЕЛЬСТВО АСТ

УДК 004.43
ББК 32.973.26-018.1
К57

В подготовке книги принимали участие:
Митчелл Эйкенс, Киа Моранте, Элизабет Коглиати,
Джеймс А. Браун, Джо Оппегаард, Бенджамин Хелл

Коглиати, Джош

К57 Python для непрограммистов. Самоучитель в примерах / Дж. Коглиати. — Москва : Издательство АСТ, 2024. — 96 с. : ил. — (Учимся программировать).

ISBN 978-5-17-162198-8.

Эта книга в кратчайшие сроки поможет на базовом уровне освоить Python — идеальный для новичка, доступный и понятный язык программирования, позволяющий легко создавать интересные и креативные приложения. Буквально с первых страниц читатель, даже если он никогда в жизни не писал код, с легкостью начнет постигать Python и на основе простых практических заданий приступит к написанию собственных программ, сначала состоящих из двух-трех строк кода, а затем, по мере усвоения базовых навыков программирования, все более сложных. Информация в книге изложена простым языком, четко, последовательно и логично, с изрядной долей юмора, поэтому учебный материал легко воспринимается и запоминается. Делая упор на самом главном, автор книги предоставляет читателю основные знания о переменных, типах и структурах данных, функциях, циклах, логических выражениях и отладке программ, сразу же иллюстрируя теорию подробно прокомментированными практическими примерами.

УДК 004.43
ББК 32.973.26-018.1

ISBN 978-5-17-162198-8

Перевод на русский язык: ООО «Интеджер».
Издание на русском языке: ООО «Издательство АСТ».

Содержание

Глава 1

Предисловие

Все примеры исходного кода Python в этом руководстве являются общественным достоянием, поэтому вы можете изменять его и переиздавать под любой лицензией. Поскольку предполагается, что вы будете изучать программирование, лицензия Creative Commons Attribution-ShareAlike требует, чтобы все программы, созданные на основе исходного кода из этого учебника, были под этой лицензией. Но исходный код Python является общественным достоянием, так что это требование отменяется.

Данный учебник является относительно полной версией «Учебника для непрограммистов по Python 2.6». Более старые его версии и некоторые версии на корейском, испанском, итальянском и греческом языках доступны на сайте http://jjc.freeshell.org/easytut/.

Данное учебное пособие предоставляет начальные знания о языке программирования Python и предназначено для читателей, не имеющих опыта программирования.

Если вы ранее программировали на других языках, я рекомендую использовать учебник «Python для программистов» (https://docs.python.org/3/tutorial/index.html), написанный Гвидо ван Россумом.

Если в процессе изучения книги у вас возникнут вопросы или комментарии, пожалуйста, пишите автору. Я приветствую любые вопросы и комментарии по этому учебнику и постараюсь ответить на них как можно более полно и исчерпывающе.

Спасибо Джеймсу А. Брауну за написание большей части главы об установке Python на Windows. Спасибо также Элизабет Коглиати за то, что она много жаловалась :) на оригинальный учебник (практически непригодный для непрограммиста), за его вычитку, а также за множество идей и комментариев к нему. Спасибо Джо Оппегаарду за написание почти всех упражнений. Спасибо всем, кого я не упомянул.

Другие ресурсы

- Домашняя страница Python (http://www.python.org)
- Документация по Python 3 (https://docs.python.org/3/)
- «A Byte of Python», пособие, написанное автором C. H. Swaroop (http://www.swaroopch.com/notes/python)
- Переход на Python 3: подробное руководство (http://python3porting.com/)

Глава 2

Введение

Первым делом

Итак, вы никогда раньше не программировали. При помощи этого пособия я попытаюсь научить вас создавать программы. На самом деле есть только один способ научиться программировать: вы должны *читать код* и *писать код* (так часто называют компьютерные программы). Я собираюсь показать вам много кода. Вы должны набирать код, который я привожу, чтобы посмотреть, что получится. «Играйте» с этим кодом и вносите в него свои изменения. Худшее, что может случиться, — он не будет работать.

Когда я ввожу код, он форматируется следующим образом:

```
##Python легко изучать
print("Hello, World!")
```

Это нужно для того, чтобы его можно было легко отличить от остального текста.

Если компьютер выводит что-то на экран, это будет отформатировано следующим образом:

```
Привет, мир!
```

(Обратите внимание, что мы применяем оператор печати `print`, но текст выводится на экран, и мы не используем бумагу. До того как у компьютеров появились экраны, результаты работы компьютерных программ печатались на бумаге).

Обратите внимание, что это учебник по Python 3, а это значит, что большинство примеров не будут работать в Python 2.7 и более ранних версиях. Кроме того, некоторые дополнительные библиотеки (библиотеки сторонних разработчиков) еще не конвертированы. Возможно, вы захотите изучить «Учебник для непрограммистов по Python 2.6». Однако различия между версиями не очень существенны, так что если вы изучите одну из них, то сможете читать программы, написанные для другой, без особых проблем.

Иногда текст, который вы вводите по запросу программы, будет выделен жирным шрифтом (но не во всех случаях), и текст, который вы увидите на экране, будет выглядеть следующим образом:

```
Стой!
Кто идет? Джош
Ты можешь идти, Джош.
```

Я также познакомлю вас с терминологией программирования — например, с тем, что программирование часто называют *кодингом*. Это не только даст вам возможность понять, о чем говорят программисты, но и поможет в процессе обучения. Теперь перейдем к более важным

вещам. Чтобы программировать на Python, вам необходимо установить на свой компьютер Python 3. Если у вас еще не установлен Python, зайдите на сайт www.python.org/download (http://www.python.org/download/) и получите нужную версию для вашей операционной системы. Скачайте ее, прочитайте инструкции и установите.

Установка Python

Для программирования на Python вам понадобятся рабочая инсталляция Python и текстовый редактор. Python поставляется с собственным редактором, *IDLE*, который довольно неплох и вполне достаточен для начала. По мере углубления в программирование вы, вероятно, перейдете на какой-нибудь другой редактор, например emacs, vi или какой-либо еще.

Страница загрузки Python находится по адресу http://www.python.org/download. *Python 2.7 и более старые версии не будут работать с материалами этого учебника.* На сайте для скачивания доступны различные установочные файлы для разных компьютерных платформ. Здесь приведены некоторые конкретные инструкции для наиболее распространенных операционных систем.

Установка на Linux, BSD и Unix

Возможно, вам повезло, и Python уже установлен на вашей машине. Чтобы проверить это, введите `python3` в командной строке. Если вы увидите что-то похожее на то, что показано в следующей главе, значит, все готово.

IDLE может потребоваться установить отдельно, из собственного пакета, например `idle3`, или как часть `python-tools`.

Если нужно установить Python, сначала попробуйте воспользоваться менеджером пакетов операционной системы или зайдите в репозиторий, где доступны ваши пакеты, и поищите Python 3. Python 3.0 был выпущен уже давно, во всех дистрибутивах должен быть доступен Python 3, так что вам, возможно, не придется компилировать его с нуля. В Ubuntu и Fedora доступны бинарные пакеты Python 3, но они еще не установлены по умолчанию, поэтому их нужно устанавливать специально.

Примерно так выглядит компиляция Python из исходного кода в Unix (если эти шаги не совсем понятны, возможно, вам стоит прочитать другое введение в *nix, например, «Введение в Linux» (http://tldp.org/LDP/intro-linux/html/index.html)):

- Скачайте файл .tgz (используйте веб-браузер, чтобы получить gzipped tar-файл с сайта https://www.python.org/downloads/release/python-343)
- Распакуйте tar-файл (укажите правильный путь к месту его загрузки):

```
$ tar -xvzf ~/Download/Python-3.4.3.tgz
... список файлов в несжатом виде
```

- Перейдите в каталог и попросите компьютер скомпилировать и установить программу:

```
$ cd Python-3.4/
$ ./configure --prefix=$HOME/python3_install
... много выходных данных. Следите за сообщениями об ошибках здесь ...
$ make
... еще больше выходных данных. Надеюсь, никаких сообщений об ошибках ...
$ make install
```

- Добавьте Python 3 в свой путь. Сначала вы можете проверить его, указав полный путь. Вы должны добавить `$HOME/python3_install/bin` в переменную PATH bash:

```
$ ~/python3_install/bin/python3
Python 3.4.3 (... информация о размере и дате...)
[GCC 4.5.2] на linux2
Введите "help", "copyright", "credits" или "license" для получения
дополнительной информации.
>>>
```

Приведенные выше команды установят Python 3 в ваш домашний каталог, что, вероятно, и нужно, но если пропустить `--prefix=$HOME/python3_install`, он будет установлен в `/usr/local`. Если вы хотите использовать графический редактор кода IDLE, вам нужно убедиться, что в системе установлены библиотеки `tk` и `tcl`, а также файлы их разработки. Если они недоступны, то на этапе `make` вы получите предупреждение.

Установка на Mac

Начиная с Mac OS X Tiger Python поставляется с операционной системой по умолчанию, но вам придется обновить его до Python 3, пока OS X не начнет включать Python 3 (проверьте версию, запустив `python3` в терминале командной строки). Также в стандартной установке может отсутствовать IDLE (редактор Python). Если вы хотите (заново) установить Python, возьмите дистрибутив для MacOS с сайта загрузки Python (https://www.python.org/downloads/release/python-343/).

Установка на Windows

Загрузите соответствующий установщик Windows (установщик MSI для x86 (https://www.python.org/ftp/python/3.4.3/python-3.4.3.msi), если у вас не 64-битный процессор AMD или Intel). Запустите программу установки двойным щелчком мыши и следуйте подсказкам.

Дополнительную информацию см. на сайте https://docs.python.org/3/using/windows.html#installing-python.

Настройка переменной окружения PATH

Переменная среды PATH — это список папок, разделенных точками с запятой, в которых Windows будет искать программу, когда вы попытаетесь выполнить ее, набрав ее имя в командной строке. Вы можете узнать текущее значение PATH, набрав эту команду в командной строке:

```
echo %PATH%
```

Самый простой способ постоянно изменять переменные окружения — вызвать встроенный редактор переменных окружения в Windows. В разных версиях Windows этот редактор выглядит по-разному.

В **Windows 8**: Нажмите клавишу Windows и введите *Control Panel*, чтобы найти панель управления Windows. Открыв *Панель управления*, выберите *Вид: Крупные значки*, затем нажмите на *Система*. В появившемся окне щелкните ссылку *Дополнительные параметры системы*, а затем нажмите кнопку *Переменные среды...*.

В **Windows 7** или **Vista:** Нажмите кнопку *Пуск* в левом нижнем углу экрана, наведите курсор на *Компьютер*, щелкните правой кнопкой мыши и выберите *Свойства* во всплывающем меню. Щелкните ссылку *Дополнительные параметры системы*, а затем нажмите кнопку *Переменные среды...*.

В **Windows XP:** Щелкните правой кнопкой мыши значок *Мой компьютер* на рабочем столе и выберите *Свойства*. Выберите вкладку *Дополнительно*, затем нажмите кнопку *Переменные среды...*.

Вызвав редактор переменных окружения, вы проделаете одни и те же действия, независимо от того, какая версия Windows у вас установлена. В разделе *Системные переменные* в нижней части редактора найдите переменную `PATH`. Если она есть, выберите ее и нажмите *Edit*. Полагая, *что* ваш корень Python — `C:\Python34`, добавьте эти две папки в свой путь (и убедитесь, что вы правильно расставили точки с запятой; между каждой папкой в списке должна быть точка с запятой):

```
C:\Python34
C:\Python34\Scripts
```

Примечание: если вы хотите дважды щелкнуть и запустить свои программы Python из папки Windows, чтобы при этом не исчезало окно консоли, вы можете добавить следующий код в нижнюю часть каждого скрипта:

```
#прекращает выход из консоли
 end_prog = ""
 while end_prog != "q":
        end_prog = input("введите q для выхода")
```

Интерактивный режим

Зайдите в IDLE (также называемый графическим интерфейсом Python). Вы должны увидеть окно с текстом, похожим на этот:

```
Python 3.0 (r30:67503, Dec 29 2008, 21:31:07)
[GCC 4.3.2 20081105 (Red Hat 4.3.2-7)] на linux2
Введите "copyright", "credits" или "license()" для получения
дополнительной информации.

**********************************************************************
Программное обеспечение персонального брандмауэра может предупредить
о соединении, которое IDLE устанавливает со своим подпроцессом,
```

```
используя внутренний интерфейс loopback этого компьютера. Это
соединение не видно ни на одном внешнем интерфейсе, и никакие данные
не отправляются в Интернет и не принимаются из него.
**********************************************************
IDLE 3.0
>>>
```

Символ >>> — это способ Python сообщить вам, что вы находитесь в интерактивном режиме. В интерактивном режиме то, что вы вводите, немедленно запускается. Попробуйте вести `1+1`. Python ответит `2`. Интерактивный режим позволяет протестировать и посмотреть, что будет делать Python. Если вам захочется «поиграть» с новыми операторами Python, перейдите в интерактивный режим и опробуйте их.

Создание и запуск программ

Зайдите в IDLE, если вы еще там не находитесь. В меню вверху выберите *File*, затем *New File*.

В появившемся окне введите следующее:

```
print("Hello, World!")
```

Теперь сохраните программу: выберите в меню *File*, затем *Save*. Сохраните ее под именем «`hello.py`» (вы можете сохранить ее в любой папке). Теперь, когда программа сохранена, ее можно запустить.

Затем запустите программу, перейдя в меню *Run*, затем *Run Module* (или, если у вас старая версия IDLE, используйте *Edit*, затем *Run Script*). Это выведет `Hello, World!` в окно **Python Shell**.

Более подробное введение в IDLE можно найти по адресу http://hkn.eecs.berkeley.edu/~dyoo/python/idle_intro/index.html.

Имена файлов программ

Очень полезно придерживаться некоторых правил относительно имен файлов программ на Python. Для самих программ эти правила не так важны, но если не соблюдать их для имен модулей (модули будут рассмотрены позже), могут возникнуть неожиданные проблемы.

1. Всегда сохраняйте программу с расширением `.py`. Не ставьте больше нигде точку в имени файла.
2. Для имен файлов используйте только стандартные символы: буквы, цифры, тире (-) и подчеркивание (_).
3. Пробелы (" ") не должны использоваться вообще (вместо них используйте подчеркивание).
4. Используйте в начале имени файла только буквы (не цифры!).
5. Не используйте «неанглийские» символы (такие как å, ƀ, ç, ð, é, õ, ü) в именах файлов — или, что еще лучше, вообще не используйте их при программировании.

Использование Python из командной строки

Если вы не хотите использовать Python из командной строки, это не обязательно, просто используйте IDLE. Чтобы перейти в интерактивный режим, просто введите `python3` без каких-либо аргументов. Чтобы запустить программу, создайте ее в текстовом редакторе (в Emacs есть хороший режим Python), а затем запустите ее с помощью команды `python3 имя_программы`. Чтобы использовать Python в Vim, вы можете посетить вики-страницу «Python в Vim» (http://wiki.python.org/moin/Vim).

Запуск программ на Python в *nix

Если вы используете Unix (Linux, Mac OS X или BSD), сделайте программу исполняемой с помощью `chmod`, а в качестве первой строки укажите:

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

Вы можете запустить программу Python с помощью `./hello.py`, как и любую другую команду.

Где получить помощь

На определенном этапе изучения вы, вероятно, «застрянете» и не будете знать, как решить проблему, над которой вам предстоит работать. В этом учебнике рассматриваются только основы программирования на Python, но вы можете найти в других источниках много дополнительной информации.

Документация по Python

Python очень хорошо документирован. Возможно, на вашем компьютере даже есть копии этих документов, которые поставляются вместе с инсталлятором Python:

- Официальный учебник Python 3 (http://docs.python.org/3/tutorial/) Гвидо ван Россума — хорошая отправная точка для решения общих вопросов.
- Для решения вопросов о стандартных модулях (о том, что это такое, вы узнаете позже) следует обратиться к справочнику по библиотекам Python 3 (http://docs.python.org/3/library/).
- Если вы действительно хотите разобраться в тонкостях языка, то Python 3 Reference Manual (http://docs.python.org/3/reference/) является исчерпывающим источником, но довольно сложным для начинающих.

Сообщество пользователей Python

Есть много других пользователей Python, и обычно они готовы помочь вам. Это очень активное сообщество пользователей организовано в основном через списки рассылки и группу новостей:

- Список рассылки Tutor (http://mail.python.org/mailman/listinfo/tutor) предназначен для тех, кто хочет задать вопросы о том, как научиться программировать на языке Python.

- Список рассылки python-help (http://www.python.org/community/lists/#python-help) — это служба поддержки python.org. Там вы можете задать свои вопросы по Python группе опытных «добровольцев».
- Группа новостей Python comp.lang.python (news:comp.lang.python) (архив Google groups (http://groups.google.com/group/comp.lang.python/)) является местом для общих обсуждений Python и центральным «местом встречи» сообщества.
- В Python wiki есть список локальных групп пользователей (http://wiki.python.org/moin/LocalUserGroups), вы можете присоединиться к списку рассылки группы и задавать вопросы. Вы также можете участвовать в собраниях групп пользователей.
- Сабреддит LearnPython (https://www.reddit.com/r/learnpython) — это еще одно место, где начинающие могут задать вопросы.

Чтобы не изобретать велосипед и не обсуждать одни и те же вопросы снова и снова, люди будут очень признательны, если вы *поищете решение своей проблемы в Интернете, прежде чем обращаться к этим спискам!*

Глава 3

Здравствуй, мир!

Что вы должны знать

Прочитав и освоив эту главу, вы должны знать, как редактировать программы в текстовом редакторе или IDLE, сохранять их на жестком диске и запускать после сохранения.

Вывод на экран

С незапамятных времен учебники по программированию начинались с маленькой программы под названием «Hello, World!» («Здравствуй, мир!»).

Итак, вот она:

```
print("Здравствуй, мир!")
```

Если вы используете командную строку для запуска программ, то наберите эту программу в текстовом редакторе, сохраните как `hello.py` и запустите с помощью `python3 hello.py`. В противном случае войдите в IDLE, создайте новое окно и создайте программу, как описано в разделе «Создание и запуск программ».

При запуске этой программы она выводит:

```
Здравствуй, мир!
```

Я не собираюсь напоминать вам об этом каждый раз, но когда я показываю вам программу, я рекомендую вам набрать ее и запустить. Я лучше учусь, когда набираю текст, и вы, вероятно, тоже.

А вот более сложная программа:

```
print("Джек и Джилл поднялись на холм,")
print("чтобы принести ведро воды;")
print("Джек упал и разбил свою корону,")
print("а Джилл полетела следом.")
```

Когда вы запускаете эту программу, она выводит сообщение:

```
Джек и Джилл поднялись на холм,
чтобы принести ведро воды;
Джек упал и сломал свою корону,
а Джилл полетела следом.
```

Когда компьютер запускает эту программу, он сначала видит строку:

```
print("Джек и Джилл поднялись на холм,")
```

...поэтому он выводит на экран:

```
Джек и Джилл поднялись на холм,
```

Затем компьютер переходит к следующей строке и видит:

```
print("чтобы принести ведро воды;")
```

Компьютер выполняет команду, переходит к следующей строке и продолжает выполнять команды, пока не дойдет до конца программы.

Терминология

Сейчас, вероятно, самое время объяснить вам, что происходит, и немного рассказать о терминологии программирования.

Выше мы использовали *функцию*, которая называется `print`. За именем функции — `print` — следуют круглые скобки, содержащие ноль или более аргументов. В данном примере:

```
print("Здравствуй, мир!")
```

...есть один аргумент, которым является `"Здравствуй, мир!"`. Обратите внимание, что этот аргумент представляет собой группу символов, заключенных в двойные кавычки (`""`). Это принято называть *строкой символов*, или просто *строкой* для краткости. Другой пример строки — `"Джек и Джилл поднялись на холм"`. Комбинация функции и круглых скобок с аргументами — это *вызов функции*.

Функция и ее аргументы — это один из типов утверждений, которые есть в Python, поэтому

```
print("Здравствуй, мир!")
```

...является примером оператора. В принципе, можно считать, что оператор — это одна строка в программе. Пожалуй, на сегодня это более чем исчерпывающая терминология.

Выражения

Вот еще одна программа:

```
print("2 + 2 равно", 2 + 2)
print("3 * 4 равно", 3 * 4)
print("100 - 1 равно", 100 - 1)
print("(33 + 2) / 5 + 11,5 равно", (33 + 2) / 5 + 11,5)
```

А вот *вывод* при запуске программы:

```
2 + 2 равно 4
3 * 4 равно 12
100 - 1 - 99
(33 + 2) / 5 + 11,5 равно 18,5
```

Как видите, Python может превратить ваш тысячедолларовый компьютер в пятидолларовый калькулятор.

Арифметические выражения

В этом примере за функцией `print` следуют два аргумента, причем каждый из них разделен запятой. Таким образом, в первой строке программы:

```
print("2 + 2 равно", 2 + 2)
```

...первым аргументом является строка `"2 + 2 равно"`, а вторым — арифметическое выражение `2 + 2`, которое является одним из видов *выражения*.

Важно отметить, что строка выводится как есть (без двойных кавычек), а *выражение оценивается* или преобразуется в свое фактическое значение.

В Python есть семь основных операций над числами:

Операция	Символ	Пример
Возведение в степень	**	5 ** 2 == 25
Умножение	*	2 * 3 == 6
Деление	/	14 / 3 == 4.666666666666667
Целочисленное деление	//	14 // 3 == 4
Остаток от деления	%	14 % 3 == 2
Сложение	+	1 + 2 == 3
Вычитание	-	4 - 3 == 1

Обратите внимание, что есть два способа деления: один возвращает повторяющуюся десятичную дробь, а другой может получить остаток и целое число. Порядок операций такой же, как и в математике:

- скобки ()
- возведение в степень **
- умножение *, деление /, целочисленное деление // и остаток %
- сложение + и вычитание -

Поэтому при необходимости используйте круглые скобки для структурирования формул.

Общение с людьми (и другими разумными существами)

Часто при программировании вы делаете что-то сложное и позже не можете вспомнить, что именно вы делали. В таких случаях программу следует прокомментировать. Комментарий — это записка для вас и других программистов, объясняющая, что происходит. Например:

```
# Не совсем PI, но достоверное моделирование
print(22 / 7)
```

Программа выводит:

```
3.14285714286
```

Обратите внимание, что комментарий начинается с хеша: #. Комментарии используются для общения с другими людьми, читающими программу, и с вашим будущим «я», чтобы прояснить сложные моменты.

За комментарием может следовать любой текст, а при выполнении программы текст после # и до конца строки игнорируется. Знак # не обязательно должен стоять в начале новой строки:

```
# Вывод PI на экран
print(22 / 7) # Ну, просто хорошая аппроксимация
```

Примеры

В каждой главе (со временем) будут приведены примеры функций программирования, представленных в этой главе. Вам следует хотя бы просмотреть их и убедиться, что вы их понимаете. Если не понимаете, возможно, вам захочется ввести их и посмотреть, что получится. Повозитесь с ними, измените их и посмотрите на результат.

Denmark.py

```
print("Прогнило что-то в Датском королевстве.")
print("              -- Шекспир")
```

Вывод:

```
Прогнило что-то в Датском королевстве.
              -- Шекспир
```

School.py

```
# Это не совсем верно за пределами США
# и основано на моих смутных воспоминаниях о юных годах.
print("Первый класс")
print("1 + 1 =", 1 + 1)
print("2 + 4 =", 2 + 4)
print("5 - 2 =", 5 - 2)
print()
print("Третий класс")
print("243 - 23 =", 243 - 23)
```

```
print("12 * 4 =", 12 * 4)
print("12 / 3 =", 12 / 3)
print("13 / 3 =", 13 // 3, "остаток", 13 % 3)
print() print("Средняя школа")
print("123.56 - 62.12 =", 123.56 - 62.12)
print("(4 + 3) * 2 =", (4 + 3) * 2)
print("4 + 3 * 2 =", 4 + 3 * 2)
print("3 ** 2 =", 3 ** 2)
```

Вывод:

```
Первый класс
1 + 1 = 2
2 + 4 = 6
5 - 2 = 3

Третий класс
243 - 23 = 220
12 * 4 = 48
12 / 3 = 4
13 / 3 = 4 остаток 1

Средняя школа
123.56 - 62.12 = 61.44
(4 + 3) * 2 = 14
4 + 3 * 2 = 10
3 ** 2 = 9
```

Упражнения

1. Напишите программу, которая выводит ваше полное имя и день рождения в виде отдельных строк.

2. Напишите программу, которая демонстрирует использование всех семи математических функций.

Решение 1-го задания:

```
print("Ада Лавлейс", "родилась", "27 ноября 1852 года")
print("Альберт Эйнштейн", "родился", "14 марта 1879 года")
print("Джон Смит", "родился", "14 марта 1879 года")
```

Решение 2-го задания:

```
print("5**5 = ", 5**5)
print("6*7 = ", 6*7)
print("56/8 = ", 56/8)
print("14//6 = ", 14//6)
print("14%6 = ", 14%6)
print("5+6 = ", 5+6)
print("9-0 = ", 9-0)
```

Глава 4

Кто идет?

Ввод и переменные

Теперь я думаю, что настало время для действительно сложной программы. Вот она:

```
print("Стой!")
user_input = input("Кто идет?")
print("Вы можете пройти", user_input)
```

Когда я запустил ее, вот что появилось на моем экране:

```
Стой!
Кто идет? Джош
Ты можешь пройти, Джош.
```

Примечание: после выполнения кода по нажатии *F5* оболочка Python будет выдавать только вывод:

```
Стой!
Кто идет?
```

Вам нужно ввести свое имя в оболочке Python, а затем нажать клавишу *Enter* для остального вывода. Конечно, когда вы запустите программу, ваш экран будет выглядеть иначе из-за оператора `input()`.

Когда вы запускали программу, вы, вероятно, заметили (вы ведь запускали программу, верно?), что вам пришлось ввести ваше имя, а затем нажать *Enter*. После этого программа вывела еще текст, а также ваше имя. Это пример *ввода*. Программа достигает определенного момента, а затем ждет, пока пользователь введет некоторые данные, которые она сможет использовать в дальнейшем.

Конечно, получение информации от пользователя было бы бесполезным, если бы нам не было куда ее поместить, и здесь на помощь приходят переменные. В предыдущей программе `user_input` — это *переменная*. Переменные похожи на ячейки, в которых можно хранить некоторые данные. Вот программа, демонстрирующая примеры переменных:

```
a = 123.4
b23 = 'Spam'
first_name = "Билл"
b = 432
c = a + b
print("a + b равно",c)
print("имя - ",first_name)
print("Sorted Parts, After Midnight or", b23)
```

И вот результат:

```
a + b равно 555,4
имя - Билл
Sorted Parts, After Midnight or Spam
```

Переменные хранят данные. В приведенной выше программе переменными являются a, b23, first_name, b и c. Два основных типа переменных — это *строки* и *числа*. Строки — это последовательность букв, цифр и других символов. В этом примере b23 и first_name — переменные, в которых хранятся строки. Spam, Билл, a + b равно, имя - и Sorted Parts, After Midnight or являются строками в этой программе. Символы окружены символами " или '.

Другой тип переменных — это *числа*. Помните, что переменные используются для хранения значения, в них не применяются кавычки (" и '). Если вы хотите использовать фактическое *значение*, необходимо использовать *кавычки*.

```
value1 == Pim
value2 == "Pim"
```

Оба выглядят одинаково, но в первом случае Python проверяет, совпадает ли значение, хранящееся в переменной value1, со значением, хранящимся в *переменной* Pim. Во втором случае Python проверяет, совпадает ли строка (собственно, буквы P, i и m) со значением value2 (далее вы узнаете больше о строках и о символе ==).

Присвоение

Итак, у нас есть ячейки, называемые переменными, а также данные, которые могут попасть в переменную. Компьютер увидит строку типа first_name = "Bill" и прочитает ее как «Поместите строку Bill в ячейку (или переменную) first_name». Позже он увидит утверждение c = a + b и прочитает его как «поместите в c сумму a + b или 123,4 + 432, что равно 555,4». Правая часть оператора (a + b) оценивается, и результат записывается в переменную в левой части (c). Это называется присваиванием, и не стоит путать знак равенства присваивания (=) с «равенством» в математическом смысле (именно для этого == будет использоваться позже).

Вот еще один пример использования переменных:

```
a = 1
print(a)
a = a + 1
print(a)
a = a * 2
print(a)
```

И вот результат:

```
1
2
4
```

Даже если одна и та же переменная оказывается по обе стороны знака равенства (например, `spam = spam`), компьютер все равно читает это так: «Сначала найдите данные для хранения, а затем найдите, куда эти данные девать».

Еще одна программа, прежде чем я закончу эту главу:

```
number = float(input("Введите число: "))
integer = int(input("Введите целое число: "))
text = input("Введите строку: ")
print("number =", number)
print("число - это", type(number))
print("число * 2 =", number * 2)
print("целое число =", integer)
print("целое число - это", type(integer))
print("целое число * 2 =", integer * 2)
print("текст =", text)
print("текст - это", type(text))
print("текст * 2 =", text * 2)
```

В результате я получил следующее:

```
Введите число: 12.34
Введите целое число: -3
Введите строку: Hello
число = 12.34
число - это <class 'float'>
число * 2 = 24,68
целое число = -3
Целое число - это <class 'int'>
целое число * 2 = -6
текст = Hello
текст - это <class 'str'>
текст * 2 = HelloHello
```

Обратите внимание, что число было создано с помощью `float(input())`, а текст — с помощью `input()`. `input()` возвращает строку, а функция `float` возвращает число из строки. `int` возвращает целое число, то есть число без десятичной точки. Если вы хотите, чтобы пользователь ввел десятичную дробь, используйте `float(input())`, если вы хотите, чтобы пользователь ввел целое число, используйте `int(input())`, а если вы хотите, чтобы пользователь ввел строку, используйте `input()`.

Во второй половине программы используется функция `type()`, которая сообщает, к какому типу относится переменная. Числа имеют тип `int` или `float`, что является сокращением от *integer* и *floating point* (в основном используется для десятичных чисел) соответственно. Текстовые строки имеют тип `str`, сокращение от *string*. С целыми и плавающими числами можно работать с помощью математических функций, а со строками — нет. Обратите внимание, что когда Python умножает число на целое число, происходит ожидаемое. Однако когда строка умножается на целое число, в результате получается несколько копий строки (например, `text * 2 = HelloHello`).

Операции со строками выполняются иначе, чем операции с числами. Кроме того, некоторые операции работают только с числами (как целыми, так и с числами с плавающей точкой) и выдают ошибку, если используется строка. Вот несколько примеров в интерактивном режиме:

```
>>> print("Это" + "и" + "это" + "соединено.")
Это и это соединено.
>>> print("Ха, " * 5)
Ха, Ха, Ха, Ха, Ха,
>>> print("Ха, " * 5 + "ха!")
Ха, Ха, Ха, Ха, Ха, ха!
>>> print(3 - 1)
2
>>> print("3" - "1")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'str'
>>>
```

Вот список некоторых операций со строками:

Операция	Символ	Пример
Повторение	*	"i" * 5 == "iiiii"
Конкатенация	+	"Hello, " + "World!" == "Hello, World!"

Примеры

Rate_times.py

```
# Эта программа вычисляет скорость и расстояние.
print("Введите скорость (rate) и расстояние (distance)")
rate = float(input("Скорость: "))
distance = float(input("Расстояние: "))
print("Время:", (расстояние / скорость))
```

Пробные запуски:

```
Введите скорость (rate) и расстояние (distance)
Скорость: 5
Расстояние: 10
Время: 2.0

Скорость: 3.52
Расстояние: 45.6
Время: 12.9545454545
```

Area.py

```
# Эта программа вычисляет периметр и площадь прямоугольника
print("Вычислить информацию о прямоугольнике")
```

```
length = float(input("Длина: "))
width = float(input("Ширина: "))
print("Площадь:", length * width)
print("Периметр:", 2 * length + 2 * width)
```

Пробные запуски:

```
Вычислите информацию о прямоугольнике
Длина: 4
Ширина: 3
Площадь: 12.0
Периметр: 14.0

Длина: 2.53
Ширина: 5.2
Площадь: 13.156
Периметр: 15.46
```

Temperature.py

```
# Эта программа преобразует шкалу Фаренгейта в шкалу Цельсия.
fahr_temp = float(input("Температура по Фаренгейту: "))
print("Температура по Цельсию:", (fahr_temp - 32.0) * 5.0 / 9.0)
```

Пробные запуски:

```
Температура по Фаренгейту: 32
Температура по Цельсию: 0.0

Температура по Фаренгейту: -40
Температура по Цельсию: -40.0

Температура по Фаренгейту: 212
Температура по Цельсию: 100.0


Температура по Фаренгейту: 98,6
Температура по Цельсию: 37.0
```

Упражнение

Напишите программу, которая получает от пользователя две строковые переменные и две числовые переменные, конкатенирует (соединяет их вместе без пробелов) и выводит строки на экран, а затем перемножает два числа на новой строке.

Решение:

```
string1 = input('Строка 1: ')
string2 = input('Строка 2: ')
float1 = float(input('Число 1: '))
float2 = float(input('Строка 2: '))
print(string1 + string2)
print(float1 * float2)
```

Глава 5

Сосчитайте до 10

Циклы While

Представляем нашу первую *управляющую структуру*. Обычно компьютер начинает с первой строки, а затем спускается вниз. Управляющие структуры изменяют порядок выполнения операторов или решают, будет ли выполняться определенный оператор. Вот исходный текст программы, в которой используется управляющая структура `while`:

```
a = 0           # СНАЧАЛА установите начальное значение переменной a
                # равным 0 (нулю).
while a < 10: # Пока значение переменной a меньше 10, делайте:
    a = a + 1 # Увеличивайте значение переменной a на 1: a = a + 1!
    print(a)  # Выведите на экран текущее значение переменной a.
                # ПОВТОРЯЙТЕ до тех пор, пока значение переменной a не
                # станет равным 9!? См. примечание.

                # ПРИМЕЧАНИЕ:
                # Значение переменной a увеличится на 1
                # с каждым повтором или циклом 'БЛОКА оператора while'.
                # например, a = 1, затем a = 2, затем a = 3 и т. д.
                # до a = 9, затем...
                # код завершит прибавление 1 к a (теперь a = 10), выведет
                # результат, а затем выйдет из 'БЛОКА оператора while'.
                #               --
                #  While a < 10: |
                #       a = a + 1 |<--[ БЛОК оператора while ]
                #       print (a) |
                #               --
```

И вот наш интересный результат (цифры будут выведены в столбик):

```
1 2 3 4 5 6 7 8 9 10
```

(А вы думали, что после превращения вашего компьютера в пятидолларовый калькулятор хуже уже быть не может?)

Что же делает программа? Сначала она видит строку `a = 0` и устанавливает `a` в ноль. Затем она видит `while a < 10:`, и таким образом компьютер проверяет, что `a < 10`. Когда компьютер впервые видит это утверждение, `a` равно нулю, поэтому оно меньше 10. Другими словами, пока `a` меньше десяти, компьютер будет выполнять вложенные операторы. В итоге `a` станет равным десяти (путем добавления единицы к `a` снова и снова), и выражение `while a < 10` перестанет быть истинным. Достигнув этой точки, программа прекратит выполнение строк с отступами.

Всегда помните, что в конце строки оператора `while` нужно ставить двоеточие «:»! Вот еще один пример использования `while`:

```
a = 1
s = 0
print('Введите числа для добавления к сумме.')
print('Введите 0, чтобы выйти.')
while a != 0:
    print('Текущая сумма:', s)
    a = float(input('Число? '))
    s = s + a
print('Общая сумма =', s)


Введите числа для добавления к сумме.
Введите 0, чтобы выйти.
Текущая сумма: 0
Число? 200
Текущая сумма: 200.0
Число? -15.25
Текущая сумма: 184,75
Число? -151.85
Текущая сумма: 32,9
Число? 10.00
Текущая сумма: 42,9
Число? 0
Общая сумма = 42,9
```

Обратите внимание, что `print('Общая сумма =', s)` выполняется только в конце. Оператор `while` воздействует только на строки, отступы которых содержат пробельные символы. Оператор `!=` означает «не равно», поэтому `while a != 0:` означает, что до тех пор, пока `a` не равно нулю, выполняются следующие за ним операторы с табуляцией.

Обратите внимание, что `a` — это число с плавающей точкой, но не все числа с плавающей точкой могут быть точно представлены, поэтому использование `!=` для них иногда может не сработать. Попробуйте ввести 1.1 в интерактивном режиме.

Бесконечные циклы

Теперь, когда у нас есть циклы `while`, можно создавать программы, которые выполняются вечно. Простой способ сделать это — написать программу, подобную этой:

```
while 1 == 1:
    print("Помогите, я застрял в цикле")
```

Оператор «==» используется для проверки равенства выражений по обе стороны от оператора, так же как «<» раньше использовался для обозначения «меньше, чем» (полный список всех операторов сравнения вы получите в следующей главе).

Эта программа будет выводить `Помогите, я застрял в цикле` до тепловой смерти Вселенной или пока вы не остановите ее, потому что 1 всегда будет равна 1. Способ остановить ее — нажать кнопку Control (или *Ctrl*) и *C* (букву) одновременно. Это приведет к остановке программы. (**Примечание:** иногда после Control-C нужно нажать Enter.) На некоторых системах ничто не остановит программу, кроме «убийства» процесса, так что избегайте этого!

Примеры

Последовательность Фибоначчи

Fibonacci-method1.py

```
# Эта программа вычисляет последовательность Фибоначчи.
a = 0
b = 1
count = 0
max_count = 20

while count < max_count:
    count = count + 1
    print(a, end=" ")# Обратите внимание на магическое end=" "
                     # в аргументах функции print,
    # которое не позволяет ей создавать новую строку.
    old_a = a # нам нужно отслеживать a с тех пор, как мы его изменили.
    a = b
    b = old_a + b
print() # получает новую (пустую) строку.
```

Вывод:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

Обратите внимание, что вывод осуществляется в одну строку из-за дополнительного аргумента `end=" "` в аргументах оператора `print`.

Fibonacci-method2.py

```
# Упрощенный и ускоренный метод вычисления последовательности Фибоначчи
a = 0
b = 1
count = 0
max_count = 10

while count < max_count:
    count = count + 1
    print(a, b, end=" ") # Обратите внимание на магическое end=" "
    a = a + b
    b = a + b
```

```
print() # получает новую (пустую) строку.
```

Вывод:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

Fibonacci-method3.py

```
a = 0
b = 1
count = 0
maxcount = 20

#как только цикл запущен, мы остаемся в нем

while count < maxcount:
    count += 1
    olda = a
    a = a + b
    b = olda
    print(olda,end=" ")
print()
```

Вывод:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

Введите пароль

Password.py

```
# Ждет, пока не будет введен пароль. Используйте Control-C, чтобы
# выйти из игры без пароля

#Обратите внимание, что это не должен быть пароль, чтобы цикл
# while выполнялся хотя бы один раз.
password = str()

# обратите внимание, что != означает "не равно"
while password != "unicorn":
    password = input("Пароль: ")
print("Добро пожаловать")
```

Пробный запуск:

```
Пароль: auo
Пароль: y22
Пароль: password
Пароль: open sesame
Пароль: unicorn
Добро пожаловать в систему
```

Упражнение

Напишите программу, которая запрашивает у пользователя имя пользователя и пароль. Затем, когда пользователь набирает «lock», он должен ввести свое имя и пароль, чтобы разблокировать программу.

Решение:

```
name = input("What is your UserName: ")
password = input("What is your Password: ")
print("Чтобы заблокировать компьютер, введите lock.")
command = None
input1 = None
input2 = None
while command != "lock":
    command = input("Какая у вас команда: ")
while input1 != name:
    input1 = input("Какое у вас имя пользователя: ")
while input2 != password:
    input2 = input("Какой у вас пароль: ")
print("Добро пожаловать в вашу систему!")
```

Если вы хотите, чтобы программа выполнялась непрерывно, просто добавьте цикл `while 1 == 1:` вокруг всей программы. Вам придется сделать отступ для остальной части программы, когда вы добавите это в верхней части кода, но не волнуйтесь, вам не придется делать это вручную для каждой строки! Просто выделите все, что вы хотите отступить, и нажмите на «Отступ» в разделе «Формат» в верхней панели окна Python.

Другой способ сделать это может быть следующим:

```
name = input('Задайте имя: ')
password = input('Задайте пароль: ')
while 1 == 1:
    nameguess=""
    passwordguess=""
    key=""
    while (nameguess != name) or (passwordguess != password):
        nameguess = input('Name? ')
        passwordguess = input('Пароль? ')
    print("Добро пожаловать,", name, ". Введите lock, чтобы
заблокировать.")
    while key != "lock":
        key = input("")
```

Обратите внимание на оператор `or` в `while (nameguess != name) or (passwordguess != password)`, который мы еще не рассматривали. Вы, вероятно, сможете понять, как это работает.

Глава 6

Решения

Оператор if

Как всегда, я считаю, что каждую главу следует начинать с разминочного упражнения по набору текста, поэтому здесь представлена короткая программа для вычисления абсолютного значения целого числа:

```
n = int(input("Число? "))
if n < 0:
    print("Абсолютное значение", n, "равно", -n)
else:
    print("Абсолютное значение", n, "равно", n)
```

Вот результаты двух запусков этой программы:

```
Число? -34
Абсолютное значение -34 равно 34

Номер? 1
Абсолютное значение 1 равно 1
```

Что же делает компьютер, когда видит этот кусок кода? Сначала он запрашивает у пользователя число с помощью утверждения «`n = int(input("Число? "))`». Затем он читает строку «`if n < 0:`». Если `n` меньше нуля, Python запускает строку «`print("The absolute value of", n, "is", -n)`». В противном случае выполняется строка «`print("Абсолютное значение", n, "равно", n)`».

Более формально, Python смотрит, является ли *выражение* `n < 0` истинным или ложным. За оператором `if` следует блок операторов с отступами, которые выполняются, когда выражение истинно. По желанию после оператора `if` следуют оператор `else` и еще один блок операторов с отступом. Этот второй блок операторов выполняется, если выражение ложно.

Существует несколько различных проверок (тестов), которые могут быть у выражения. Здесь приведена таблица всех этих проверок:

оператор	функция
`<`	менее
`<=`	меньше или равно
`>`	больше, чем
`>=`	больше или равно
`==`	равно
`!=`	не равно

Еще одна особенность команды `if` — оператор `elif`. Он расшифровывается как else if и означает, что если исходное выражение `if` ложно, а часть `elif` истинна, то выполняется часть `elif`. А если ни выражение `if`, ни `elif` не являются истинными, то следует выполнить то, что указано в блоке `else`.

Вот пример:

```
a = 0
while a < 10:
    a = a + 1
    if a > 5:
        print(a, ">", 5)
    elif a <= 3:
        print(a, "<=", 3)
    else:
        print("Ни один из тестов не был верен")
```

...и вывод:

```
1 <= 3
2 <= 3
3 <= 3
Ни один из тестов не был верен
Ни один из тестов не был верен
6 > 5
7 > 5
8 > 5
9 > 5
10 > 5
```

Обратите внимание, что `elif a <= 3` проверяется только в том случае, если оператор `if` не является истинным. Выражений `elif` может быть несколько, что позволяет проводить несколько проверок в одном операторе `if`.

Примеры

```
# Эта программа демонстрирует использование оператора ==
# с использованием чисел
print(5 == 6)
# Использование переменных
x = 5
y = 8
print(x == y)
```

И вывод:

```
False
False
```

high_low.py

```
# Играет в угадайку - больше или меньше.

# Это должно быть что-то полуслучайное, например
# последние цифры времени или что-то еще, но это
# подождет до следующей главы.
# (Дополнительно модифицируйте его так, чтобы он был случайным
# после главы "Модули")
number = 7
guess = -1

print("Угадайте число!")
while guess != number:
    guess = int(input("Это... "))

    if guess == number:
        print("Ура! Вы угадали правильно!")
    elif guess < number:
        print("Оно больше...")
    elif guess > number:
        print("Оно не такое уж и большое.")
```

Пробный запуск:

```
Угадайте число!
Это... 2
Оно больше...
Это... 5
Оно больше...
Это... 10
Оно не такое уж и большое.
Это... 7
Ура! Вы угадали правильно!
```

even.py

```
# Запрашивает число.
# Выводит, четное оно или нечетное

number = float(input("Назовите число: "))
if number % 2 == 0:
    print(int(number), "- четное")
elif number % 2 == 1:
    print(int(number), "- нечетное")
else:
    print(number, "- очень странно")
```

Пробные запуски:

```
Назовите число: 3
3 - нечетное
Назовите число: 2
2 - четное
Назовите число: 3.4895
3.4895 - очень странно
```

average1.py

```
# продолжает запрашивать числа, пока не будет введено 0.
# Выводит среднее значение.

count = 0
sum = 0.0
number = 1 # установите значение, которое не приведет к немедленному
завершению цикла while.

print("Введите 0, чтобы выйти из цикла")

while number != 0:
    number = float(input("Введите число: "))
    if number != 0:
        count = count + 1
        sum = sum + number
    if number == 0:
        print("Среднее значение составило:", sum / count)
```

Пробные запуски:

```
Введите 0, чтобы выйти из цикла
Введите число: 3
Введите число: 5
Введите число: 0
Среднее значение составило: 4.0

Введите 0, чтобы выйти из цикла
Введите число: 1
Введите число: 4
Введите число: 3
Введите число: 0
Среднее значение составило: 2.66666666667
```

average2.py

```
# продолжает запрашивать числа до тех пор, пока не будут введены все
числа.
```

```
# Выводит среднее значение.

# Обратите внимание, что мы используем целое число для отслеживания
количества чисел,
# а числа с плавающей точкой для ввода каждого числа.
sum = 0,0
print("Эта программа возьмет несколько чисел и усреднит их")
count = int(input("Сколько чисел вы хотите усреднить: "))
current_count = 0

while current_count < count:
    current_count = current_count + 1
    print("Число", current_count)
    number = float(input("Введите число: "))
    sum = sum + number

print("Среднее значение составило:", sum / count)
```

Пробные запуски:

```
Эта программа возьмет несколько чисел и усреднит их
Сколько чисел вы хотите усреднить: 2
Число 1
Введите число: 3
Число 2
Введите число: 5
Среднее значение составило: 4.0

Эта программа возьмет несколько чисел и усреднит их
Сколько чисел вы хотите усреднить: 3
Число 1
Введите число: 1
Число 2
Введите число: 4
Число 3
Введите число: 3
Среднее значение составило: 2.66666666667
```

Упражнения

Напишите программу, которая спрашивает у пользователя его имя, если он вводит ваше имя, скажите ему «Это хорошее имя», если он вводит «Джон Клиз» или «Майкл Пэлин», скажите ему, что вы о них думаете ;), в противном случае скажите ему «У вас красивое имя».

Решение:

```
name = input('Ваше имя: ')
```

```
if name == 'Bryn':
    print('Это хорошее имя').
elif name == 'John Cleese':
    print('... немного смешного текста').
elif name == 'Michael Palin':
    print('... немного смешного текста').
else:
    print('У вас красивое имя').
```

Модифицируйте программу «угадай число» из этого раздела, чтобы отслеживать, сколько раз пользователь ввел неправильное число. Если больше трех раз, то в конце выведите «Должно быть, это было сложно.», в противном случае выведите «Хорошая работа!».

Решение:

```
number = 7
guess = -1
count = 0

print("Угадайте число!")
while guess != number:
    guess = int(input("Это... "))
    count = count + 1
    if guess == number:
        print("Ура! Вы угадали правильно!")
    elif guess < number:
        print("Оно больше...")
    elif guess > number:
        print("Оно не такое уж и большое").

if count > 3:
    print("Должно быть, это было сложно.")
else:
    print("Хорошая работа!")
```

Напишите программу, которая запрашивает два числа. Если сумма чисел больше 100, выведите «Это большое число».

Решение:

```
number1 = float(input('1-е число: '))
number2 = float(input('2-е число: '))
if number1 + number2 > 100:
    print('Это большое число').
```

Глава 7

Отладка

Что такое отладка?

«Как только мы начали программировать, мы с удивлением обнаружили, что создавать правильные программы не так просто, как мы думали. Пришлось открыть для себя отладку. Я помню тот самый момент, когда я понял, что с этого момента большая часть моей жизни будет посвящена поиску ошибок в моих собственных программах».

Морис Уилкс открывает для себя отладку, 1949 год

Если вы уже возились с программами, то наверняка обнаружили, что иногда программа делает то, чего вы от нее не хотели. Это довольно частое явление. Отладка — это процесс выяснения того, что делает компьютер, и последующего принуждения его делать то, что хотите вы. Это может быть непросто. Однажды я потратил почти неделю на поиск и исправление ошибки, которая была вызвана тем, что кто-то поставил `x` там, где должен был быть `y`.

Эта глава будет более абстрактной, чем предыдущие.

Что должна делать программа?

Первое, что нужно сделать (это звучит очевидно), — выяснить, что должна делать программа, если она работает правильно. Придумайте несколько тестовых примеров и посмотрите, что произойдет.

Допустим, у меня есть программа для вычисления периметра прямоугольника (сумма длин сторон). У меня есть следующие тестовые случаи:

высота	ширина	периметр
3	4	14
2	3	10
4	4	16
2	2	8
5	1	12

Теперь я запускаю свою программу на всех тестовых примерах и смотрю, делает ли она то, чего я от нее ожидаю. Если нет, то нужно выяснить, что делает компьютер. Чаще всего некоторые из тестовых примеров работают, а некоторые нет. В этом случае вам следует попытаться выяснить, что общего у работающих примеров. Например, вот вывод для программы периметра (код вы увидите через минуту):

```
Высота: 3
Ширина: 4
периметр = 15

Высота: 2
Ширина: 3
периметр = 11

Высота: 4
Ширина: 4
периметр = 16

Высота: 2
Ширина: 2
периметр = 8

Высота: 5
Ширина: 1
периметр = 8
```

Обратите внимание, что он не сработал для первых двух входов, сработал для следующих двух и не сработал для последнего. Попробуйте выяснить, что общего между работающими входами. Как только вы получите представление о том, в чем проблема, найти причину будет проще. При работе с собственными программами вы должны попробовать больше тестовых случаев, если они вам нужны.

Что делает программа?

Следующее, что нужно сделать, это просмотреть исходный код. Одна из самых важных вещей, которые необходимо делать при программировании, — чтение исходного кода. Основной способ сделать это — прохождение кода.

Прохождение кода начинается с первой строки и продолжается до тех пор, пока программа не будет завершена. Циклы `while` и операторы `if` означают, что некоторые строки могут не выполняться никогда, а некоторые выполняются много раз. В каждой строке вы выясняете, что сделал Python. Начнем с простой программы для работы с периметром. Не вводите ее, вы будете ее читать, а не запускать. Исходный код:

```
height = int(input("Высота: "))
width = int(input("Ширина: "))
print("Периметр =", width + height + width + width)
```

***Вопрос:* какую строку Python запускает первой?**

Ответ: первая строка всегда выполняется первой. В данном случае это:

```
height = int(input("Высота: "))
```

Что делает эта строка?

Печатает `Высота:` , ждет, пока пользователь введет данные, а затем преобразует их в целочисленную переменную `height`.

Какая строка будет следующей?

В общем, это следующая строка, которая выглядит так:

```
width = int(input("Ширина: "))
```

Что делает эта линия?

Печатает `Ширина:` , ждет, пока пользователь введет число, и помещает то, что он ввел, в переменную ширины `width`.

Какая строка будет следующей?

Если следующая строка не имеет отступа больше или меньше, чем текущая, то это строка, расположенная сразу за ней, так что это выглядит так: `print("Периметр = ", width + height + width + width)` (также может быть запущена функция в текущей строке, но об этом в следующей главе).

Что делает эта линия?

Сначала выводится `Периметр =` , затем сумма значений, содержащихся в переменных, `width` и `height`, от `width + height + width + width`.

Правильно ли вычисляется периметр по схеме `width + height + width + width`**?**

Давайте посмотрим, периметр прямоугольника — это нижняя часть (ширина) плюс левая сторона (высота) плюс верхняя часть (ширина) плюс правая сторона. Последним элементом должна быть длина правой стороны, или высота (`height`).

Понимаете ли вы, почему в некоторых случаях периметр был рассчитан «правильно»?

Он вычислялся правильно, когда ширина и высота были равны.

Следующая программа, которую мы рассмотрим, — это программа, которая должна вывести на экран пять точек. Однако программа выводит:

```
. . . .
```

А вот и программа:

```
number = 5
while number > 1:
    print(".",end=" ")
    number = number - 1
print()
```

Эта программа будет более сложной, поскольку в ней теперь есть отступы (или управляющие структуры). Давайте начнем.

Какую строку нужно запустить первой?

Первая строка файла: `number = 5`

Что она делает?

Помещает число 5 в переменную `number`.

Какая следующая строка?

Следующая строка: `while number > 1:`

Что она делает?

Операторы `while` в общем случае смотрят на свое выражение, и если оно истинно, то выполняют следующий блок кода с отступом, в противном случае пропускают следующий блок кода с отступом.

Что же оператор while делает прямо сейчас?

Если `number > 1` истинно, то будут запущены следующие две строки.

Так `number > 1`**?**

Последнее значение, помещенное в `number`, было `5`, а `5 > 1`, так что да.

Какова же следующая строка?

Так как `while` был истинным, следующая строка будет: `print(".",end="" )`

Что делает эта строка?

Печатает одну точку, и поскольку дополнительный аргумент `end=" "` существует, следующий напечатанный текст не будет находиться на другой строке экрана.

Какова следующая строка?

`number = number - 1`, так как это следующая строка и отступы не меняются.

Что она делает?

Она вычисляет `number - 1`, которое является текущим значением `number` (или 5), вычитает из него 1 и делает это новым значением числа. То есть, по сути, он меняет значение числа (`number`) с 5 на 4.

Какова следующая строка?

Уровень отступа уменьшается, поэтому нужно посмотреть, что это за управляющая структура. Это цикл `while`, поэтому нам придется вернуться к строке `while`, в которой `while number > 1`:

Что она делает?

Она «смотрит» на значение числа, которое равно 4, и сравнивает его с 1, а поскольку `4 > 1`, цикл `while` продолжается.

Какая следующая строка?

Поскольку цикл был истинным, следующая строка: `print(".",end=" ")`

Что она делает?

Она печатает вторую точку в строке, заканчивающуюся пробелом.

Какая следующая строка?

Отступы не меняются, так что получается: `number = number - 1`.

И что она делает?

Она берет текущее значение числа (4), вычитает из него 1, что дает 3, и, наконец, делает 3 новым значением числа.

Какова следующая строка?

Поскольку отступ изменился из-за окончания цикла `while`, следующей строкой будет: `while number > 1:`

Что она делает?

Сравнивает текущее значение числа (3) с 1. `3 > 1`, поэтому цикл `while` продолжается.

Какова следующая строка?

Поскольку условие цикла `while` было истинным, следующая строка будет выглядеть так: `print(".",end=" ")`

И что она делает?

В строке печатается третья точка.

Какова следующая строка?

Это: `number = число - 1`

Что она делает?

Она берет текущее значение числа (3), вычитает из него 1 и делает 2 новым значением числа.

Какова следующая строка?

Вернитесь к началу цикла `while`: `while number > 1:`

Что она делает?

Сравнивает текущее значение числа (2) с 1. Поскольку `2 > 1`, цикл `while` продолжается.

Какая следующая строка?

Поскольку цикл `while` продолжается: `print(".",end=" ")`

Что она делает?

Она открывает смысл жизни, Вселенной и всего остального. Шутка. (Я должен был убедиться, что вы не спите). Она выводит 4-ю точку на экране.

Какова следующая строка?

Это: `number = number - 1`

Что она делает?

Берет текущее значение числа (2), вычитает 1 и делает 1 новым значением числа.

Какова следующая строка?

Вернитесь к циклу `while`: `while number > 1:`

Что делает строка?

Сравнивает текущее значение числа (1) с 1. Поскольку `1 > 1` ложно (единица не больше единицы), цикл `while` завершается.

Какова следующая строка?

Поскольку условие цикла `while` было ложным, следующей строкой будет строка после выхода из цикла `while`, или: `print()`

Что делает эта строка?

Заставляет экран перейти к следующей строке.

Почему программа не печатает 5 точек?

Цикл завершается на 1 точку ранее.

Как мы можем это исправить?

Сделайте выход из цикла на 1 точку позже.

И как нам это сделать?

Есть несколько способов. Один из них — изменить цикл `while` на: `while number > 0:` Другой — изменить условие на: `number >= 1` Есть и другие.

Как исправить мою программу?

Вам нужно понять, что делает программа. Вам нужно понять, что должна делать программа. Поймите, в чем разница между этими понятиями. Отладка — это навык, который нужно практиковать, чтобы научиться находить ошибки в программе. Если вы не можете разобраться с проблемой в течение часа, сделайте перерыв, поговорите с кем-нибудь о проблеме или поразмышляйте о чем-то другом. Вернитесь через некоторое время, и у вас наверняка появятся новые идеи. Удачи.

Глава 8

Определение функций

Создание функций

В начале этой главы я приведу пример того, что можно, но не нужно делать (поэтому не набирайте этот код):

```
a = 23
b = -23

if a < 0:
    a = -a
if b < 0:
    b = -b
if a == b:
    print("Абсолютные значения", a, "и", b, "равны").
else:
    print("Абсолютные значения", a, "и", b, "различны").
```

...с выводом:

```
Абсолютные значения 23 и 23 равны.
```

Программа кажется немного повторяющейся. Программисты ненавидят повторяться — в конце концов, для этого и существуют компьютеры! (Заметьте также, что нахождение абсолютного значения изменило значение переменной, поэтому она выводит 23, а не -23). К счастью, Python позволяет создавать функции для удаления дублирования. Вот переписанный пример:

```
a = 23
b = -23

def absolute_value(n):
    if n < 0:
        n = -n
    return n

if absolute_value(a) == absolute_value(b):
    print("Абсолютные значения", a, "и", b, "равны").
else:
    print("Абсолютные значения", a, "и", b, "различны").
```

...с выводом:

```
Абсолютные значения 23 и -23 равны.
```

Ключевой особенностью этой программы является оператор `def`. Этот оператор `def` (сокращение от define) начинает определение функции. За `def` следует имя функции `absolute_value`. Далее идет '(', за которой следует параметр `n` (`n` передается из программы в функцию при ее вызове). Утверждения после ':' выполняются при использовании функции. Выражения продолжаются до тех пор, пока не закончится отступ или не будет встречен оператор `return`. Оператор `return` возвращает значение туда, откуда была вызвана функция. В нашей самой первой программе мы уже встречали функцию `print`. Теперь мы можем создавать новые функции.

Обратите внимание, что значения `a` и `b` не изменились. Функции можно использовать для повторения задач, которые не возвращают значения. Вот несколько примеров:

```
def hello():
    print("Здравствуйте")

def area(width, height):
    return width * height

def print_welcome(name):
    print("Добро пожаловать,", name)

hello()
hello()

print_welcome("Фред")
w = 4
h = 5
print("ширина =", w, " высота =", h, " площадь =", area(w, h))
```

...с выводом:

```
Здравствуйте
Здравствуйте
Добро пожаловать, Фред.
ширина = 4 высота = 5 площадь = 20
```

Этот пример показывает, что еще можно делать с функциями. Обратите внимание, что вы можете использовать как отсутствие аргументов, так и два аргумента или более. Заметьте также, что функции не нужно отправлять обратно значение, возврат необязателен.

Переменные в функциях

При устранении повторяющегося кода в нем часто встречаются переменные. В Python они обрабатываются особым образом. До сих пор все переменные, которые мы видели, были глобальными. В функциях есть особый тип переменных, называемых локальными переменными.

Эти переменные существуют только во время работы функции. Если локальная переменная имеет то же имя, что и другая переменная (например, глобальная), то локальная переменная скрывает другую. Звучит непонятно? Что же, следующие примеры (немного надуманные) должны помочь прояснить ситуацию.

```
a = 4

def print_func():
    a = 17
    print("in print_func a =", a)

print_func()
print("a = ", a)
```

При запуске мы получим результат:

```
 in print_func a = 17
a = 4
```

Назначения переменных внутри функции не отменяют глобальные переменные, они существуют только внутри функции. Даже если a было присвоено новое значение внутри функции, это новое значение имело значение только для print_func, когда функция завершит работу и значения a будут напечатаны снова, мы увидим первоначально присвоенные значения.

Вот еще один более сложный пример.

```
a_var = 10
b_var = 15
e_var = 25

def a_func(a_var):
    print("в a_func a_var =", a_var)
    b_var = 100 + a_var
    d_var = 2 * a_var
    print("in a_func b_var =", b_var)
    print("in a_func d_var =", d_var)
    print("in a_func e_var =", e_var)
    return b_var + 10

c_var = a_func(b_var)

print("a_var =", a_var)
print("b_var =", b_var)
print("c_var =", c_var)
print("d_var =", d_var)
```

Вывод:

```
in a_func a_var = 15
in a_func b_var = 115
```

```
in a_func d_var = 30
in a_func e_var = 25
a_var = 10
b_var = 15
c_var = 125 d_var =

Traceback (most recent call last):
File "C:\def2.py", line 19, in <module>
    print("d_var = ", d_var)
NameError: name 'd_var' is not defined
```

В этом примере переменные `a_var`, `b_var` и `d_var` являются локальными переменными, когда они находятся внутри функции `a_func`. После выполнения оператора `return b_var + 10` все они перестают существовать. Переменная `a_var` автоматически является локальной переменной, поскольку это имя параметра. Переменные `b_var` и `d_var` являются локальными переменными, поскольку они появляются слева от знака равенства в функции в операторах `b_var = 100 + a_var` и `d_var = 2 * a_var` .

Внутри функции `a_var` не имеет присвоенного ей значения. Когда функция вызывается с `c_var = a_func(b_var)`, `a_var` присваивается значение 15, поскольку в этот момент `b_var` равен 15, что делает вызов функции `a_func(15)`. В итоге `a_var` принимает значение 15, когда находится внутри `a_func`.

Как видите, после завершения работы функции локальные переменные `a_var` и `b_var`, которые скрывали одноименные глобальные переменные, исчезли. Тогда оператор `print("a_var = ", a_var)` выводит значение 10, а не 15, поскольку локальная переменная, скрывавшая глобальную переменную, исчезла.

Еще одна вещь, на которую следует обратить внимание, — это ошибка `NameError`, возникающая в конце. Она возникает, поскольку переменная `d_var` больше не существует после завершения `a_func`. Все локальные переменные удаляются при выходе из функции. Если вы хотите получить что-то из функции, то вам придется использовать `return`.

И последнее, на что следует обратить внимание: значение `e_var` остается неизменным внутри `a_func`, поскольку она не является параметром и никогда не появляется слева от знака равенства внутри функции `a_func`. Когда к глобальной переменной обращаются внутри функции, она становится глобальной переменной извне.

Функции позволяют использовать локальные переменные, существующие только внутри функции, и могут скрывать другие переменные, находящиеся вне функции.

Примеры

temperature2.py

```
#! /usr/bin/python
#-*-кодировка: utf-8 -*-
# переводит температуру в градусы Фаренгейта или Цельсия
```

```
def print_options():
    print("Опции:")
    print(" 'p' print options")
    print(" 'c' конвертировать из Цельсия")
    print(" 'f' конвертировать из Фаренгейта")
    print(" 'q' выйти из программы")

def celsius_to_fahrenheit(c_temp):
    return 9.0 / 5.0 * c_temp + 32

def fahrenheit_to_celsius(f_temp):
    return (f_temp - 32.0) * 5.0 / 9.0

choice = "p"
while choice != "q":
    if choice== "c":
        c_temp = float(input("Температура по Цельсию: "))
        print("Фаренгейт:", celsius_to_fahrenheit(c_temp))
        choice = input("option: ")
    elif choice == "f":
        f_temp = float(input("Температура по Фаренгейту: "))
        print("Цельсий:", fahrenheit_to_celsius(f_temp))
        choice = input("option: ")
    elif choice == "p": # Альтернативный вариант choice != "q":
                        # чтобы выводить на экран,
                        # когда вводится что-нибудь неожиданное
        print_options()
        choice = input("option: ")
```

Пробный запуск:

```
Опции:
'p' параметры печати
'c' преобразовать из Цельсия
'f' преобразовать из Фаренгейта
'q' выйти из программы
вариант: c
Температура по Цельсию: 30
Фаренгейт: 86.0
вариант: f
Температура по Фаренгейту: 60
Цельсий: 15.5555555556
вариант: q
```

area2.py

```
#! /usr/bin/python
#-*-кодировка: utf-8 -*-
# вычисляет площадь заданного прямоугольника
```

```
def hello():
    print('Привет!')

def area(width, height):
    return width * heighth

def print_welcome(name):
    print('Добро пожаловать', name)

def positive_input(prompt):
    number = float(input(prompt))
    while number <= 0:
        print('Должно быть положительным числом')
        number = float(input(prompt))
    return number

name = input('Ваше имя: ')
hello()
print_welcome(name)
print()
print('Чтобы найти площадь прямоугольника,')
print('введите ширину и высоту ниже')
print() w = positive_input('Ширина: ')
h = positive_input('Высота: ')

print('Ширина =', w, ' Высота =', h, ' поэтому площадь =', area(w, h))
```

Пробный запуск:

```
Ваше имя: Джош
Привет! Добро пожаловать, Джош.

Чтобы найти площадь прямоугольника,
введите ширину и высоту ниже.

Ширина: -4
Должно быть положительным числом
Ширина: 4
Высота: 3
Ширина = 4 Высота = 3, поэтому площадь = 12
```

Упражнения

Перепишите программу area2.py из примеров выше, чтобы в ней была отдельная функция для площади квадрата, площади прямоугольника и площади круга (`3,14 * radius**2`). Эта программа должна содержать интерфейс с меню.

Решение:

```
def square(L):
     return L * L

 def rectangle(width, height):
     return width * height

 def circle(radius):
     return 3,14159 * radius ** 2

def options():
    print()
    print("Options:")
    print("s = вычислить площадь квадрата.")
    print("c = вычислить площадь круга.")
    print("r = вычислить площадь прямоугольника.")
    print("q = выйти.")
    print()

print("Эта программа вычислит площадь квадрата, круга или
прямоугольника.")
choice = "x"
options()
while choice != "q":
    choice = input("Пожалуйста, введите ваш выбор: ")
    if choice == "s":
        L = float(input("Длина стороны квадрата: "))
        print("Площадь этого квадрата равна", square(L))
        options()
    elif choice == "c":
        radius = float(input("Радиус круга: "))
        print("Площадь круга равна", circle(radius))
        options()
    elif choice == "r":
        width = float(input("Ширина прямоугольника: "))
        height = float(input("Высота прямоугольника: "))
        print("Площадь прямоугольника составляет", rectangle(width,
height))
        options()
    elif choice == "q":
        print(" ",end="")
    else:
        print("Нераспознанная опция.")
        options()
```

Глава 9

Пример продвинутых функций

Некоторые люди находят этот раздел полезным, а некоторые — запутанным. Если он кажется вам непонятным, вы можете его пропустить. Сейчас мы рассмотрим следующую программу:

```
def mult(a, b):
    if b == 0:
        return 0
    rest = mult(a, b - 1)
    value = a + rest
    return value
result = mult(3, 2)
print("3 * 2 = ", result)
```

По сути, эта программа создает функцию умножения целых положительных чисел (которая намного медленнее встроенной функции умножения), а затем демонстрирует использование этой функции. Она демонстрирует использование рекурсии, то есть формы итерации (повторения), в которой есть функция, многократно вызывающая сама себя, пока не будет выполнено условие выхода. Она использует повторные сложения для получения того же результата, что и умножение: например, 3 + 3 (сложение) дает тот же результат, что и 3 * 2 (умножение).

***Вопрос*: что первым делом делает программа?**

Ответ: Первым делом определяется функция `mult` с помощью кода:

```
def mult(a, b):
    if b == 0:
        return 0
    rest = mult(a, b - 1)
    value = a + rest
    return value
```

Это создает функцию, которая принимает два параметра и возвращает значение по завершении работы. Позже эту функцию можно будет запустить.

Что будет дальше?

Выполняется строка после функции, `result = mult(3, 2)`.

Что делает эта строка?

Эта строка присвоит возвращаемое значение `mult(3, 2)` переменной `result`.

А что возвращает `mult(3, 2)`?

Чтобы это выяснить, нам нужно пройтись по функции `mult`.

Что будет дальше?

Переменной `a` присваивается значение 3, а переменной `b` — значение 2.

А потом?

Выполняется строка `if b == 0:`. Так как `b` имеет значение 2, это ложно, поэтому строка `return 0` пропускается.

И что дальше?

Выполняется строка `rest = mult(a, b - 1)`. Эта строка устанавливает локальную переменную `rest` в значение `mult(a, b - 1)`. Значение `a` равно 3, а значение `b` равно 2, поэтому вызов функции будет `mult(3,1)`.

Так каково же значение `mult(3, 1)`?

Нам нужно будет запустить функцию `mult` с параметрами 3 и 1.

Что же будет дальше?

Локальные переменные в *новом* выполнении функции устанавливаются так, чтобы `a` имело значение 3, а `b` — значение 1. Поскольку это локальные значения, они не влияют на предыдущие значения `a` и `b`.

А потом?

Поскольку `b` имеет значение 1, оператор `if` ложен, поэтому следующая строка это `rest = mult(a, b - 1)`.

Что делает эта строка?

Эта строка присвоит значение `mult(3, 0)` остальным.

Какое это значение?

Чтобы выяснить, нам придется запустить функцию еще раз. На этот раз `a` имеет значение 3, а `b` имеет значение 0.

Что же будет дальше?

Первая строка в функции, которую нужно выполнить, — `if b == 0:`. Здесь `b` имеет значение 0, поэтому следующая строка, которую нужно выполнить, — `return 0`.

А что делает строка `return 0`?

Эта строка возвращает значение `0` из функции.

И что?

Итак, теперь мы знаем, что `mult(3, 0)` имеет значение 0. Теперь мы знаем, что строка `rest = mult(a, b - 1)`, так как мы запустили функцию `mult` с параметрами 3 и 0. Мы закончили выполнение `mult(3, 0)` и теперь вернулись к выполнению `mult(3, 1)`. Переменной `rest` присваивается значение 0.

Какая строка будет следующей?

Далее выполняется строка `value = a + rest`. В этом запуске функции `a = 3` и `rest = 0`, так что теперь `value = 3`.

Что будет дальше?

Запускается строка `return value`. Это возвращает 3 из функции. Это также выход из выполнения функции `mult(3, 1)`. После вызова `return` мы возвращаемся к выполнению `mult(3, 2)`.

Где мы были в `mult(3, 2)`?

У нас были переменные `a = 3` и `b = 2`, и мы исследуем строку `rest = mult(a, b - 1)`.

Что же теперь будет?

Переменной `rest` присваивается значение 3. Следующая строка `value = a + rest` устанавливает `value` значение `3 + 3` или 6.

И что теперь будет?

Выполняется следующая строка, которая возвращает 6 из функции. Теперь мы вернулись к выполнению строки `result = mult(3, 2)`, которая теперь может присвоить переменной `result` значение 6.

Что будет дальше?

После функции выполняется следующая строка, `print("3 * 2 = ", result)`.

И что это дает?

Выводит `3 * 2 =` и значение результата `result`, которое равно 6. Полная строка выводится как `3 * 2 = 6`.

Что происходит в целом?

В основном мы использовали два факта, чтобы вычислить кратное двух чисел. Первый заключается в том, что любое число, умноженное на 0, равно 0 (`x * 0 = 0`). Второй — число, умноженное на другое число, равно первому числу плюс первое число, умноженное на единицу меньше второго числа (`x * y = x + x * (y - 1)`). Итак, сначала `3 * 2` преобразуется в `3 + 3 * 1`. Затем `3 * 1` преобразуется в `3 + 3 * 0`. Тогда мы знаем, что любое число, умноженное на 0, равно 0, поэтому `3 * 0` равно 0. Тогда мы можем вычислить, что `3 + 3 * 0` равно `3 + 0`, что равно 3. Теперь мы знаем, что такое `3 * 1`, поэтому можем вычислить, что `3 + 3 * 1` равно `3 + 3`, что равно 6.

Вот как все это работает:

```
mult(3, 2)
3 + mult(3, 1)
3 + 3 + mult(3, 0)
3 + 3 + 0
3 + 3
6
```

Рекурсия

Программные конструкции, решающие задачу путем решения меньшей версии той же задачи, называются *рекурсивными*. В примерах, приведенных в этой главе, рекурсия реализуется путем определения функции, вызывающей саму себя. Это облегчает реализацию решений задач программирования, так как может быть достаточно рассмотреть следующий шаг задачи, а не всю проблему сразу. Это также полезно, так как позволяет выразить некоторые математические понятия с помощью прямого, легко читаемого кода.

Любая задача, которую можно решить с помощью рекурсии, может быть реализована с помощью циклов. Использование последних обычно

приводит к повышению производительности. Однако эквивалентные реализации с использованием циклов обычно сложнее сделать правильно.

Вероятно, самое интуитивное определение *рекурсии* таково:

Рекурсия

Если вы все еще не поняли, смотрите раздел «Рекурсия».

Попробуйте пройтись по примеру с факториалом, если пример с умножением не был понятен.

Примеры

factorial.py

```
#определяет функцию, которая вычисляет факториал
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)

print("2! =", factorial(2))
print("3! =", factorial(3))
print("4! =", factorial(4))
print("5! =", factorial(5))
```

Вывод:

```
2! = 2
3! = 6
4! = 24
5! = 120
```

countdown.py

```
def count_down(n):
    print(n)
    if n > 0:
        return count_down(n-1)

count_down(5)
```

Вывод:

```
5
4
3
2
1
0
```

Глава 10

Списки

Переменные с более чем одним значением

Вы уже познакомились с обычными переменными, которые хранят одно значение. Однако другие типы переменных могут хранить более одного значения. Их называют контейнерами, потому что они могут содержать более одного объекта. Самый простой тип называется списком. Пример:

```
which_one = int(input("Какой месяц (1-12)? "))
months = ['январь', 'февраль', 'март', 'апрель', 'май', 'июнь', 'июль',
'август', 'сентябрь', 'октябрь', 'ноябрь', 'декабрь']

if 1 <= which_one <= 12:
    print("Месяц - ", months[which_one - 1])
```

...и пример вывода:

```
Какой месяц (1-12)? 3
Месяц - март
```

В этом примере месяцы (`months`) — это список. Он определяется строками `months = ['January', 'February', 'March', 'April', 'May', 'June', 'July'` и `'August', 'September', 'October', 'November', 'December']` (обратите внимание, что `\` можно также использовать для разделения длинной строки, но в данном случае это не нужно, поскольку Python достаточно умен, чтобы понять, что все внутри скобок принадлежит друг другу). Знаки `[` и `]` начинают и заканчивают список, разделенный запятыми (`,`) на элементы. Список используется в `months[which_one - 1]`. Он состоит из элементов, которые нумеруются, начиная с 0. Другими словами, если вам нужен январь, вы будете использовать `months[0]`. Задайте списку номер, и он вернет значение, хранящееся в этом месте.

Утверждение `if 1 <= which_one <= 12:` будет истинным только в том случае, если `which_one` находится в диапазоне от одного до двенадцати включительно (другими словами, это то, что вы ожидаете увидеть, если видели это в алгебре).

Списки можно представить как ряд ячеек. Каждая ячейка имеет свое значение. Например, ячейки, созданные с помощью `demolist = ['жизнь', 42, 'Вселенная', 6, 'и', 9]`, будут выглядеть следующим образом:

номер ячейки	0	1	2	3	4	5
demolist	"жизнь"	42	"Вселенная"	6	"и"	9

Каждая ячейка обозначается своим номером, так что утверждение `demolist[0]` получит `"жизнь"`, `demolist[1]` — `42` и так далее, вплоть до ячейки `demolist[5]`, из которой будет получено `9`.

Дополнительные возможности списков

Следующий пример призван продемонстрировать множество других возможностей списков (я не ожидаю, что вы введете их, но, возможно, вам стоит «поиграть» со списками в интерактивном режиме, пока вы не освоитесь с ними). Итак:

```
demolist = ["жизнь", 42, "Вселенная", 6, "и", 9]
print("demolist = ",demolist)
demolist.append("все")
print("после добавления 'все' demolist стал выглядеть так:")
print(demolist)
print("len(demolist) =", len(demolist))
print("demolist.index(42) =", demolist.index(42))
print("demolist[1] =", demolist[1])

# Далее мы пройдемся циклом по списку
for c in range(len(demolist)):
    print("demolist[", c, "] =", demolist[c])

del demolist[2]
print("После удаления 'Вселенная' demolist стал:")
print(demolist)
if "жизнь" in demolist:
    print("'жизнь' найдена в demolist")
else:
    print("'жизнь' не найдена в demolist")

if "амеба" in demolist:
    print("'амеба' найдена в demolist")
if "амеба" not in demolist:
    print("'амеба' не найдена в demolist")

another_list = [42,7,0,123]
another_list.sort()
    print("Отсортированный список another_list имеет вид", another_list)
```

На выходе получаем:

```
demolist = ['жизнь', 42, 'Вселенная', 6, 'и', 9]
после добавления 'все' demolist стал выглядеть так:
["жизнь", 42, "Вселенная", 6, "и", 9, "все"]
len(demolist) = 7
demolist.index(42) = 1
demolist[1] = 42
demolist[ 0 ] = жизнь
demolist[ 1 ] = 42
demolist[ 2 ] = Вселенная
```

```
demolist[ 3 ] = 6
demolist[ 4 ] = и
demolist[ 5 ] = 9
demolist[ 6 ] = все
После удаления 'Вселенная' demolist стал:
['жизнь', 42, 6, 'и', 9, 'все']
'жизнь' найдена в demolist
'амеба' не найдена в demolist
Отсортированный список another_list имеет вид [0, 7, 42, 123]
```

В этом примере используется целая куча новых функций. Обратите внимание, что вы можете просто вывести (`print`) весь список. Далее используется функция `append` для добавления нового элемента в конец списка. `len` возвращает количество элементов в списке. Допустимые индексы (числа, которые можно использовать внутри `[]`) списка лежат в диапазоне от 0 до `len - 1`. Функция `index` указывает, где в списке находится первый элемент. Обратите внимание, что `demolist.index(42)` возвращает 1, а при выполнении `demolist[1]` возвращает 42. Чтобы получить справку по всем функциям, которые предоставляет список, введите `help(list)` в интерактивном интерпретаторе Python.

Строка `# Далее мы пройдемся циклом по списку` — это просто напоминание программисту (*комментарий*). Python игнорирует все, что написано после символа `#` в текущей строке. Следующие строки:

```
for c in range(len(demolist)):
    print("demolist[", c, "] =", demolist[c])
```

...создают переменную `c`, которая начинается с 0 и увеличивается до тех пор, пока не достигнет последнего индекса списка. Тем временем оператор `print` выводит каждый элемент списка.

Гораздо лучше поступить следующим образом:

```
for c, x in enumerate(demolist):
    print("demolist[", c, "] =", x)
```

Команду `del` можно использовать для удаления заданного элемента в списке. В следующих нескольких строках используется оператор `in` для проверки наличия или отсутствия элемента в списке. Функция `sort` сортирует список. Это полезно, если вам нужен список в порядке от наименьшего числа к наибольшему или в алфавитном порядке. Обратите внимание, что при этом список перестраивается. В общем, для списка применяются следующие операции:

пример	объяснение
`demolist[2]`	получает доступ к элементу с индексом 2
`demolist[2] = 3`	устанавливает элемент с индексом 2 в значение 3
`del demolist[2]`	удаляет элемент с индексом 2
`len(demolist)`	возвращает длину demolist

пример	объяснение
"value" in demolist	будет True, если "value" является элементом demolist
"value" not in demolist	будет True, если "value" не является элементом demolist
another_list.sort()	сортирует another_list. Обратите внимание, что список должен содержать все числа или все строки для сортировки.
demolist.index("value")	возвращает индекс первой позиции, где встречается "value"
demolist.append("value")	добавляет элемент "value" в конец списка
demolist.remove("value")	удаляет первое вхождение значения из demolist (то же самое, что del demolist[demolist.index("value")])

В следующем примере эти функции используются более «полезным» образом:

```
menu_item = 0
namelist = []
while menu_item != 9:
    print("--------------------")
    print("1. Вывести список")
    print("2. Добавить имя в список")
    print("3. Удалить имя из списка")
    print("4. Изменить элемент в списке")
    print("9. Выйти")
    menu_item = int(input("Выберите пункт меню: "))
    if menu_item == 1:
        current = 0
        if len(namelist) > 0:
            while current < len(namelist):
                print(current, ".", namelist[current])
                current = current + 1
        else:
            print("Список пуст")
    elif menu_item == 2:
        name = input("Введите имя для добавления: ")
        namelist.append(name)
    elif menu_item == 3:
        del_name = input("Какое имя вы хотите удалить: ")
        if del_name in namelist:
            # namelist.remove(del_name) будет работать так же хорошо.
            item_number = namelist.index(del_name)
            del namelist[item_number]
            # Код выше удаляет только первое вхождение
            # имени. Приведенный ниже код удаляет все.
            # while del_name in namelist:
```

```
            #    item_number = namelist.index(del_name)
            #    del namelist[item_number]
        else:
            print(del_name, "не найдено")
    elif menu_item == 4:
        old_name = input("Какое имя вы хотели бы изменить: ")
        if old_name in namelist:
            item_number = namelist.index(old_name)
            new_name = input("Какое новое имя: ")
            namelist[item_number] = new_name
        else:
            print(old_name, "не найдено")
print("До свидания")
```

А вот часть вывода:

```
--------------------
1. Вывести список
2. Добавить имя в список
3. Удалить имя из списка
4. Изменить элемент в списке
9. Выйти

Выберите пункт меню: 2
Введите имя для добавления: Джек

Выберите пункт меню: 2
Введите имя для добавления: Джилл

Выберите пункт меню: 1
0 . Джек
1 . Джилл

Выберите пункт меню: 3
Какое имя вы хотите удалить: Джек

Выберите пункт меню: 4
Какое имя вы хотели бы изменить: Джилл
Какое новое имя: Джилл Питерс

Выберите пункт меню: 1
0 . Джилл Питерс

Выберите пункт меню: 9
До свидания
```

Это была длинная программа. Давайте посмотрим на исходный код. Строка `namelist = []` делает переменную `namelist` списком без элементов. Следующая важная строка — `while menu_item != 9:`. Эта строка запускает цикл, который позволяет создать систему меню для этой программы.

Следующие несколько строк выводят меню и решают, какую часть программы запускать.

Раздел

```
current = 0
if len(namelist) > 0:
    while current < len(namelist):
        print(current, ".", namelist[current])
        current = current + 1
else:
    print("Список пуст")
```

проходит по списку и выводит каждое имя. `len(namelist)` показывает, сколько элементов в списке. Если `len` возвращает `0`, значит, список пуст.

Затем через несколько строк появляется оператор `namelist.append(name)`. Он использует функцию `append` для добавления элемента в конец списка. Перейдите еще на две строки вниз и обратите внимание на этот участок кода:

```
item_number = namelist.index(del_name)
del namelist[item_number]
```

Здесь функция `index` используется для нахождения значения индекса, который в дальнейшем будет использоваться для удаления элемента. `del namelist[item_number]` используется для удаления элемента списка.

Следующий раздел

```
old_name = input("Какое имя вы хотели бы изменить: ")
if old_name in namelist:
    item_number = namelist.index(old_name)
    new_name = input("Какое новое имя: ")
    namelist[item_number] = new_name
else:
    print(old_name, "не найдено")
```

использует `index`, чтобы найти номер элемента (`item_number`), а затем помещает новое имя (`new_name`) туда, где было старое имя (`old_name`).

Поздравляем, теперь вы знаете достаточно, чтобы выполнять любые вычисления, которые может делать компьютер (это технически известно как полнота по Тьюрингу). Конечно, существует еще множество функций, которые используются для облегчения жизни программиста.

Примеры

test.py

```
## Эта программа проводит тест на знание

# Сначала получите вопросы теста.
# Позже это будет изменено, чтобы использовать файловый ввод/вывод.
def get_questions():
    # обратите внимание, что данные хранятся в виде списка списков
    return [["Какого цвета дневное небо в ясный день?", "голубого"],
```

```
            ["Что является ответом на вопрос о жизни, Вселенной и всем
остальном?", "42"],
            ["Какое слово из трех букв обозначает мышеловку?", "кот"].

# Это проверит один вопрос
# для этого нужен один вопрос
# возвращает True, если пользователь ввел правильный ответ, иначе False

def check_question(question_and_answer):
    # извлеките вопрос и ответ из списка
    # Функция принимает список с двумя элементами - вопросом и ответом.
    question = question_and_answer[0]
    answer = question_and_answer[1]
    # задать вопрос пользователю
    given_answer = input(question)
    # сравнить ответ пользователя с ответом проверяющего
    if answer == given_answer:
        print("Правильно")
        return True
    else:
        print("Неверно, правильный ответ:", answer)
        return False

# Это позволит просмотреть все вопросы.
def run_test(questions):
    if len(questions) == 0:
        print("Не было задано ни одного вопроса.")
        # return завершает работу функции
        return
    index = 0
    right = 0
    while index < len(questions):
        # Проверьте вопрос
        # Обратите внимание, что это извлекает список вопросов
и ответов из списка списков.
        if check_question(questions[index]):
            right = right + 1
        # перейти к следующему вопросу
        index = index + 1
    # обратите внимание на порядок вычислений: сначала умножение, затем
деление
    print("Вы получили", right * 100 / len(questions),\
        "% правильно из", len(questions))

# теперь давайте получим вопросы из функции get_questions,
# и отправим полученный список списков в качестве аргумента функции
run_test.

run_test(get_questions())
```

Значения `True` и `False` указывают на 1 и 0 соответственно. Они часто используются при проверке правильности, в условиях цикла и т. д. Подробнее об этом вы узнаете чуть позже (глава «Булевы выражения»). Обратите внимание, что `get_questions()` — это, по сути, список, потому что, хотя технически это функция, возвращающая список списков, — это единственное, что она делает.

Образец вывода:

```
Какого цвета дневное небо в ясный день? зеленого
Неверно, правильный ответ: голубого
Что является ответом на вопрос о жизни, Вселенной и всем остальном? 42
Правильно
Какое слово из трех букв обозначает мышеловку? кот
Правильно
Вы получили 66 % правильно из 3
```

Упражнения

Дополните программу *test.py* так, чтобы в ней появилось меню с возможностью пройти тест, просмотреть список вопросов и ответов, а также выйти из программы. Также добавьте новый вопрос: «Какой звук издает по-настоящему продвинутая машина?» с ответом «пинг».

Решение:

```
## Эта программа выполняет тест на знание
questions = [["Какого цвета дневное небо в ясный день?", "голубого"],
             ["Что является ответом на вопрос о жизни, Вселенной и всем
остальном?", "42"],
             ["Какое слово из трех букв обозначает мышеловку?", "кот"],
             ["Какой звук издает по-настоящему продвинутая машина?",
"пинг"]].

# Это проверит один вопрос
# для этого нужен один вопрос
# возвращает True, если пользователь ввел правильный ответ, иначе False


def check_question(question_and_answer):
    # извлеките вопрос и ответ из списка
    question = question_and_answer[0]
    answer = question_and_answer[1]
     # задать вопрос пользователю
     given_answer = input(question)
     # сравните ответ пользователя с ответом проверяющего
    if answer == given_answer:
        print("Правильно")
        return True
    else:
    print("Неверно, правильный ответ:", answer)
    return False
```

```
# Это позволит просмотреть все вопросы.

def run_test(questions):

    if len(questions) == 0:
        print("Ни один вопрос не был задан.")
        # return завершает работу функции
        return
    index = 0
    right = 0
    while index < len(questions):
        # Проверьте вопрос
        if check_question(questions[index]):
            right = right + 1
        # перейти к следующему вопросу
        index = index + 1
    # обратите внимание на порядок вычислений: сначала умножение, затем
деление
    print("Вы получили", right * 100 / len(questions), "% правильно
из", len(questions))
#показ списка вопросов и ответов
def showquestions():
    q = 0
    while q < len(questions):
        a = 0
        print("Q:" , questions[q][a])
        a = 1
        print("A:" , questions[q][a])
        q = q + 1

# теперь давайте определим функцию меню
def menu():
    print("-----------------")
    print("Меню:") print("1 - Пройти тест")
    print("2 - Просмотр списка вопросов и ответов")
    print("3 - Просмотр меню")
    print("5 - Выход")
    print("-----------------")

choice = "3"
while choice != "5":
    if choice == "1":
        run_test(questions)
    elif choice == "2":
        showquestions()
    elif choice == "3":
        menu()
print()
choice = input("Выберите вариант из меню выше: ")
```

Глава 11

Циклы for

А вот и новое упражнение по набору текста для этой главы:

```
onetoten = range(1, 11)
for count in onetoten:
    print(count)
```

...и вечно актуальный вывод (он будет отображен в столбик):

```
1 2 3 4 5 6 7 8 9 10
```

Вывод выглядит ужасно знакомым, но программный код выглядит иначе. В первой строке используется функция `range`. Эта функция использует два аргумента, например `range(start, finish)`. `start` — это первое полученное число. `finish` — это число на единицу больше последнего. Обратите внимание, что эту программу можно было бы сделать и короче:

```
for count in range(1, 11):
    print(count)
```

Функция `range` возвращает итерируемое число. Его можно преобразовать в список с помощью функции `list`, тогда оно будет доминирующим числом. Вот несколько примеров, показывающих, что происходит при использовании команды `range`:

```
>>> range(1, 10)
range(1, 10)
>>> list(range(1, 10))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(-32, -20))
[-32, -31, -30, -29, -28, -27, -26, -25, -24, -23, -22, -21]
>>> list(range(5,21))
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> list(range(21, 5))
[]
```

В следующей строке `for count in onetoten:` используется управляющая структура `for`. Управляющая структура `for` выглядит как `for variable in list:`. Список `list` просматривается, начиная с первого элемента списка и до последнего. По мере того как `for` проходит через каждый элемент списка, он помещает каждый из них в переменную `variable`. Это позволяет использовать переменную в каждом последующем цикле `for`. Вот еще один пример (его можно не набирать) для демонстрации:

```
demolist = ['жизнь', 42, 'Вселенная', 6, 'и', 7, 'все'].
for item in demolist:
    print("Текущий элемент:", item)
```

На выходе получаем:

```
Текущий элемент: жизнь
Текущий элемент: 42
Текущий пункт: Вселенная
Текущий пункт: 6
Текущий элемент: и
Текущий элемент: 7
Текущий элемент: все
```

Обратите внимание, как цикл `for` проходит и устанавливает элемент на каждый элемент списка. Итак, для чего же нужен `for`? Первое применение — перебрать все элементы списка и что-то сделать с каждым из них. Вот быстрый способ сложить все элементы:

```
list = [2, 4, 6, 8]
sum = 0
for num in list:
    sum = sum + num
print("Сумма:", sum)
```

...с простым выводом:

```
Сумма: 20
```

Или вы можете написать программу, которая будет выяснять, есть ли в списке дубликаты, как это делает данная программа:

```
list = [4, 5, 7, 8, 9, 1, 0, 7, 10]
list.sort()
prev = None
 for item in list:
    if prev == item:
        print("Дубликат", prev, "найден")
    prev = item
```

...и для пущей убедительности:

```
Дубликат 7 найден
```

Итак, как это работает? Вот специальная отладочная версия, которая поможет вам в этом разобраться (набирать ее не нужно):

```
l = [4, 5, 7, 8, 9, 1, 0, 7, 10]
print("l = [4, 5, 7, 8, 9, 1, 0, 7, 10]", "\t\tl:", l)
l.sort()
print("l.sort()", "\t\tl:", l)
prev = l[0]
print("prev = l[0]", "\t\tprev:", prev)
del l[0]
print("del l[0]", "\t\tl:", l)
for item in l:
    if prev == item:
        print("Дубликат", prev, "найден")
```

```
    print("if prev == item:", "\t\tprev:", prev, "\titem:", item)
    prev = item
    print("prev = item", "\t\tprev:", prev, "\titem:", item)
```

...с выводом:

```
l = [4, 5, 7, 8, 9, 1, 0, 7, 10]    l: [4, 5, 7, 8, 9, 1, 0, 7, 10]
l.sort()                    l: [0, 1, 4, 5, 7, 7, 8, 9, 10]
prev = l[0]                 prev: 0
del l[0]                    l: [1, 4, 5, 7, 7, 8, 9, 10]
if prev == item:            prev: 0        item: 1
prev = item                 prev: 1        item: 1
if prev == item:            prev: 1        item: 4
prev = item                 prev: 4        item: 4
if prev == item:            prev: 4        item: 5
prev = item                 prev: 5        item: 5
if prev == item:            prev: 5        item: 7
prev = item                 prev: 7        item: 7
Дубликат 7 найден
if prev == item:            prev: 7        item: 7
prev = item                 prev: 7        item: 7
if prev == item:            prev: 7        item: 8
prev = item                 prev: 8        item: 8
if prev == item:            prev: 8        item: 9
prev = item                 prev: 9        item: 9
if prev == item:            prev: 9        item: 10
prev = item                 prev: 10       item: 10
```

Я поместил в код так много операторов печати, чтобы вы могли видеть, что происходит в каждой строке. (Кстати, если вы не можете понять, почему программа не работает, попробуйте вставить много операторов печати в те места, где вы хотите узнать, что происходит). Сначала программа начинает со скучного старого списка, затем сортирует его, чтобы все дубликаты были помещены рядом друг с другом. Затем программа инициализирует переменную `prev` (previous, предыдущую). После этого первый элемент списка удаляется, чтобы первый элемент не был ошибочно принят за дубликат. Далее запускается цикл `for`. Каждый элемент списка проверяется на то, совпадает ли он с предыдущим. Если да, то дубликат найден. Затем значение `prev` изменяется таким образом, чтобы при следующем запуске цикла `for` `prev` был предыдущим элементом по отношению к текущему. Конечно, 7 оказывается дубликатом. (Обратите внимание, `\t` используется для табуляции).

Другой способ использования циклов `for` — это выполнение определенных действий определенное количество раз. Вот код, который выведет первые 9 чисел ряда Фибоначчи:

```
a = 1
b = 1
```

```
for c in range(1, 10):
    print(a, end=" ")
    n = a + b
    a = b
    b = n
```

...с удивительным результатом:

```
1 1 2 3 5 8 13 21 34
```

Все, что можно сделать с помощью циклов `for`, можно сделать и с помощью циклов `while`, но циклы `for` дают простой способ перебрать все элементы в списке или сделать что-то определенное количество раз.

Глава 12

Булевы выражения

Вот небольшой пример булевых выражений (набирать не обязательно):

```
a = 6
b = 7
c = 42
print(1, a == 6)
print(2, a == 7)
print(3, a == 6 and b == 7)
print(4, a == 7 and b == 7)
print(5, not a == 7 and b == 7)
print(6, a == 7 or b == 7)
print(7, a == 7 or b == 6)
print(8, not (a == 7 and b == 6))
print(9, not a == 7 and b == 6)
```

На выходе получаем:

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

Что происходит? Программа состоит из кучи забавно выглядящих операторов печати. Каждый оператор печати выводит число и выражение. Число нужно для того, чтобы определить, с каким оператором я имею дело. Обратите внимание, что каждое выражение в итоге оказывается либо `False`, либо `True`. В Python `False` также можно записать как 0, а `True` — как 1.

Строки:

```
print(1, a == 6)
print(2, a == 7)
```

...выводят `True` и `False` соответственно, как и ожидалось, поскольку первое из них истинно, а второе — ложно. Третий вывод, `print(3, a == 6 and b == 7)`, немного отличается. Оператор `and` означает, что если и предыдущее, и последующее утверждения истинны, то все выражение истинно, иначе все выражение ложно. Следующая строка, `print(4, a == 7 and b == 7)`, показывает, что если часть выражения `and` ложна, то ложно и все выражение. Поведение `and` можно описать следующим образом:

выражение	результат
истинно and истинно	истинно
истинно and ложно	ложно
ложно and истинно	ложно
ложно and ложно	ложно

Обратите внимание, что если первое выражение ложно, Python не проверяет второе выражение, поскольку знает, что все выражение ложно. Попробуйте выполнить `False and print("Hi")` и сравните это с выполнением `True and print("Hi")`. Технический термин для этого — оценка короткого замыкания. В следующей строке, `print(5, not a == 7 and b == 7)`, используется оператор `not`. Он просто дает противоположное выражение. (Выражение можно переписать как `print(5, a != 7 and b == 7)`). Вот таблица:

выражение	результат
not истинно	ложно
not ложно	истинно

В двух следующих строках, `print(6, a == 7 or b == 7)` и `print(7, a == 7 or b == 6)`, используется оператор `or`. Оператор `or` возвращает «истинно», если первое выражение истинно, или если истинно второе выражение, или оба выражения истинны. Если ни одно из них не истинно, он возвращает «ложно». Вот таблица:

выражение	результат
истинно or истинно	истинно
истинно or ложно	истинно
ложно or истинно	истинно
ложно or ложно	ложно

Обратите внимание, что если первое выражение истинно, Python не проверяет второе выражение, поскольку знает, что все выражение истинно. Это работает, поскольку `or` истинно, если хотя бы одна половина

выражения истинна. Первая часть истинна, поэтому вторая часть может быть как ложной, так и истинной, но все равно в итоге выражение истинно.

Следующие две строки, `print(8, not (a == 7 and b == 6))` и `print(9, not a == 7 и b == 6)`, показывают, что круглые скобки можно использовать чтобы группировать выражения и заставить одну часть оцениваться первой. Обратите внимание, что круглые скобки изменили выражение с «ложно» на «истинно». Это произошло потому, что круглые скобки заставляют применять `not` ко всему выражению, а не только к части `a == 7`.

Вот пример использования подобного логического выражения:

```
list = ["Жизнь", "Вселенная", "Все", "Джек", "Джилл", "Жизнь", "Джилл"].

# создайте копию списка. О том, что означает [:], читайте в главе
"Подробнее о списках".
copy = list[:]
# сортировка копии
copy.sort()
prev = copy[0]
del copy[0]

count = 0

# пройдитесь по списку в поисках совпадения
while count < len(copy) and copy[count] != prev:
    prev = copy[count]
    count = count + 1

# Если совпадение не найдено, то count не может быть < len
# так как цикл while продолжается, пока count < len
# и совпадение не найдено.

if count < len(copy):
    print("Первое совпадение:", prev)
```

И вот результат:

```
Первое совпадение: Джилл
```

Эта программа работает, продолжая проверять совпадение, пока `count < len(copy) and copy[count] is not equal to prev`. Когда либо `count` больше последнего индекса `copy`, либо совпадение найдено, то `and` больше не является истиной, и цикл завершается. Функция `if` просто проверяет, что цикл `while` завершился, потому что было найдено совпадение.

В этом примере используется еще один «трюк» с `and`. Если вы посмотрите на таблицу для `and`, то заметите строку «ложно `and` ложно». Если `count >= len(copy)` (другими словами, `count < len(copy)` — ложно), то `copy[count]` никогда не просматривается. Это происходит потому, что Python знает, что если первая часть ложная, то обе они не могут быть истинными. Это известно как короткое замыкание и полезно, если второе выражение вызовет ошибку, если что-то не так. Я использовал первое выражение

(`count < len(copy)`), чтобы проверить, является ли `count` допустимым индексом для `copy`. (Если не верите, удалите совпадения «Джилл» и «Жизнь», проверьте, что все еще работает, а затем измените порядок: `count < len(copy) and copy[count] != prev` на `copy[count] != prev and count < len(copy)`).

Булевы выражения можно использовать, когда нужно проверить два или более разных условий одновременно.

Заметка о булевых операторах

Частой ошибкой новичков в программировании является непонимание того, как работают булевы операторы, что связано с тем, как интерпретатор Python читает эти выражения. Например, после первоначального изучения операторов «и» и «или» можно предположить, что выражение `x == ('a' or 'b')` будет проверять, эквивалентна ли переменная `x` одной из строк `'a'` или `'b'`. Это не так. Чтобы понять, о чем я говорю, запустите интерактивную сессию с интерпретатором и введите следующие выражения:

```
>>> 'a' == ('a' or 'b')
>>> 'b' == ('a' or 'b')
>>> 'a' == ('a' and 'b')
>>> 'b' == ('a' and 'b')
```

И это будет неинтуитивный результат:

```
>>> 'a' == ('a' or 'b')
True
>>> 'b' == ('a' or 'b')
False
>>> 'a' == ('a' and 'b')
False
>>> 'b' == ('a' and 'b')
True
```

На этом этапе операторы `and` и `or` кажутся нерабочими. Нелогично, что для первых двух выражений `'a'` эквивалентно `'a'` или `'b'`, а `'b'` — нет. Более того, не имеет смысла, что `'b'` эквивалентно `'a'` и `'b'`. После изучения того, что интерпретатор делает с булевыми операторами, эти результаты на самом деле делают именно то, что вы от них требуете, просто это не то же самое, что вы думаете.

Когда интерпретатор Python просматривает выражение `or`, он берет первое утверждение и проверяет, истинно ли оно. Если первое утверждение истинно, то Python возвращает значение объекта, не проверяя второе утверждение. Это происходит потому, что для выражения `or` истинным является только одно из значений; программе не нужно беспокоиться о втором утверждении. С другой стороны, если первое значение оценивается как ложное, Python проверяет вторую половину и возвращает это значение. Эта вторая половина определяет истинностное значение всего выражения, поскольку первая половина была ложной. Такая «лень» со стороны интерпретатора называется «замыканием» и является обычным способом оценки булевых выражений во многих языках программирования.

Аналогично для выражения and Python использует технику короткого замыкания, чтобы ускорить оценку истинностного значения. Если первое утверждение ложно, то и все должно быть ложно, поэтому возвращается это значение. В противном случае, если первое значение истинно, проверяется второе и возвращается это значение. Следует отметить, что булево выражение возвращает значение True или False, но Python считает, что истинностное значение присваивается множеству различных объектов. Чтобы проверить истинностное значение любого объекта x, вы можете использовать функцию bool(x), чтобы узнать его истинностное значение. Ниже приведена таблица с примерами истинностных значений различных объектов:

Правда	Ложь
Правда	Ложь
1	0
Числа, отличные от нуля	Строка 'None'
Непустые строки	Пустые строки
Непустые списки	Пустые списки
Непустые словари	Пустые словари

Теперь можно понять, к чему приводили странные результаты, которые мы получали при тестировании этих булевых выражений. Давайте посмотрим, что «видит» интерпретатор при выполнении этого кода:

Первый случай:

```
>>> 'a' == ('a' or 'b') # Сначала смотрим на круглые скобки, поэтому
оцениваем выражение "('a' или 'b')"
              # 'a' - непустая строка, поэтому первое значение - True
              # Верните первое значение: 'a'
>>> 'a' == 'a'# строка 'a' эквивалентна строке 'a', поэтому выражение
True
True
```

Второй случай:

```
>>> 'b' == ('a' или 'b') # Сначала смотрим на круглые скобки, поэтому
оцениваем выражение "('a' или 'b')"
              # 'a' - непустая строка, поэтому первое значение - True
              # Верните первое значение: 'a'
>>> 'b' == 'a'# строка 'b' не эквивалентна строке 'a', поэтому
выражение False
False
```

Третий случай:

```
>>> 'a' == ('a' и 'b') # Сначала посмотрите на скобки, поэтому
оцените выражение "('a' и 'b')".
```

```
                # 'a' - непустая строка, поэтому первое значение -
                True, исследуем второе значение
                # 'b' - непустая строка, поэтому второе значение -
                True
                # Возвращаем второе значение как результат всего
                выражения: 'b'
>>> 'a' == 'b'# строка 'a' не эквивалентна строке 'b', поэтому
выражение False
False
```

Четвертый случай:

```
>>> 'b' == ('a' и 'b') # Сначала посмотрите на скобки, поэтому
оцените выражение "('a' и 'b')".
                # 'a' - непустая строка, поэтому первое значение -
                True, исследуем второе значение
                # 'b' - непустая строка, поэтому второе значение -
                True
                # Возвращаем второе значение как результат всего
                выражения: 'b'
>>> 'b' == 'b'# строка 'b' эквивалентна строке 'b', поэтому
выражение True
True
```

Так что Python действительно выполнял свою работу, когда выдавал эти явно ложные результаты. Как уже говорилось ранее, важно понять, какое значение вернет ваше булево выражение, когда оно будет оценено, потому что это не всегда очевидно.

Возвращаясь к исходным выражениям, вот как их можно записать, чтобы они вели себя так, как вам нужно:

```
>>> 'a' == 'a' or 'a' == 'b'
True
>>> 'b' == 'a' or 'b' == 'b'
True
>>> 'a' == 'a' and 'a' == 'b'
False
>>> 'b' == 'a' and 'b' == 'b'
False
```

Когда эти сравнения оцениваются, они возвращают истинные значения в терминах «истинно» или «ложно», а не строки, поэтому мы получаем правильные результаты.

Примеры

password1.py

```
## Эта программа запрашивает у пользователя имя и пароль.
```

```
# Затем она проверяет их, чтобы убедиться, что пользователю разрешен
вход.

name = input("Как вас зовут? ")
password = input("Какой пароль? ")
if name == "Джош" and password == "Friday":
    print("Добро пожаловать, Джош")
elif name == "Фред" and password == "Rock":
    print("Добро пожаловать, Фред")
else:
    print("Я вас не знаю.")
```

Пробные прогоны:

```
Как вас зовут? Джош
Какой пароль? Friday
Добро пожаловать, Джош.

Как вас зовут? Билл
Какой пароль? Деньги
Я вас не знаю.
```

Упражнения

Напишите программу, в которой пользователь должен угадать ваше имя, но у него есть только три попытки сделать это.

Решение:

```
print("Попробуйте угадать мое имя!")
count = 1
name = "гильерме"
guess = input("Как меня зовут?")
while count < 3 and guess.lower() != name:
# .lower позволяет таким вещам, как Гильерме, все еще соответствовать
    print("Вы ошиблись!")
    guess = input("Как меня зовут? ")
    count = count + 1

if guess.lower() != name:
    print("Вы ошиблись!")
    # это сообщение не было выведено в третьей попытке, поэтому мы
выводим его сейчас
    print("У вас закончились попытки").
else:
    print("Да! Меня зовут", name + "!")
```

Глава 13

Словари

Эта глава посвящена словарям. В словарях есть ключи и значения. Ключи используются для поиска значений. Вот пример использования словаря:

```
def print_menu():
    print('1. Вывод телефонных номеров')
    print('2. Добавить номер телефона')
    print('3. Удалить номер телефона')
    print('4. Поиск телефонного номера')
    print('5. Выход')
    print()

numbers = {}
menu_choice = 0
print_menu()
while menu_choice != 5:
    menu_choice = int(input("Введите число (1-5): "))
    if menu_choice == 1:
       print("Номера телефонов:")
       for x in numbers.keys():
           print("Имя: ", x, "\tНомер:", numbers[x])
       print()
    elif menu_choice == 2:
       print("Добавить имя и номер")
       name = input("Имя: ")
       phone = input("Номер: ")
       numbers[name] = phone
    elif menu_choice == 3:
       print("Удалить имя и номер")
       name = input("Имя: ")
       if name in numbers:
           del numbers[name]
       else:
           print(name, "не найдено")
    elif menu_choice == 4:
        print("Поиск номера")
        name = input("Имя: ")
        if name in numbers:
            print("Число равно", numbers[name])
        else:
            print(name, "не найдено")
    elif menu_choice != 5:
        print_menu()
```

И вот мой вывод:

```
1. Вывод телефонных номеров
```

```
2. Добавить номер телефона
3. Удалить номер телефона
4. Поиск телефонного номера
5. Выход

Введите число (1-5): 2
Добавить имя и номер
Имя: Джо
Номер: 545-4464
Введите число (1-5): 2
Добавить имя и номер
Имя: Джил
Номер: 979-4654
Введите число (1-5): 2
Добавить имя и номер
Имя: Фред
Номер: 132-9874
Введите число (1-5): 1
Номера телефонов:
Имя: Джил      Номер: 979-4654
Имя: Джо       Номер: 545-4464
Имя: Фред      Номер: 132-9874

Введите число (1-5): 4
Поиск номера
Имя: Джо
Номер 545-4464
Введите номер (1-5): 3
Удалить имя и номер
Имя: Фред
Введите число (1-5): 1
Номера телефонов:
Имя: Джил      Номер: 979-4654
Имя: Джо       Номер: 545-4464
Введите число (1-5): 5
```

Эта программа похожа на список имен, описанный ранее в главе о списках. Вот как работает программа. Сначала определяется функция `print_menu`. Она просто печатает меню, которое впоследствии дважды используется в программе. Далее идет забавно выглядящая строка `numbers = {}`. Все, что делает эта строка, это сообщает Python, что `numbers` — это словарь. Следующие несколько строк просто заставляют меню работать. Строки

```
for x in numbers.keys():
    print("Имя:", x, "\tНомер:", numbers[x])
```

просматривают словарь и выводят всю информацию. Функция `numbers.keys()` возвращает список, который затем используется в ци-

кле for. Список, возвращаемый функцией keys(), не имеет определенного порядка, поэтому, если вы хотите получить его в алфавитном порядке, его необходимо отсортировать. Как и в случае со списками, оператор numbers[x] используется для доступа к определенному элементу словаря. Конечно, в данном случае x — это строка. Следующая строка numbers[name] = phone добавляет в словарь имя и номер телефона. Если имя name уже было в словаре, то phone заменит то, что было до этого. Следующие строки

```
if name in numbers:
    del numbers[name]
```

дают команду проверить, есть ли имя в словаре, и удалить его, если есть. Оператор name in numbers возвращает true, если имя есть в numbers, а в противном случае возвращает false. Строка del numbers[name] удаляет ключ name и значение, связанное с этим ключом. Строки

```
if name in numbers:
    print("Номер: ", numbers[name])
```

проверяют, есть ли в словаре определенный ключ, и если есть, выводят связанный с ним номер. Наконец, если выбор меню недействителен, он перепечатывает меню для вашего удовольствия.

Вкратце: в словарях есть ключи и значения. Ключи могут быть строками или числами и указывают на значения. Значениями могут быть переменные любого типа (включая списки и даже словари (эти словари или списки, конечно, могут сами содержать словари или списки (страшно, правда? :-))). Вот пример использования списка в словаре:

```
max_points = [25, 25, 50, 25, 100]
assignments = ['hw ch 1', 'hw ch 2','quiz   , 'hw ch 3', 'test']
students = {'#Max': max_points}

def print_menu():
    print("1. Добавить студента")
    print("2. Удалить студента")
    print("3. Вывести оценки")
    print("4. Записать оценку")
    print("5. Меню вывода")
    print("6. Выход")

def print_all_grades():
    print('\t', end=' ')
    for i in range(len(assignments)):
        print(assignments[i], '\t', end=' ')
    print()
    keys = list(students.keys())
    keys.sort()
    for x in keys:
        print(x, '\t', end=' ')
```

```
        grades = students[x]
        print_grades(grades)

def print_grades(grades):
    for i in range(len(grades)):
        print(grades[i], '\t', end=' ')
    print()

print_menu()
menu_choice = 0
while menu_choice != 6:
    print()
    menu_choice = int(input("Выбор меню (1-6): "))
    if menu_choice == 1:
        name = input("Студент для добавления: ")
        students[name] = [0] * len(max_points)
    elif menu_choice == 2:
        name = input("Студент, которого нужно удалить: ")
        if name in students:
            del students[name]
        else:
            print("Студент:", имя, "не найден")
    elif menu_choice == 3:
        print_all_grades()
    elif menu_choice == 4:
        print("Записать оценку")
        name = input("Студент: ")
        if name in students:
            grades = students[name]
            print("Введите номер оценки для записи")
            print("Введите 0 (ноль) для выхода")
            for i in range(len(assignments)):
                print(i + 1, assignments[i], '\t', end=' ')
            print()
            print_grades(grades)
            which = 1234
            while which != -1:
                which = int(input("Изменить оценку: "))
                which -= 1   #same as which = which - 1
                if 0 <= which < len(grades):
                    grade = int(input("Оценка: "))
                    grades[which] = grade
                elif which != -1:
                    print("Неверное число оценки")
            else:
                print("Студент не найден")
    elif menu_choice != 6:
        print_menu()
```

...и вот пример вывода:

```
1. Добавить студента
2. Удалить студента
3. Вывести оценки
4. Записать оценку
5. Меню вывода
6. Выход
Выбор меню (1-6):     3
       hw ch 1        hw ch 2        quiz    hw ch 3        test
#Max      25             25            50       25            100

Выбор меню (1-6):     5
1. Добавить студента
2. Удалить студента
3. Печатные оценки
4. Рекордная оценка
5. Меню печати
6. Выход

Выбор меню (1-6):     1
Добавить студента: Билл

Выбор меню (1-6): 4
Записать оценку
Студент: Билл
Введите номер оценки для записи
Введите 0 (ноль) для выхода
1      hw ch 12       hw ch 23       quiz    4       hw ch 35       test
0              0              0              0              0
Изменить которую оценку: 1
Оценка: 25
Изменить которую оценку: 2
Оценка: 24
Изменить которую оценку: 3
Оценка: 45
Изменить которую оценку: 4
Оценка: 23
Изменить которую оценку: 5
Оценка: 95
Изменить которую оценку: 0

Выбор меню (1-6):     3
       hw ch 1        hw ch 2        quiz    hw ch  3       test
#Max   25               25            50       25           100
Билл   25               24            45       23           95

Выбор меню (1-6): 6
```

Вот как работает программа. По сути, переменная `students` — это словарь, ключами которого являются имена студентов, а значениями — их оценки. Первые две строки просто создают два списка. Следующая строка `students = {'#Max': max_points}` создает новый словарь с ключом `{#Max}` и значением `[25, 25, 50, 25, 100]` (так как именно таким было значение `max_points` при назначении) (я использую ключ `#Max`, так как `#` сортируется впереди любых буквенных символов). Далее определяется `print_menu`. Затем определяется функция `print_all_grades` в строках:

```
def print_all_grades():
    print('\t',end=" ")
    for i in range(len(assignments)):
        print(assignments[i], '\t',end=" ")
    print()
    keys = list(students.keys()) keys.sort()
    for x in keys:
        print(x, '\t',end=' ')
        grades = students[x]
        print_grades(grades)
```

Обратите внимание, что сначала ключи извлекаются из словаря `students` с помощью функции `keys` в строке `keys = list(students.keys())`. `keys` — это итерируемая переменная, и она преобразуется в список, чтобы можно было использовать все функции для списков. Далее ключи сортируются в строке `keys.sort()`. `for` используется для перебора всех ключей. Оценки хранятся в виде списка внутри словаря, поэтому задание `grades = students[x]` дает оценкам список, хранящийся по ключу `x`.

Функция `print_grades` просто выводит список и определена несколькими строками позже.

Последующие строки программы реализуют различные опции меню. Строка `students[name] = [0] * len(max_points)` добавляет студента к ключу его имени. Нотация `[0] * len(max_points)` просто создает список из 0, длина которого равна длине списка `max_points`.

Запись «Удалить студента» просто удаляет студента, как в примере с телефонной книгой. Выбор записи оценок немного сложнее. Оценки извлекаются в строке `grades = students[name]`, которая получает ссылку на оценки имени студента. Затем оценка записывается в строке `grades[which] = grade`. Вы можете заметить, что оценка никогда не помещается обратно в словарь `students` (как в случае со `students[name] = grades`). Причина отсутствия этого утверждения в том, что `grades` — это фактически другое имя для `students[name]`, и поэтому изменение `grades` изменяет `student[name]`.

Словари предоставляют простой способ связать ключи со значениями. Их можно использовать для удобного отслеживания данных, которые привязаны к различным ключам.

Глава 14

Использование модулей

Вот упражнение по набору текста в этой главе (назовите его cal.py (import на самом деле ищет файл с именем calendar.py и считывает его. Если файл называется calendar.py и он видит «import calendar», то пытается прочитать его сам, что в лучшем случае работает плохо)):

```
import calendar
year = int(input("Введите год: "))
calendar.prcal(year)
```

Вот часть полученного результата:

```
Введите год: 2001

                                  2001

      January                   February                   March
Mo Tu We Th Fr Sa Su   Mo Tu Wd Th Fr Sa Su     Mo Tu Wd Th Fr Sa Su
 1  2  3  4  5  6  7                1  2  3  4               1  2  3  4
 8  9 10 11 12 13 14    5  6  7  8  9 10 11      5  6  7  8  9 10 11
15 16 17 18 19 20 21   12 13 14 15 16 17 18     12 13 14 15 16 17 18
22 23 24 25 26 27 28   19 20 21 22 23 24 25     19 20 21 22 23 24 25
29 30 31               26 27 28                 26 27 28 29 30 31
```

(Я пропустил часть вывода, но, думаю, вы поняли идею.) Итак, что же делает программа? Первая строка `import calendar` использует новую команду `import`. Команда `import` загружает модуль (в данном случае модуль `calendar`). Чтобы узнать, какие команды доступны в стандартных модулях, загляните в справочник по библиотекам для Python (если вы его скачали) или зайдите на сайт http://docs.python.org/3/library/. Если вы посмотрите документацию к модулю `calendar`, там будет указана функция `prcal`, которая печатает календарь на год.

Строка `calendar.prcal(year)` использует эту функцию. В общем, чтобы использовать модуль, импортируйте его, а затем используйте `module_name.function` для функций в модуле. Другой вариант написания программы:

```
from calendar import prcal

year = int(input("Введите год: "))
prcal(year)
```

Эта версия импортирует определенную функцию из модуля. Вот еще одна программа, использующая библиотеку Python (назовите ее как-нибудь вроде clock.py) (нажмите Ctrl и клавишу 'c' одновременно, чтобы завершить программу):

```
from time import time, ctime

prev_time = ""
```

```
while True:
    the_time = ctime(time())
    if prev_time != the_time:
        print("Время:", ctime(time()))
        prev_time = the_time
```

Часть вывода:

```
Время: Sun Aug 20 13:40:04 2000
Время: Sun Aug 20 13:40:05 2000
Время: Sun Aug 20 13:40:06 2000
Время: Sun Aug 20 13:40:07 2000

Traceback (innermost last):
 File "clock.py", line 5, in ?
    the_time = ctime(time())

KeyboardInterrupt
```

Вывод, конечно, бесконечен, поэтому я его отменил (или вывод, по крайней мере, продолжается до нажатия Ctrl+C). Программа просто выполняет бесконечный цикл (`True` всегда истинно, поэтому `while True:` идет вечно) и каждый раз проверяет, изменилось ли время, и печатает его, если изменилось. Обратите внимание, как в строке `from time import time, ctime` используются несколько имен после оператора `import`.

Библиотека Python содержит множество полезных функций. Эти функции расширяют возможности ваших программ, а многие из них могут упростить программирование на Python.

Упражнения

Перепишите программу `high_low.py` из главы «Решения», чтобы она использовала случайное целое число от 0 до 99 вместо жестко закодированного 78. Используйте документацию Python, чтобы найти подходящий модуль и функцию для этого.

Решение:

```
from random import randint
number = randint(0, 99)
guess = -1
while guess != number:
    guess = int(input ("Угадайте число: "))
    if guess > number:
        print("Слишком много")
    elif guess < number:
        print("Слишком мало")
print("То, что нужно")
```

Глава 15

Подробнее о списках

Мы уже познакомились со списками и способами их использования. Теперь, когда у вас есть дополнительная информация, я расскажу о списках более подробно. Сначала мы рассмотрим другие способы доступа к элементам списка, а затем поговорим о копировании.

Вот несколько примеров использования индексации для доступа к одному элементу списка:

```
>>> some_numbers = ['ноль', 'один', 'два', 'три', 'четыре', 'пять'].
>>> some_numbers[0]
'ноль'
>>> some_numbers[4]
'четыре'
>>> some_numbers[5]
'пять'
```

Все эти примеры должны показаться вам знакомыми. Если вам нужен первый элемент в списке, просто посмотрите на индекс 0. Второй элемент — индекс 1 и так далее по списку. Однако что если вам нужен последний элемент в списке? Одним из способов может быть использование функции `len()`, например `some_numbers[len(some_numbers) - 1]`. Этот способ работает, поскольку функция `len()` всегда возвращает последний индекс плюс один. Тогда вторым от последнего будет `some_numbers[len(some_numbers) - 2]`. Есть и более простой способ сделать это. В Python последний элемент всегда имеет индекс -1. Предпоследний — это индекс -2 и так далее. Вот еще несколько примеров:

```
>>> some_numbers[len(some_numbers) - 1]
'пять'
>>> some_numbers[len(some_numbers) - 2]
'четыре'
>>> some_numbers[-1]
'пять'
>>> some_numbers[-2]
'четыре'
>>> some_numbers[-6]
'ноль'
```

Таким образом, любой элемент списка может быть проиндексирован двумя способами: спереди и сзади.

Еще один полезный способ проникнуть в части списков — это использование нарезки. Вот еще один пример, дающий представление о том, для чего их можно использовать:

```
>>> things = [0, 'Фред', 2, 'S.P.A.M.', 'Чулок', 42, 'Джек', 'Джилл']
>>> things[0]
0
```

```
>>> things[7]
'Джилл'
>>> things[0:8]
[0, 'Фред', 2, 'S.P.A.M.', 'Чулок', 42, 'Джек', 'Джилл'].
>>> things[2:4]
[2, 'S.P.A.M.']
>>> things[4:7]
['Чулок', 42, 'Джек'].
>>> things[1:5]
['Фред', 2, 'S.P.A.M.', 'Чулок'].
```

Нарезка используется для возврата части списка. Оператор нарезки имеет вид `things[first_index:last_index]`. Нарезка разрезает список до `first_index` и до `last_index` и возвращает части между ними. Вы можете использовать оба типа индексации:

```
>>> things[-4:-2]
['Чулок', 42].
>>> things[-4]
'Чулок'
>>> things[-4:6]
['Чулок', 42].
```

Еще один трюк с нарезкой — неуказанный индекс. Если первый индекс не указан, предполагается начало списка. Если не указан последний индекс, то предполагается вся остальная часть списка. Вот несколько примеров:

```
>>> things[:2]
[0, 'Fred']
>>> things[-2:]
['Джек', 'Джилл'].
>>> things[:3]
[0, 'Фред', 2]
>>> things[:-5]
[0, 'Фред', 2]
```

Вот пример программы (на основе HTML) (скопируйте и вставьте определение стихотворения, если хотите):

```
poem = ["<B>", "Jack", "and", "Jill", "</B>", "went", "up", "the",
"hill", "to", "<B>", "fetch", "a", "pail", "of", "</B>",
"water.", "Jack", "fell", "<B>", "down", "and", "broke",
"</B>", "his", "crown", "and", "<B>", "Jill", "came",
"</B>", "tumbling", "after"].

def get_bolds(text):
    true = 1
    false = 0
    ## is_bold сообщает, выделен ли в данный момент полужирным (bold)
    ## шрифтом участок текста.
    is_bold = false
    ## start_block - это индекс начала не выделенного полужирным
```

```
    ## участка текста или выделенного полужирным участка.
    start_block = 0
    for index in range(len(text)):
        ## Обработка начала полужирного текста
        if text[index] == "<B>":
            if is_bold:
                print("Error: Extra Bold")
            ## print "Not Bold:", text[start_block:index]
            is_bold = true
            start_block = index + 1
        ## Обработка конца полужирного текста
        ## Помните, что последнее число в срезе - это индекс
        ## после последнего использованного индекса.
        if text[index] == "</B>":
                if not is_bold:
                    print("Error: Extra Close Bold")
                print("Bold [", start_block, ":", index, "]",
text[start_block:index])
                is_bold = false
                start_block = index + 1

get_bolds(poem)
```

...с выводом:

```
Bold [ 1 : 4 ] ['Jack', 'and', 'Jill']
Bold [ 11 : 15 ] ['fetch', 'a', 'pail', 'of'].
Bold [ 20 : 23 ] ['down', 'and', 'broke'].
Bold [ 28 : 30 ] ['Jill', 'came'].
```

Функция `get_bold()` получает список, разбитый на слова и лексемы. В качестве искомых лексем выступают `<B>`, с которого начинается полужирный текст, и `</B>`, которым заканчивается полужирный текст. Функция `get_bold()` перебирает и ищет начальные и конечные лексемы.

Следующая особенность списков — их копирование. Если вы попробуете сделать что-то простое, например:

```
>>> a = [1, 2, 3]
>>> b = a
>>> print(b)
[1, 2, 3]
>>> b[1] = 10
>>> print(b)
[1, 10, 3]
>>> print(a)
[1, 10, 3]
```

Вы, вероятно, получите удивительный результат, поскольку изменение `b` привело к изменению `a`. Дело в том, что оператор `b = a` делает `b` *ссылкой* на `a`. Это означает, что `b` можно рассматривать как другое имя для `a`. Следо-

вательно, любое изменение b изменяет и a. Однако некоторые присваивания не создают двух имен для одного списка:

```
>>> a = [1, 2, 3]
>>> b = a * 2
>>> print(a)
[1, 2, 3]
>>> print(b)
[1, 2, 3, 1, 2, 3]
>>> a[1] = 10
>>> print(a)
[1, 10, 3]
>>> print(b)
[1, 2, 3, 1, 2, 3]
```

В этом случае b не является ссылкой на a, поскольку выражение `a * 2` создает новый список. Оператор `b = a * 2` дает b ссылку на `a * 2`, а не ссылку на a. Все операции присваивания создают ссылку. Когда вы передаете список в качестве аргумента функции, вы тоже создаете ссылку. В большинстве случаев вам не нужно беспокоиться о создании ссылок, а не копий. Однако когда вам нужно внести изменения в один список, не меняя при этом имя другого списка, вы должны быть уверены, что действительно создали копию.

Существует несколько способов создания копии списка. Самый простой, который работает чаще всего, — это оператор нарезки, поскольку он всегда создает новый список, даже если он является фрагментом целого списка:

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b[1] = 10
>>> print(a)
[1, 2, 3]
>>> print(b)
[1, 10, 3]
```

Использование нарезки `[:]` создает новую копию списка. Однако этим способом копируется только внешний список. Любой подсписок внутри него по-прежнему является ссылкой на подсписок в исходном списке. Поэтому, когда список содержит списки, внутренние списки также должны быть скопированы. Вы можете сделать это вручную, но в Python уже есть модуль для этого. Вы используете функцию `deepcopy` модуля `copy`:

```
>>> import copy
 >>> a = [[1, 2, 3], [4, 5, 6]]
>>> b = a[:]
>>> c = copy.deepcopy(a)
>>> b[0][1] = 10
>>> c[1][1] = 12
>>> print(a)
[[1, 10, 3], [4, 5, 6]]
```

```
>>> print(b)
[[1, 10, 3], [4, 5, 6]]
>>> print(c)
[[1, 2, 3], [4, 12, 6]]
```

Прежде всего обратите внимание, что `a` — это список списков. Затем обратите внимание, что при выполнении `b[0][1] = 10` изменяются и `a`, и `b`, а `c` — нет. Это происходит потому, что при использовании оператора нарезки внутренние массивы все еще являются ссылками. Однако при использовании `deepcopy` (глубокого копирования) `c` был полностью скопирован.

Так стоит ли мне беспокоиться о ссылках каждый раз, когда я использую функцию или `=`? Хорошая новость заключается в том, что вам нужно беспокоиться о ссылках только при использовании словарей и списков. Числа и строки создают ссылки при присвоении, но каждая операция над числами и строками, которая их изменяет, создает новую копию, поэтому вы никогда не сможете изменить их неожиданно. О ссылках нужно думать, когда вы изменяете список или словарь.

Сейчас вы, вероятно, задаетесь вопросом, зачем вообще используются ссылки. Основная причина — скорость. Гораздо быстрее сделать ссылку на список из тысячи элементов, чем копировать все элементы. Другая причина заключается в том, что это позволяет вам иметь функцию для модификации введенного списка или словаря. Просто помните о ссылках, если у вас когда-нибудь возникнет странная проблема с изменением данных, когда это не должно происходить.

Глава 16

Месть строк

А теперь представляем вам крутой трюк, который можно проделать со строками:

```
def shout(string):
    for character in string:
        print("Дайте мне " + character)
        print("'" + character + "'")

shout("Lose")

def middle(string):
    print("Средний символ:", string[len(string) // 2])

middle("abcdefg")
middle("The Python Programming Language")
middle("Atlanta")
```

И на выходе получаем:

```
Дайте мне L
'L'
Дайте мне o.
'o'
Дайте мне s.
's'
Дайте мне e.
'e'
Средний символ: d
Средний символ: r
Средний символ: a
```

Эти программы демонстрируют, что строки похожи на списки, несколькими способами. Функция `shout()` показывает, что циклы `for` можно использовать со строками так же, как и со списками. Процедура `middle` показывает, что строки также могут использовать функцию `len()`, индексы и срезы массивов. Большинство функций списков работает и со строками.

Следующая функция демонстрирует некоторые особенности строк:

```
def to_upper(string):
    ## Преобразование строки в верхний регистр
    upper_case = ""
    for character in string:
        if 'a' <= character <= 'z':
            location = ord(character) - ord('a')
            new_ascii = location + ord('A')
            character = chr(new_ascii)
        upper_case = upper_case + character
    return upper_case

print(to_upper("Это текст"))
```

...с выводом:

```
ЭТО ТЕКСТ
```

Это работает потому, что компьютер представляет символы строки как числа от 0 до 1 114 111. Например, 'A' — это 65, 'B' — 66, а א — 1488. Значения — это значения юникода. В Python есть функция `ord()` (сокращение от ordinal), которая возвращает символ в виде числа. Также есть соответствующая функция `chr()`, которая преобразует число в символ. С учетом этого программа должна стать понятной. Первая деталь — это строка: `if 'a' <= character <= 'z':`, которая проверяет, является ли буква строчной. Если да, то используются следующие строки. Сначала она преобразуется в местоположение, так что a = 0, b = 1, c = 2 и так далее, с помощью строки: `location = ord(character) - ord('a')`. Затем находится новое значение `new_ascii = location + ord('A')`. Это значение преобразуется обратно в символ, который теперь находится в верхнем регистре. Обратите внимание, что если вам действительно нужен верхний регистр буквы, используйте `u=var.upper()`, который будет работать и с другими языками.

А теперь немного интерактивных упражнений по набору текста:

```
>>> # Целое число в строку
>>> 2
2
>>> repr(2)
'2'
>>> -123
-123
>>> repr(-123)
'-123'
>>> # Перевод строки в целое число
>>> "23"
'23'
>>> int("23")
23
>>> "23" * 2
'2323'
>>> int("23") * 2
46
>>> # Преобразование числа с плавающей точкой в строку
>>> 1.23
1.23
>>> repr(1.23)
'1.23'
>>> # Преобразование чисел с плавающей точкой в целые
>>> 1.23
1.23
>>> int(1.23)
1
>>> int(-1.23)
-1
>>> # Перевод строки в число с плавающей точкой
>>> float("1.23")
1.23
>>> "1.23"
'1.23'
>>> float("123")
123.0
```

Если вы еще не догадались, функция `repr()` может преобразовать целое число в строку, а функция `int()` — строку в целое число. Функция `float()` может преобразовать строку в число. Функция `repr()` возвращает печатное представление чего-либо. Вот несколько примеров:

```
>>> repr(1)
'1'
>>> repr(234.14)
'234.14'
>>> repr([4, 42, 10])
'[4, 42, 10]'
```

Функция `int()` пытается преобразовать строку (или число с плавающей точкой) в целое число. Существует также похожая функция `float()`, которая преобразует целое число или строку в число с плавающей точкой. Еще одна функция, которая есть в Python, — это функция `eval()`. Функция `eval()` берет строку и возвращает данные того типа, который, по мнению Python, она нашла. Например:

```
>>> v = eval('123')
>>> print(v, type(v))
123 <type 'int'>
>>> v = eval('645.123')
>>> print(v, type(v))
645.123 <type 'float'>
>>> v = eval('[1, 2, 3]')
>>> print(v, type(v))
[1, 2, 3] <type 'list'>
```

Если вы используете функцию `eval()`, убедитесь, что она возвращает тот тип, который вы ожидаете. Одной из полезных строковых функций является метод `split()`. Вот пример:

```
>>> "Это куча слов".split()
['Это', 'куча', 'слов'].
>>> text = "Первая партия, вторая партия, третья, четвертая".
>>> text.split(",")
['Первая партия', 'вторая партия', 'третья', 'четвертая'].
```

Обратите внимание, как `split()` преобразует строку в список строк. По умолчанию строка разделяется пробелами или необязательным аргументом (в данном случае запятой). Вы также можете добавить еще один аргумент, который сообщает функции `split()`, сколько раз разделитель будет использоваться для разделения текста. Например:

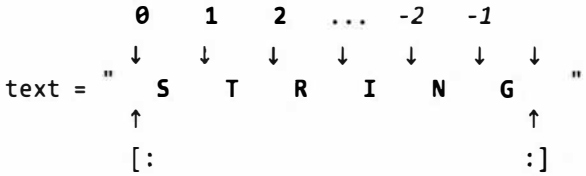
```
>>> list = text.split(",")
>>> len(list)
4
>>> list[-1]
' четвертый'
>>> list = text.split(",", 2)
>>> len(list)
3
>>> list[-1]
' третий, четвертый'
```

Нарезка строк (и списков)

Строки можно разрезать на части — так же, как это было показано для списков в предыдущей главе, — с помощью оператора нарезки `[]`. Оператор нарезки работает так же, как и раньше: `text[first_index:last_index]`

(в очень редких случаях могут быть еще одно двоеточие и третий аргумент, как в примере, показанном ниже).

Чтобы не запутаться в номерах индексов, проще всего рассматривать их как *места обрезки*, позволяющие разрезать строку на части. Вот пример, показывающий места обрезки и их индексные номера для простой текстовой строки:



Обратите внимание, что индексы, выделенные полужирным, отсчитываются от начала строки, а курсивные — от конца строки назад. (Важно, что в конце строки нет курсивного -0, что могло бы показаться логичным. Поскольку `-0 == 0`, -0 также означает «начало строки»). Теперь мы готовы использовать индексы для операций нарезки:

```
text[1:4]     →     "TRI"
text[:5]      →     "STRIN"
text[:-1]     →     "STRIN"
text[-4:]     →     "RING"
text[2]       →     "R"
text[:]       →     "STRING"
text[::-1]    →     "GNIRTS"
```

`text[1:4]` дает нам всю текстовую строку между местами обрезки 1 и 4, `"TRI"`. Если вы опустите один из аргументов `[first_index:last_index]`, то получите начало или конец строки по умолчанию: `text[:5]` дает `"STRIN"`. Для аргументов `first_index` и `last_index` мы можем использовать любую из схем нумерации: `text[:-1]` дает то же самое, что и `text[:5]`, поскольку индекс -1 в данном случае находится там же, где и 5. Если мы не используем аргумент, содержащий двоеточие, число обрабатывается по-другому: `text[:2]` дает один символ после второй точки обрезания, `"R"`. Специальная операция нарезки `text[:]` означает «от начала до конца» и создает копию всей строки (или списка, как было показано в предыдущей главе).

И последнее, но не менее важное: операция нарезки может иметь второе двоеточие и третий аргумент, который интерпретируется как «размер шага»: `text[::-1]` — это текст от начала до конца с размером шага -1. -1 означает «каждый символ, но в другом направлении». `"STRING"` в обратном направлении — это `"GNIRTS"` (проверьте длину шага 2, если вы еще не поняли, о чем идет речь).

Все эти операции нарезки работают и со списками. В этом смысле строки — это просто частный случай списков, где элементами списка явля-

ются одиночные символы. Просто запомните концепцию мест обрезания, и индексы для нарезки станут гораздо менее запутанными.

Примеры

```
# Эта программа требует отличного понимания десятичных чисел.
def to_string(in_int):
    """Преобразует целое число в строку"""
    out_str = ""
    prefix = ""
    if in_int < 0:
        prefix = "-"
        in_int = -in_int
    while in_int // 10 != 0:
        out_str = str(in_int % 10) + out_str
        in_int = in_int // 10
    out_str = str(in_int % 10) + out_str
    return prefix + out_str

def to_int(in_str):
    """Преобразует строку в целое число"""
    out_num = 0
    if in_str[0] == "-":
        multiplier = -1
        in_str = in_str[1:]
    else:
        multiplier = 1
    for c in in_str:
        out_num = out_num * 10 + int(c)
    return out_num * multiplier

print(to_string(2))
print(to_string(23445))
print(to_string(-23445))
print(to_int("14234"))
print(to_int("12345"))
print(to_int("-3512"))
```

Выходные данные:

```
2
23445
-23445
14234
12345
-3512
```

Глава 17

Файловый ввод/вывод

Вот простой пример файлового ввода/вывода:

```
# Записать файл
with open("test.txt", "wt") as out_file:
    out_file.write("Этот текст отправляется в файл вывода\nПосмотрите
на него и увидите!")

# Чтение файла
with open("test.txt", "rt") as in_file:
    text = in_file.read()

print(text)
```

Вывод и содержимое файла `test.txt` таковы:

```
Этот текст отправляется в файл вывода
Посмотрите на него и увидите!
```

Обратите внимание, что он записал файл `test.txt` в папку, из которой вы запустили программу. Символ `\n` в строке указывает Python начинать *с новой строки*.

Обзор файлового ввода-вывода:

- Получите объект файла с помощью функции `open`.
- Чтение или запись в файловый объект (в зависимости от того, как он был открыт).
- Если вы не использовали функцию `with` для открытия файла, вам придется закрыть его вручную.

Первым шагом будет получение объекта файла. Для этого используется функция `open`. Формат: `file_object = open(filename, mode)`, где `file_object` — переменная для размещения файлового объекта, `filename` — строка с именем файла, а `mode` — `"rt"` для чтения файла как текста или `"wt"` для записи файла как текста (и некоторые другие, которые мы здесь опустим). Далее можно вызывать функции файловых объектов. Две наиболее распространенные функции — `read` и `write`. Функция `write` добавляет строку в конец файла. Функция `read` считывает следующую запись в файле и возвращает ее в виде строки. Если аргумент не указан, она вернет весь файл (как это сделано в примере).

Теперь перед вами новая версия программы телефонных номеров, которую мы написали ранее:

```
def print_numbers(numbers):
    print("Телефонные номера:")
    for k, v in numbers.items():
        print("Имя:", k, "\tНомер:", v)
    print()

def add_number(numbers, name, number):
```

```
    numbers[name] = number

def lookup_number(numbers, name):
    if name in numbers:
        return "Номер - " + numbers[name]
    else:
        return name + " не найдено"

def remove_number(numbers, name):
    if name in numbers:
        del numbers[name]
    else:
        print(name," не найдено")

def load_numbers(numbers, filename):
    in_file = open(filename, "rt")
    while True:
        in_line = in_file.readline()
        if not in_line:
            break
        in_line = in_line[:-1]
        name, number = in_line.split(",")
        numbers[имя] = number
    in_file.close()

def save_numbers(numbers, filename):
    out_file = open(filename, "wt")
    for k, v in numbers.items():
        out_file.write(k + "," + v + "\n")
    out_file.close()

def print_menu():
    print('1. Вывод телефонных номеров')
    print('2. Добавить номер телефона')
    print('3. Удалить номер телефона')
    print('4. Поиск телефонного номера')
    print('5. Загрузить номера')
    print('6. Сохранить номера')
    print('7. Выход')
    print()

phone_list = {}
menu_choice = 0
print_menu()
while True:
    menu_choice = int(input("Введите число (1-7): "))
    if menu_choice == 1:
        print_numbers(phone_list)
```

```
    elif menu_choice == 2:
        print("Добавить имя и номер")
        name = input("Имя: ")
        phone = input("Номер: ")
        add_number(phone_list, name, phone)
    elif menu_choice == 3:
        print("Удалить имя и номер")
        name = input("Имя: ")
        remove_number(phone_list, name)
    elif menu_choice == 4:
        print("Поиск номера")
        name = input("Имя: ")
        print(lookup_number(phone_list, name))
    elif menu_choice == 5:
        filename = input("Имя файла для загрузки: ")
        load_numbers(phone_list, filename)
    elif menu_choice == 6:
        filename = input("Имя файла для сохранения: ")
        save_numbers(phone_list, filename)
    elif menu_choice == 7:
        break
    else:
        print_menu()
print("До свидания")
```

Обратите внимание, что теперь она включает сохранение и загрузку файлов. Вот некоторые результаты двухкратного запуска:

```
1. Вывод телефонных номеров
2. Добавить номер телефона
3. Удалить номер телефона
4. Поиск телефонного номера
5. Загрузить номера
6. Сохранить номера
7. Выход

Введите номер (1-7): 2
Добавить имя и номер
Имя: Джил
Номер: 1234
Введите номер (1-7): 2
Добавить имя и номер
Имя: Фред
Номер: 4321
Введите число (1-7): 1
Номера телефонов:
```

```
Имя: Джил      Номер: 1234
Имя: Фред      Номер: 4321

Введите число (1-7): 6
Имя файла для сохранения: numbers.txt
Введите число (1-7): 7
До свидания

1. Вывод телефонных номеров
2. Добавить номер телефона
3. Удалить номер телефона
4. Поиск телефонного номера
5. Загрузить номера
6. Сохранить номера
7. Выход

Введите число (1-7): 5
Имя файла для загрузки: numbers.txt
Введите номер (1-7): 1
Телефонные номера:
Имя: Джил      Номер: 1234
Имя: Фред      Номер: 4321

Введите число (1-7): 7
До свидания
```

Новыми частями этой программы являются:

```
def load_numbers(numbers, filename):
    in_file = open(filename, "rt")
    while True:
        in_line = in_file.readline()
        if not in_line:
            break
        in_line = in_line[:-1]
        name, number = in_line.split(",")
        numbers[name] = number
    in_file.close()

def save_numbers(numbers, filename):
    out_file = open(filename, "wt")
    for k, v in numbers.items():
        out_file.write(k + "," + v + "\n")
out_file.close()
```

Сперва мы рассмотрим часть программы, связанную с сохранением. Сначала создается объект файла командой `open(filename, "wt")`. Затем он создает строку для каждого телефонного номера командой `out_file.write(k + "," + v + "\n")`. В результате записывается строка, содержащая имя, запятую, номер и после него новую строку.

Часть загрузки немного сложнее. Она начинается с получения объекта файла. Затем используется цикл `while True:` для продолжения цикла до тех пор, пока не будет встречен оператор `break`. Затем он получает строку со строкой `in_line = in_file.readline()`. Функция `readline` вернет пустую строку, когда будет достигнут конец файла. Оператор `if` проверяет это и выходит из цикла `while`, когда это происходит. Конечно, если бы функция `readline` не возвращала новую строку в конце строки, то не было бы способа определить, пустая это строка или конец файла, поэтому новая строка остается в том, что возвращает `readline`. Следовательно, мы должны избавиться от новой строки. Строка `in_line = in_line[:-1]` делает это за нас, отбрасывая последний символ. Далее строка `name, number = in_line.split(",")` разделяет строку через запятую на имя и число. Затем они добавляются в словарь `numbers`.

Продвинутое использование файлов .txt

Возможно, вы скажете себе: «Я знаю, как читать и записывать в текстовый файл, но что если я хочу распечатать файл, не открывая другую программу?»

Есть несколько различных способов сделать это. Самый простой способ — открыть другую программу, но при этом все делается в коде Python, и пользователю не нужно указывать файл для печати. Этот способ предполагает вызов подпроцесса другой программы. Помните файл, в который мы записывали вывод в приведенной выше программе? Давайте воспользуемся этим файлом. Помните, что для предотвращения некоторых ошибок в этой программе используются понятия из следующей главы. Пожалуйста, не поленитесь вернуться к этому примеру после изучения следующей главы.

```
import subprocess
def main():
    try:
        print("Эта небольшая программа вызывает функцию печати
в приложении Notepad")
        #Напечатаем файл, который мы создали в программе выше
        subprocess.call(['notepad','/p','numbers.txt'])
    except WindowsError:
        print("Вызываемый подпроцесс не существует или не может быть
вызван.")

main()
```

Вызов `subprocess.call` принимает три аргумента. Первым аргументом, в контексте данного примера, должно быть имя программы, из которой вы хотите вызвать подпроцесс печати. Вторым аргументом должен быть конкретный подпроцесс в этой программе. Для простоты понимания усвойте, что в данной программе `'/p'` — это подпроцесс, используемый для доступа к принтеру через указанное приложение. Последним аргументом должно быть имя файла, который вы хотите отправить в подпроцесс печати. В данном случае это тот же файл, который использовался ранее в этой главе.

Упражнения

Теперь измените программу оценки из раздела «Словари» так, чтобы она использовала файловый ввод-вывод для ведения записей о студентах.

Решение:

```
assignments = ['hw ch 1', 'hw ch 2','quiz  ', 'hw ch 3', 'test']
students = { }

def load_grades(gradesfile):
    inputfile = open(gradesfile, "r")
    grades = [ ]
    while True:
        student_and_grade = inputfile.readline()
        student_and_grade = student_and_grade[:-1]
        if not student_and_grade:
            break
        else:
            studentname, studentgrades = student_and_grade.split(",")
            studentgrades = studentgrades.split(" ")
            students[studentname] = studentgrades
    inputfile.close()
    print("Оценки загружены.")

def save_grades(gradesfile):
    outputfile = open(gradesfile, "w")
    for k, v in students.items():
        outputfile.write(k + ",")
        for x in v:
            outputfile.write(str(x) + " ")
        outputfile.write("\n")
    outputfile.close()
    print("Оценки сохранены.")

def print_menu():
    print("1. Добавить студента")
    print("2. Удалить студента")
    print("3. Загрузить оценки")
    print("4. Записать оценку")
    print("5. Распечатать оценки")
    print("6. Сохранить оценки")
    print("7. Меню печати")
    print("9. Выход")

def print_all_grades():
    if students:
        keys = sorted(students.keys())
        print('\t', end=' ')
        for x in assignments:
            print(x, '\t', end=' ')
```

```
        print()
        for x in keys:
            print(x, '\t', end=' ')
            grades = students[x]
            print_grades(grades)
    else:
        print("Нет оценок для печати.")

def print_grades(grades):
    for x in grades:
        print(x, '\t', end=' ')
    print()

print_menu()
menu_choice = 0
while menu_choice != 9:
    print()
    menu_choice = int(input("Выбор меню: "))
    if menu_choice == 1:
        name = input("Студент, которого нужно добавить: ")
        students[name] = [0] * len(assignments)
    elif menu_choice == 2:
        name = input("Студент, которого нужно удалить: ")
        if name in students:
            del students[name]
        else:
            print("Студент:", name, "не найден")
     elif menu_choice == 3:
        gradesfile = input("Загрузить оценки из какого файла? ")
        load_grades(gradesfile)
    elif menu_choice == 4:
        print("Записать оценку")
        name = input("Студент: ")
        if name in students:
            grades = students[name]
            print("Введите номер оценки для записи")
            print("Введите 0 (ноль) для выхода")
            for i,x in enumerate(assignments):
                print(i + 1, x, '\t', end=' ')
            print()
            print_grades(grades)
            which = 1234
            while which != -1:
                which = int(input("Изменить какую оценку: "))
                which -= 1
                if 0 <= which < len(grades):
                    grade = input("Grade: ") # Замените float(input())
на input(), чтобы избежать ошибки
```

```
                    grades[which] = grade
                elif which != -1:
                    print("Неверный номер оценки")
            else:
                print("Студент не найден")
        elif menu_choice == 5:
            print_all_grades()
        elif menu_choice == 6:
            gradesfile = input("Сохранить оценки в какой файл? ")
            save_grades(gradesfile)
        elif menu_choice != 9:
            print_menu()
```

Глава 18

Работа с несовершенным, или Как обрабатывать ошибки

Закрытие файлов при помощи оператора with

Мы используем оператор `with` для открытия и закрытия файлов.[1] [2]

```
with open("in_test.txt", "rt") as in_file:
    with open("out_test.txt", "wt") as out_file:
        text = in_file.read()
        data = parse(text)
        results = encode(data)
        out_file.write(results)
print("Все сделано.")
```

Если в этом коде произойдет какая-то ошибка (один из файлов окажется недоступным, функция `parse()` «подавится» поврежденными данными и т. д.), оператор `with` гарантирует, что все файлы в конечном итоге будут закрыты должным образом. Закрытие файла означает, что файл «очищается» и «освобождается» нашей программой, чтобы его можно было использовать в другой программе.

Перехват ошибок с помощью try

Итак, у вас есть идеальная программа, которая работает безупречно, за исключением одной детали — она будет аварийно завершаться при неправильном вводе данных пользователем. Не бойтесь, ведь в Python есть

[1] Оператор with (http://docs.python.org/3.4/reference/compound_stmts.html#the-with-statement)
[2] Оператор with в Python на примере (http://preshing.com/20110920/the-python-with-statement-by-example/)

специальная управляющая структура. Она называется try и пытается что-то сделать. Вот пример программы с проблемой:

```
print("Введите Control C или -1 для выхода")
number = 1
while number != -1:
    number = int(input("Введите число: "))
    print("Вы ввели:", число)
```

Обратите внимание, что при вводе @#& выводится что-то вроде:

```
Traceback (most recent call last):
 File "try_less.py", line 4, in <module>
    number = int(input("Введите число: "))
ValueError: invalid literal for int() with base 10: '\\@#&'
```

Как видите, функция int() «недовольна» числом @#& (как и должно быть). Последняя строка показывает, в чем проблема; Python обнаружил ошибку ValueError. Как наша программа может с этим справиться? Во-первых, мы поместим место, где могут возникнуть ошибки, в блок try, а во-вторых, сообщим Python, как мы хотим обрабатывать ValueErrors. Это можно сделать в следующей программе:

```
print("Введите Control C или -1 для выхода")
number = 1
 while number != -1:
    try:
        number = int(input("Введите число: "))
        print("Вы ввели:", number)
    except ValueError:
        print("Это было не число").
```

Теперь, когда мы запускаем новую программу и даем ей @#&, она сообщает нам «Это было не число» и продолжает делать то, что делала раньше. Когда в программе постоянно возникает ошибка и вы знаете, как ее обработать, поместите код в блок try, а способ обработки ошибки — в блок except.

Упражнения

Обновите программу телефонных номеров (из главы «Словари»), чтобы она не «вылетала», если пользователь не вводит никаких данных в меню.

Глава 19

Конец

Итак, мы подошли к концу, а может быть, и к началу. Этот учебник размещен на Викискладе, поэтому не стесняйтесь вносить в него улучшения. Если вы хотите узнать больше о Python, то вам подойдут «Самоучитель Python» (http://docs.python.org/3/tutorial/index.html) Гвидо ван Россума,

который упоминался в предисловии, викибук по программированию на Python, книга авторства C. H. Swaroop, которая также упоминалась выше, «Практический самоучитель по Python от доктора Эндрю Н. Харрингтона» (http://anh.cs.luc.edu/python/hands-on/3.1)/handsonHtml/index.html) и прочие викиучебники по Python.

Надеюсь, эта книга охватывает все, что вам было нужно, чтобы начать программировать. Спасибо всем, кто прислал мне письма по этому поводу. Мне было приятно их читать, даже если я не всегда был лучшим «ответчиком».

Счастливого программирования, пусть оно изменит вашу жизнь и мир.

Глава 20

Часто задаваемые вопросы

Как сделать графический интерфейс (GUI) в Python?

Вы можете использовать одну из этих библиотек: TKinter (https://docs.python.org/3.5/library/tkinter.html), PyQt (https://riverbankcomputing.com/software/pyqt/intro), PyGobject (https://wiki.gnome.org/Projects/PyGObject). Для действительно простой графики можно использовать графический режим черепахи `import turtle`.

Как сделать игру на Python?

Лучшим методом, вероятно, является использование PyGame по адресу http://pygame.org/.

Как сделать исполняемый файл из программы на Python?

Короткий ответ: Python — интерпретируемый язык, поэтому это невозможно. Длинный ответ: что-то похожее на исполняемый файл можно создать, взяв интерпретатор Python и файл, соединив их вместе и распространив. Подробнее об этой проблеме читайте на сайте http://www.python.org/doc/faq/programming/#how-can-i-create-a-stand-alone-binary-from-a-python-script.

Почему вы написали этот учебник от первого лица?

Когда-то ранняя версия книги была написана Джошем Коглиати, и она была размещена на его веб-странице http://www.honors.montana.edu/~jjc/easytut, и это было хорошо. Затем сервер rupert, как и все хорошее, что имеет начало, подошел к концу, и Джош перенес ее в Wikibooks, но написание от первого лица осталось. Если кто-то действительно хочет его изменить, я не буду против, но не вижу в этом особого смысла.

Мой вопрос остался без ответа.

Задайте вопрос на странице обсуждения или напишите одному из авторов.

Для получения ответов на другие часто задаваемые вопросы вы можете посмотреть версию самоучителя по Python для непрограммистов для Python версии 2.6 или обратиться к странице https://docs.python.org/3.5/faq/.

16+

Серия «Учимся программировать»

Справочное издание
Анықтамалық басылым

Джош Коглиати

PYTHON ДЛЯ НЕПРОГРАММИСТОВ САМОУЧИТЕЛЬ В ПРИМЕРАХ

Оформление обложки *С. А. Арутюнян*
Ответственный за выпуск *И. В. Резько*

Подписано в печать 03.04.2024.
Изготовлено в 2024 г.
Формат 70x100$^{1}/_{16}$. Бумага офсетная. Печать офсетная.
Гарнитура Noto Serif SemiCondensed.
Усл. печ. л. 7,8. Тираж 2000 экз. Заказ № 3158.

Общероссийский классификатор продукции ОК-034-2014 (КПЕС 2008);
58.11.1 — книги, брошюры печатные.

Произведено в Российской Федерации
Изготовитель: ООО «Издательство АСТ»
129085, Российская Федерация, г. Москва, Звездный бульвар, дом 21,
строение 1, комната 705, пом. I, этаж 7.
Адрес места осуществления деятельности по изготовлению продукции:
123112, Российская Федерация, г. Москва, Пресненская набережная, д. 6, стр. 2,
Деловой комплекс «Империя», 14, 15 этаж.
Наш электронный адрес: ask@ast.ru
Наш сайт: www.ast.ru Интернет-магазин: www.book24.ru

Өндіруші: «Издательство АСТ» ЖШҚ
129085, Ресей Федерациясы, Мәскеу, Звёздный бульвары, 21-үй,
1-құрылыс, 705-бөлме, I үй-жай, 7-қабат.
Өнім өндіру қызметін жүзеге асыру мекенжайы:
123112, Ресей Федерациясы, Мәскеу, Пресненская жағ., 6-үй, 2-құр.,
«Империя» іскерлік кешені, 14, 15-қабат
Біздің электрондық мекенжайымыз: www.ast.ru E-mail: ask@ast.ru
Интернет-магазин: www.book24.kz
Интернет-дукен: www.book24.kz
Импортер в Республику Казахстан и Представитель по приему претензий
в Республике Казахстан — ТОО РДЦ Алматы, г. Алматы.
Қазақстан Республикасына импорттаушы және Қазақстан Республикасында
наразылықтарды қабылдау бойынша өкіл — «РДЦ-Алматы» ЖШС,
Алматы қ., Домбровский көш., 3«а», Б литері офис 1.
Тел.: 8(727) 2 51 59 90,91, факс: 8 (727) 251 59 92 ішкі 107;
E-mail: RDC-Almaty@eksmo.kz, www.book24.kz
Тауар белгісі: «АСТ» Өндірілген жылы: 2024
Өнімнің жарамдылық мерзімі шектелмеген
Ресей Федерациясында өндірілген

Отпечатано с готовых файлов заказчика
в АО «Первая Образцовая типография»,
филиал «УЛЬЯНОВСКИЙ ДОМ ПЕЧАТИ»
432980, Россия, г. Ульяновск, ул. Гончарова, 14

Автор этого самоучителя по программированию на языке Python небезосновательно утверждает, что на самом деле существует только один способ научиться программировать: вы должны ежедневно читать код и с той же регулярностью писать его. Но чтобы начать читать на языке Python, который, по утверждению многих маститых разработчиков, идеально подходит для новичков благодаря своей доступности, простоте и вместе с тем функциональности, для начала необходимо изучить его «азбуку» и «структуру». Именно эта книга без скучных вводных лекций поможет вам освоить базовые навыки программирования на Python — свой первый код, пусть и состоящий из двух-трех строк, вы сможете написать сразу же после изучения вводной темы. Погрузиться с головой в мир программирования автор предлагает с помощью простых примеров, понятных новичку, а каждый шаг в создании, проверке и отладке вновь созданных программ подробно комментируется — четко, аргументированно и логично. Безусловно, эти качества являются главными для начинающего разработчика, и вы наверняка ими обладаете, поэтому вникнуть в основы программирования на Python для вас не составит особого труда.

книги для любого настроения здесь

ИЗДАТЕЛЬСКАЯ ГРУППА АСТ

www.ast.ru | www.book24.ru

vk.com/izdatelstvoast

ok.ru/izdatelstvoast

ISBN 978-5-17-162198-8

9 785171 621988