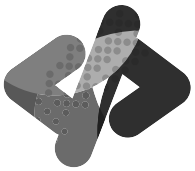
Майк МакГрат

# PYTHON

## ПРОГРАММИРОВАНИЕ ДЛЯ НАЧИНАЮЩИХ

ПЕРВЫЙ ШАГ НА ПУТИ К УСПЕШНОЙ КАРЬЕРЕ

Для версий 3.1 - 3.4



ПРОГРАММИРОВАНИЕ
ДЛЯ НАЧИНАЮЩИХ

Mike McGrath

# PYTHON IN EASY STEPS

*In Easy Steps Limited*

Майк МакГрат

# ПРОГРАММИРОВАНИЕ НА PYTHON

Москва
2022

УДК 004.43
ББК 32.973.2-018.1
М15



**МакГрат, Майк.**

М15 Программирование на Python для начинающих : [перевод с англ. М.А. Райтмана] / Майк МакГрат. — Москва : Эксмо, 2022. — 192 с. — (Программирование для начинающих).

Книга «Программирование на Python для начинающих» является исчерпывающим руководством для того, чтобы научиться программировать на языке Python.

В этой книге с помощью примеров программ и иллюстраций, показывающих результаты работы кода, разбираются все ключевые аспекты языка. Установив свободно распространяемый интерпретатор Python, вы с первого же дня сможете создавать свои собственные исполняемые программы!

**УДК 004.43**
**ББК 32.973.2-018.1**

Производственно-практическое издание

ПРОГРАММИРОВАНИЕ ДЛЯ НАЧИНАЮЩИХ

**Майк МакГрат**

**ПРОГРАММИРОВАНИЕ НА PYTHON ДЛЯ НАЧИНАЮЩИХ**
(орыс тілінде)

Директор редакции *Е. Капьёв*
Ответственный редактор *В. Обручев*
Художественный редактор *В. Брагина*

В оформлении обложки использована фотография:
Toria / Shutterstock.com
Используется по лицензии от Shutterstock.com

Сведения о подтверждении соответствия издания согласно законодательству РФ
о техническом регулировании можно получить по адресу:
http://eksmo.ru/certification/

Өндірген мемлекет: Ресей. Сертификация қарастырылмаған



Страна происхождения: Российская Федерация

Шығарылған елі: Ресей Федерациясы

**ООО «Издательство «Эксмо»**
123308, Россия, город Москва, улица Зорге, дом 1, строение 1, этаж 20, каб. 2013.
Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru
Өндіруші: «ЭКСМО» АҚБ Баспасы,
123308, Ресей, қала Москеу, Зорге көшесі,1 үй, 1 гимарат, 20 қабат, офис 2013 ж.
Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru.
Тауар белгісі: «Эксмо»
**Интернет-магазин** : www.book24.ru

**Интернет-магазин** : www.book24.kz
**Интернет-дүкен** : www.book24.kz
Импортёр в Республику Казахстан ТОО «РДЦ-Алматы».
Қазақстан Республикасындағы импорттаушы «РДЦ-Алматы» ЖШС.
Дистрибьютор и представитель по приему претензий на продукцию,
в Республике Казахстан: ТОО «РДЦ-Алматы»
Қазақстан Республикасында дистрибьютор және өнім бойынша арыз-талаптарды қабылдаушының өкілі «РДЦ-Алматы» ЖШС,
Алматы қ., Домбровский көш., 3-а», литер Б, офис 1.
Тел.: 8 (727) 251-59-90/91/92. E-mail: RDC-Almaty@eksmo.kz
Өнімнің жарамдылық мерзімі шектелмеген.
Сертификация туралы ақпарат сайтта: www.eksmo.ru/certification

Сведения о подтверждении соответствия издания согласно законодательству РФ
о техническом регулировании можно получить на сайте Издательства «Эксмо»
www.eksmo.ru/certification
Өндірген мемлекет: Ресей. Сертификация қарастырылмаған

Дата изготовления / Подписано в печать 13.04.2022.
Формат 84x108 1/16. Печать офсетная. Усл. печ. л. 20,16.
Доп. тираж 2000 экз. Заказ

12+



ISBN 978-5-699-81406-0



**ISBN 978-5-699-81406-0**

# Оглавление

# Предисловие

Создание этой книги лично для меня стало увлекательным путешествием в мир, раскрывающий возможности языка Python в сегодняшнем процедурном и объектно ориентированном программировании, используемом для обеспечения функциональности при разработке онлайн-приложений. Примеры кода, представленные в этой книге, описывают, как за несколько простых шагов создавать программы на языке Python, а на скриншотах демонстрируются реальные результаты их работы. Я искренне надеюсь, что вам понравится открывать захватывающие возможности Python и вы получите при этом не меньше удовольствия, чем я во время написания этой книги.

Для того чтобы код, описанный в примерах, стал более наглядным, он отформатирован черным шрифтом, за исключением комментариев, выделенных серым шрифтом:

```
# Пишем традиционное приветствие

greeting = 'Hello World!'

print( greeting )
```

Кроме того, для идентификации исходных файлов, описываемых в пошаговых инструкциях, на полях рядом с каждым пунктом будут появляться значок и имя соответствующего файла:


`script.py` `page.html` `image.gif`

Для удобства файлы исходных кодов всех примеров, представленных в этой книге, помещены в один ZIP-архив. Вы можете получить его, выполнив следующие простые шаги.

1. Откройте браузер и загрузите архив по ссылке http://eksmo.ru/Python_examples.zip.
2. Извлеките из скачанного архива папки *MyScripts* и *MyProjects* в ваш домашний каталог (например, в *C:*) а также скопируйте содержимое папки *htdocs* в каталог документов вашего веб-сервера.
3. Теперь вы можете, используя пошаговые инструкции, выполнять примеры с помощью интерпретатора Python и видеть результаты его работы.

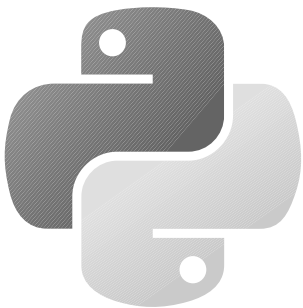
# 1 Приступаем к работе

*Добро пожаловать в увлекательный мир языка программирования Python. В этой главе показывается, как установить Python и создать вашу первую программу.*

- **Введение в язык Python**
- **Установка Python в среде Windows**
- **Установка Python в среде Linux**
- **Знакомство с интерпретатором**
- **Ваша первая программа**
- **Работа с переменными**
- **Получение введенных пользователем данных**
- **Исправление ошибок**
- **Заключение**

# Введение в язык Python

Python является высокоуровневым («человекочитаемым») языком программирования, который для вывода результатов использует интерпретатор. Python содержит обширную стандартную библиотеку модулей протестированного кода, которые легко могут быть включены в ваши собственные программы.



Будьте в курсе последних новостей проекта Python на сайте**python.org**.

Язык Python, разработанный Гвидо ван Россумом (Guido van Rossum) в конце восьмидесятых — начале девяностых годов в Национальном научно-исследовательском институте математики и компьютерных наук в Нидерландах, является производным от многих других языков, в том числе C, C ++ и командной оболочки Unix. Сегодня Python поддерживается командой разработчиков ядра в институте, хотя Гвидо ван Россум по-прежнему играет важную роль в определении направления развития языка.

Читаемость кода, делающая язык Python особенно подходящим для новичков в программировании, — один из принципов философии Python, которую можно обобщить следующим образом.

- Красивое лучше, чем уродливое.
- Явное лучше, чем неявное.
- Простое лучше, чем сложное.
- Сложное лучше, чем запутанное.
- Читаемость имеет значение.

**Совет**

Так называемое правило офсайда, которое используют некоторые языки программирования, выделяя блоки кода при помощи отступов, заимствовано из футбола.

Поскольку Python ориентирован на читаемость кода, в нем часто используются ключевые слова на английском языке там, где другие языки программирования обычно используют знаки препинания. Особое его отличие состоит в том, что для группировки инструкций в блоке кода Python использует отступы, а не ключевые слова или знаки препинания. В языке Pascal, например, начало блоков обозначается ключевым словом `begin` и заканчивается ключевым словом `end`, в то время как программисты на C используют фигурные скобки для обозначения блоков кода. Очень часто такой подход группировки блоков отступами критикуется программистами, знакомыми с другими языками, но, несомненно, использование отступов в Python позволяет программам выглядеть менее нагроможденными.

Перечислим некоторые из важнейших отличительных особенностей языка Python, которые делают его привлекательным для начинающих программистов.

- **Python бесплатен** — это свободно распространяемое программное обеспечение с открытым исходным кодом.
- **Python легок в изучении** — он имеет простой синтаксис.
- **Python позволяет создавать легко читаемый код** — он не перегружен знаками препинания.
- **Python легок в обслуживании** — имеет модульную структуру.
- **Python располагает богатым «арсеналом»** — он предлагает большую стандартную библиотеку, которая легко интегрируется в ваши программы.
- **Python портируемый** — его можно запустить на обширном множестве различных платформ, и везде он будет иметь один и тот же интерфейс.
- **Python интерпретируемый** — компиляция не требуется.
- **Python является высокоуровневым языком** — он имеет статическое распределение памяти.
- **Python расширяемый** — позволяет добавлять низкоуровневые модули.
- **Python универсален** — поддерживает как процедурный, так и объектно ориентированный методы программирования.
- **Python гибок в использовании** — с его помощью можно создавать консольные программы, приложения графического интерфейса, а также сценарии для взаимодействия внешних программ с вебсерверами.

Как и любое другое программное обеспечение, Python продолжает развиваться, его новые версии выпускаются с определенной периодичностью. Объявлено, что версия 2.7 будет окончательной в ветке 2.x. Но ее поддержка будет продлена до 2020 года. Других больших релизов в данной ветке не ожидается.

Ветка версии 3.x находится в активной разработке и уже имеет несколько стабильных релизов. Это значит, что все последние улучшения стандартных библиотек, например, окажутся доступными только в версии Python 3.x. Описанные в нашей книге особенности языка будут относиться к версии 3.x.

**На заметку**

Название языку Python было дано в честь популярного британского комедийного шоу «Летающий цирк Монти Пайтона» (Monty Python's Flying Circus) — вы можете найти упоминание об этом в документации по языку.

**Внимание**

Python 3.x обратно несовместим с версией Python 2.7.

# Установка Python в операционной системе Windows

Перед тем как вы начнете программировать на языке Python, необходимо установить на ваш компьютер интерпретатор Python, а также стандартную библиотеку модулей кода, поставляемую вместе с ним. Все это можно свободно загрузить на странице **python.org/download**. Для пользователей операционной системы Windows существуют две версии инсталлятора: для 32-битных и 64-битных систем.

1. Запустите веб-браузер, перейдите на страницу **python.org/download** и загрузите установщик, подходящий для вашей версии операционной системы — в нашем примере файл имеет имя *python-3.3.2.msi*.
2. После завершения загрузки запустите установщик, выберите режим установки для всех пользователей либо только для себя и нажмите кнопку **Next** (Далее) для продолжения.
3. Теперь подтвердите предлагаемое расположение установки, в название которого будет входить имя корневого диска, слово Python и номер версии — в данном примере установка произойдет в каталог *C:\Python33* для версии 3.3.2.


Установщики для операционной системы OS X 32-битной и 64-битной версий также доступны для загрузки на **python.org/download**.

**Совет**

Поддержка установщика MSI включена для всех версий Windows и доступна для свободной загрузки на **microsoft.com/downloads** — введите в строке поиска Windows Installer.

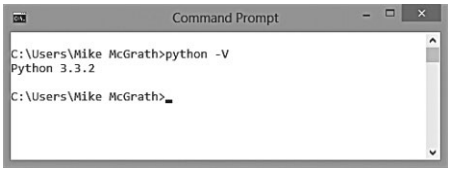
4. Нажмите кнопку **Next** (Далее) для продолжения и убедитесь, что выбран компонент **Add python.exe to Path** (Добавить путь в системную переменную Path).


**На заметку**

Убедитесь, что все компоненты установки включены, как показано в нашем примере.

5. Нажмите **Next** (Далее), чтобы начать копирование файлов на ваш компьютер, а затем — **Finish** (Готово) для завершения процесса установки.

6. Чтобы убедиться, что Python теперь доступен, перезагрузите компьютер, запустите командную строку (*cmd.exe*) и наберите команду `python -V` — в ответ интерпретатор Python выдаст номер установленной версии.

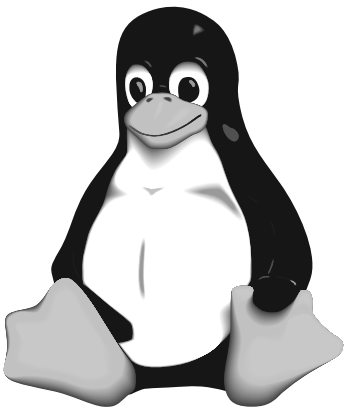



**Внимание**

Буква `V` в команде должна быть указана обязательно прописной. Перед тем как продолжать работать с примерами в книге, убедитесь, что данная команда выдает необходимый номер установленной версии.

# Установка Python в операционной системе Linux

В дистрибутивы Linux обычно включен Python — по умолчанию там используется версия 2.7. Для работы с веткой 3.x вам, очевидно, предстоит установить нужный релиз дополнительно.



Обратитесь к документации по вашей операционной системе Linux для дальнейшей установки Python.

1. Запустите терминальное окно и наберите в точности, как указано, команду `python -V` для вывода информации об установленной версии по умолчанию.

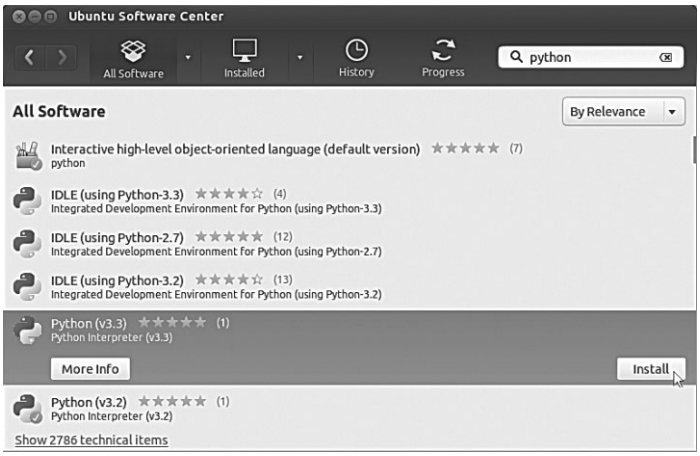

2. Затем наберите в точности команду `python3 -V` для того, чтобы увидеть информацию об установленной версии ветки 3.x, если таковая имеется.


**Внимание**

Не удаляйте версию 2.7 из вашей системы, так как во многих случаях существуют зависимые от нее приложения, работоспособность которых может нарушиться.

3. Теперь запустите на вашей Linux системе менеджер пакетов, чтобы посмотреть, какая из последних версий Python доступна для установки — например, на системах с Ubuntu вы можете использовать Центр приложений (Ubuntu Software Center).

4. Найдите в менеджере пакетов необходимое вам программное обеспечение, название которого содержит слово Python, чтобы посмотреть информацию, какие компоненты установлены или доступны для установки.




**Совет**

Вы можете по желанию установить среду разработки IDLE для Python3.3, но это совсем не обязательно, так как все примеры в книге созданы при помощи обычного текстового редактора, такого, как Nano.

5. Наконец установите последнюю версию из ветки Python3.x — в данном случае это Python3.3.

6. Для проверки доступности последней версии Python на вашем компьютере запустите терминальное окно и наберите команду `python3.3 -V`.

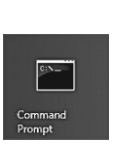

**На заметку**

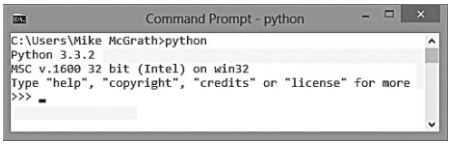
Теперь вы можете использовать команду `python3.3` для того, чтобы ваши программы отрабатывались интерпретатором именно этой версии.

# Знакомство с интерпретатором

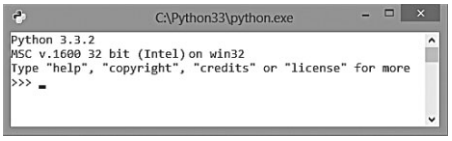
Интерпретатор Python обрабатывает текстовый код вашей программы, а также имеет интерактивный режим, полезный для отладки и тестирования фрагментов кода. В интерактивный режим Python можно попасть несколькими способами:

- из обычной командной строки — введите команду `python`, чтобы запустить начальную командную строку Python (символы >>>), в которой вы будете взаимодействовать с интерпретатором;

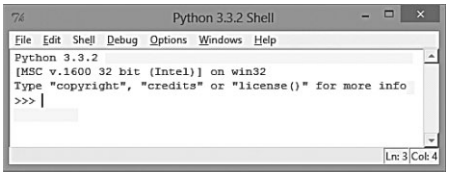



- из меню **Пуск** (Start) — выберите пункт **Python (command line)** — запустится окно, содержащее начальную командную строку интерпретатора Python с символами >>>;

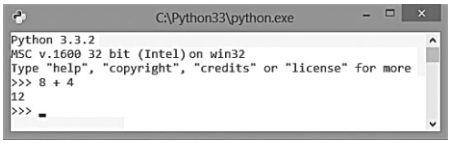




- из меню **Пуск** (Start) — выберите пункт **IDLE (Python GUI)**, чтобы запустить окно оболочки Python, содержащее командную строку с символами >>>.

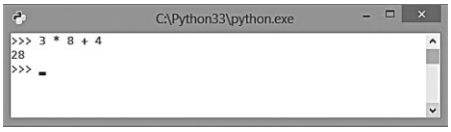
Независимо от того, какой способ вы выбрали для входа в интерактивный режим, интерпретатор Python будет одинаково реагировать на команды, вводимые в его командной строке после знаков >>>. В этом режиме его можно использовать в качестве калькулятора.

1. Войдите в интерактивный режим Python, используя любой из вышеперечисленных методов, затем наберите простое выражение с операцией сложения и нажмите кнопку **Enter**. Интерпретатор в ответ выдаст вам сумму.

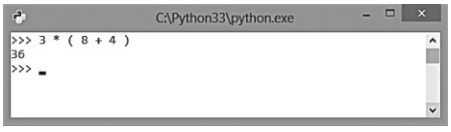



Интерпретатор Python понимает любые арифметические выражения, поэтому можно использовать скобки для указания порядка вычисления — часть выражения, заключенная в скобки, будет вычисляться первой.

2. Затем в командной строке Python наберите выражение с тремя операндами без указания порядка вычисления.




3. Теперь в командной строке Python наберите то же самое выражение, но добавьте скобки, определяющие порядок вычисления.




**Совет**

Пробелы в выражениях игнорируются, поэтому выражение 8+4, как показано здесь, можно записать с добавлением пробелов просто для красоты восприятия.

**На заметку**

Интерактивный режим используется в основном для тестирования и отладки фрагментов кода.

**Внимание**

IDLE расшифровывается как Integrated DeveLopment Environment — интегрированная среда разработки. Она имеет ограниченные функции и в данной книге не используется для демонстрации примеров.

# Ваша первая программа

Кроме того, что интерактивный режим Python полезен в качестве простейшего калькулятора, его можно использовать для создания программ. Программа на языке Python — это обычный текстовый файл, созданный с помощью простого редактора, такого как Блокнот (Notepad), и сохраненный в файле с расширением *.py*. Запустить программу на Python можно, указав имя соответствующего файла после команды `python` в командной строке интерпретатора.

**Внимание**

Не используйте текстовые процессоры для создания исходного кода программ, поскольку они добавляют дополнительное форматирование.

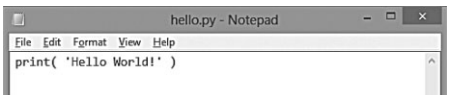
По традиции первая программа, которую создают при изучении языка программирования, просто выводит какое-либо сообщение с приветствием. На языке Python для этого используется функция `print()`, сообщение для вывода этой функции указывается в скобках. Это может быть строка символов, заключенная в кавычки. Кавычки могут быть как двойными (`""`), так и одинарными(`''`), но нельзя использовать одновременно и те, и другие.


`hello.py`

1. На компьютере под управлением операционной системы Windows запустите простой текстовый редактор, такой как, например, Блокнот (Notepad).
2. Затем наберите следующую инструкцию в пустой строке редактора:

```
print( 'Hello World!' )
```

3. Теперь создайте новый каталог *C:\MyScripts* и сохраните в нем файл под именем *hello.py*.




**Совет**

Созданный каталог *C:\MyScripts* будет использоваться во всех примерах этой книги для Windows.

4. Теперь запустите окно командной строки, перейдите в только что созданный каталог и наберите команду `python hello.py` — вы увидите, как интерпретатор Python запустит вашу программу и выведет приветственное сообщение.

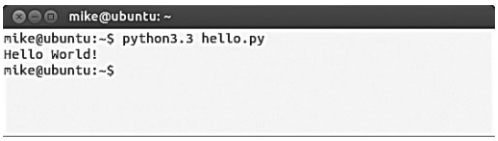
Процедура создания первой программы на Python в системе Linux ничем не отличается от той, которая делается в Windows. Однако, независимо от используемой платформы, всегда нужно помнить, что если установлены различные версии Python, то следует использовать корректную команду для вызова интерпретатора. Особенно это важно для системы Linux, которую обычно поставляют вместе с установленной версией Python 2.7, и набранная команда `python` по умолчанию вызывает именно этот интерпретатор. Если установлен Python 3.3 и вы хотите вызывать именно его для обработки вашей программы, то следует использовать команду `python3.3`, чтобы вызвать нужную версию интерпретатора.

1. В системе Linux запустите любой текстовый редактор, например Nano.
2. Затем наберите следующую инструкцию в пустом окне редактора:

```
print( 'Hello World!' )
```

3. Теперь сохраните файл в вашем домашнем каталоге под именем *hello.py*.


4. Наконец запустите терминальное окно и перейдите в ваш домашний каталог, а затем наберите команду `python3.3 hello.py` — вы увидите, как интерпретатор Python запустит вашу программу и выведет соответствующее сообщение.





hello.py



**На заметку**

Все дальнейшие примеры, описанные в этой книге, касаются операционной системы Windows (наиболее популярной среди пользователей). Но примеры также можно создать и выполнить на Linux.

# Работа с переменными

В программировании переменная представляет собой некоторый контейнер в памяти компьютера, где хранятся данные. После того как данные сохранены, их можно вызвать, используя имя этой переменной. Программист может выбрать любое имя для переменной, за исключением ключевых слов языка Python. Лучше выбирать для переменных значащие имена, которые отражают их содержание.

**На заметку**

Строковые данные должны быть заключены в кавычки, обозначающие начало и конец строки.

В программах Python данные, которые нужно хранить в переменных, вносятся с помощью оператора присваивания `=`, например, чтобы сохранить числовое значение `8` в переменной с именем `a`, нужно написать:

```
a = 8
```

Затем можно обратиться к сохраненному значению переменной, используя ее имя. Таким образом, инструкция `print( a )` выведет сохраненное значение `8`. Переменным могут быть последовательно присвоены разные значения, и, следовательно, переменная способна принимать различные значения по мере работы программы — неслучайно она так и называется: переменная.

В языке Python переменной должно быть присвоено начальное значение (инициализация переменной) в инструкции, которая объявляет эту переменную в программе, — иначе интерпретатор вызовет сообщение об ошибке not defined (неопределенная переменная).

В одной инструкции разрешается инициализировать несколько переменных с одним значением. Это можно сделать, используя оператор присваивания `=`. Например, для инициализации переменных `a`, `b` и `c` и присваивания им значения `8` мы используем запись:

```
a = b = c = 8
```

**Совет**

Языки программирования, которые требуют определения типов переменных, известны как строго типизированные в отличие от нестрого типизированных.

Наоборот, несколько переменных можно проинициализировать с различными значениями и записать все это в одной инструкции, используя запятую в качестве разделителя. Например, в качестве инициализации переменных `a`, `b` и `c` с числовыми значениями `1`, `2`, `3` мы используем запись:

```
a, b, c = 1, 2, 3
```

Некоторые языки программирования, такие как Java, требуют указания типов переменных при их объявлении. При этом резервируется определенный объем памяти. Данный прием известен как статическая типизация. На переменные в языке Python такое ограничение не накладывается, и распределение памяти происходит в соответствии с при-

сваиваемыми переменным значениями (динамическая типизация). Это означает, что переменная может содержать как целые числа, так и числа с плавающей точкой, текстовые строки или логические значения.

Вы можете добавлять в свои программы на Python комментарии для описания инструкций или разделов кода. Для этого используется символ #. Все, что идет после этого символа до конца строки, игнорируется интерпретатором Python. Комментарии очень полезны — они помогают сделать ваш код понятным для других, а также для вас самих, когда вы позже к нему возвращаетесь.

1. Запустите текстовый редактор, в котором объявите и инициализируйте переменную, затем выведите хранящееся в ней значение.


var.py

```
# Инициализируем переменную целочисленным значением

var = 8

print( var )
```

2. Затем присвойте новое значение переменной и выведите его на экран.

```
# Присваиваем переменной значение числа с плавающей точкой

var = 3.142

print( var )
```

3. Теперь присваиваем другое значение и отображаем его опять.

```
# Присваиваем переменной строковое значение

var = 'Python in easy steps'

print( var )
```

4. Наконец присваиваем еще одно значение и снова выводим результат.

```
# Присваиваем переменной логическое значение

var = True

print( var )
```

5. Сохраните файл в вашем рабочем каталоге, затем откройте командную строку из этого каталога и запустите программу, чтобы посмотреть результат ее вывода.


**Совет**

Вы можете использовать в своих программах многострочные комментарии, которые заключаются в тройные кавычки, """…""".

# Получение введенных пользователем данных

Значения переменным в языке Python можно присваивать не только с помощью программы, но и путем пользовательского ввода. Для этого используется функция `input()`. Она в качестве аргумента принимает строку, которая будет отображаться пользователю, приглашая его ввести данные, а затем читает строку, введенную пользователем.

Такие символы интерпретируются как текстовая строка, даже если на самом деле это числовые значения. Данная строка может быть присвоена любой переменной, используя оператор присваивания `=`. Впоследствии с этой переменной можно работать точно так же, как и с другими, например вывести ее значение, указав имя переменной в функции `print()`.

При помощи функции `print()` можно вывести и несколько значений переменных, указав их внутри скобок через запятую.


`input.py`

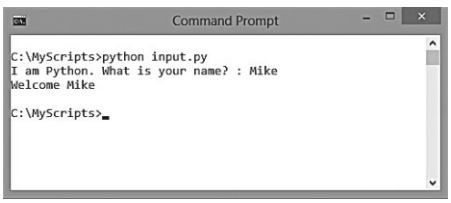
1. Запустите текстовый редактор, в котором объявите и проинициализируйте переменную с помощью запроса пользовательского ввода.

```
# Инициализируем переменную значением, введенным пользователем

user = input( 'I am Python. What is your name? : ' )
```

2. Затем отобразите ответное сообщение, подтверждая введенное пользователем значение.

```
# Выводим строку и значение переменной

print( 'Welcome' , user )
```

3. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу — введите ваше имя, нажмите клавишу **Enter**, и вы увидите ответное сообщение, содержащее ваше имя.




**Совет**

Обратите внимание, что строка подсказки заканчивается пробелом, который отображается при выводе. Это простой способ немного отделить строку пользовательского ввода.

Когда вы выводите с помощью функции **print()** несколько значений, они по умолчанию отделяются единичным пробелом. Чтобы указать альтернативный разделитель, вы можете добавить параметр **sep** для функции **print()**. Например, при использовании **sep = '*'** вы получите в выводе величи́ны, разделенные символом *.

Также по умолчанию функция **print()** выводит в конце каждой строки неотображаемый символ новой строки (**\n**). Но существует возможность указать собственный символ, используя параметр **end**. Например, использование **end = '!'** выведет в конце каждой строки знак восклицания.

4. Отредактируйте вашу программу, объявив и проинициализировав вторую переменную с помощью еще одного запроса пользовательского ввода.

```
# Инициализируем еще одну переменную значением, введенным пользователем

lang = input( 'Favorite programming language? : ' )
```

5. Выведите ответное сообщение для подтверждения ввода пользователя, указав свой разделитель вывода и символ окончания строки.

```
# Выводим строку и значение переменной

print( lang , 'Is' , 'Fun' , sep = ' * ' , end = '!\n' )
```

6. Теперь сохраните файл, откройте командную строку и запустите программу заново — введите ваше имя и название языка программирования. Затем нажмите клавишу **Enter**, и вы увидите ответное сообщение, содержащее ваш ввод.

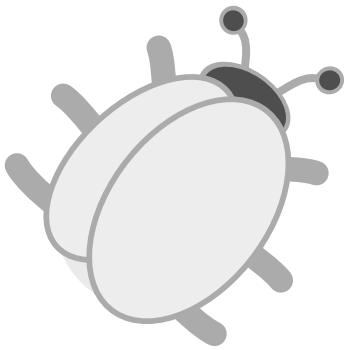

**На заметку**

Вы можете явно задать конец строки, добавив его в параметр **end**. Например, **end='!\n'** добавляет как восклицательный знак, так и символ новой строки.

**Совет**

Для большей наглядности и читаемости вы можете добавлять к разделителям пробелы, как показано в этом примере.

# Исправление ошибок



Ошибки в программах часто называются багами (от англ. bug — жук), а процесс их отслеживания называется отладкой (англ. debugging, одно из значений — удаление насекомых с растений).

При выполнении написанных на языке Python программ могут происходить ошибки. Существуют три основных типа ошибок, которые следует различать, чтобы легче их потом было исправлять.

- **Синтаксическая ошибка** — происходит, когда интерпретатор обрабатывает код, который не соответствует правилам языка Python, например отсутствие кавычек вокруг строковой переменной. Интерпретатор останавливается и сообщает об ошибке без дальнейшего выполнения программы.
- **Ошибка исполнения —** происходит во время исполнения программы. Например, когда переменная не может быть распознана из-за несоответствия типов. Интерпретатор запускает программу, останавливается на ошибке и сообщает о природе этой ошибки как об исключении.
- **Логическая ошибка (смысловая)** — происходит, когда программа ведет себя не так, как было задумано. Например, когда не определен порядок действий в вычисляемом выражении. Интерпретатор запускает программу и не сообщает об ошибке.

С исправлением синтаксических и ошибок выполнения все достаточно очевидно, поскольку интерпретатор сообщает, где ошибка произошла, и указывает на природу ее происхождения. Но для логических ошибок требуется детальное изучение кода.


syntax.py

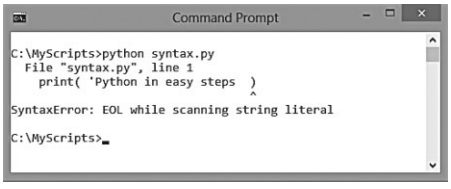
1. Запустите текстовый редактор и добавьте выражение для вывода строки, при этом опустив знак закрывающих кавычек.

```
print( 'Python in easy steps )
```

2. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу — вы увидите, что интерпретатор сообщает о синтаксической ошибке и указывает ее позицию в коде.

**Внимание**

Обычно указатель на синтаксическую ошибку располагается не на месте ошибки, а на следующем символе.

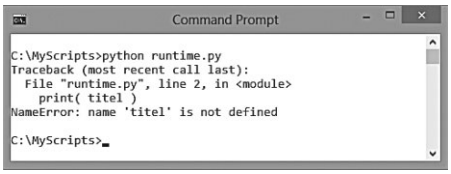


3. Добавьте символ кавычки перед закрывающейся скобкой, чтобы закончить строку, сохраните файл и запустите программу заново. Вы увидите, что ошибка исправлена.

4. Теперь начните новую программу, в которой проинициализируйте переменную, а затем попытайтесь вывести ее значение, указав неверное имя для этой переменной — вы увидите, как интерпретатор сообщит об ошибке выполнения.


runtime.py

```
title = 'Python in easy steps'

print( titel )
```




**Совет**

Подробно о том, как обрабатывать исключения, описано в главе 4.

5. Исправляем имя переменной в соответствии с ее описанием, сохраняем файл, запускаем программу заново — и видим, что ошибка исправлена.

6. Теперь начните новую программу, проинициализируйте переменную, затем попытайтесь вывести значение выражения без явного указания порядка действий. Вы увидите неожиданный результат.


semantic.py

```
num = 3

print( num * 8 + 4 )
```


7. Добавьте скобку, чтобы выражение приобрело вид `3 * ( 8 + 4 )`. Сохраните файл и запустите программу заново. Вы увидите ожидаемое значение `36` как результат корректировки смысловой ошибки.

# Заключение

- Python — высокоуровневый язык программирования, использующий для вывода результатов интерпретатор Python.
- Python использует отступы для группировки инструкций в блоки кода, в то время как в других языках для этого применяются ключевые слова либо знаки препинания.
- Python 2.7 является последней версией ветки 2.x, в то время как для ветки 3.x. доступны новые обновления.
- Пользователи операционной системы Windows могут устанавливать Python с помощью установщика MSI, а пользователи системы Linux — используя менеджер пакетов.
- Интерпретатор Python имеет интерактивный режим, в котором вы можете проверять куски кода и который очень полезен для отладки программы.
- Программа на языке Python представляет собой текстовый файл, который можно создать в простейшем текстовом редакторе и сохранить с расширением *.py*.
- Функция `print()` выводит на экран строку, указанную внутри ее скобок.
- Строковые значения должны быть заключены в кавычки.
- Если на одной системе установлено несколько версий Python, то очень важно указать желаемый интерпретатор.
- Переменная в языке Python представляет собой именованный контейнер, в котором хранится значение. Доступ к этому значению может быть осуществлен по имени переменной.
- Переменная в языке Python может содержать данные любого типа, но при объявлении ей должно быть задано начальное значение.
- Функция `input()` выводит строку, указанную внутри ее скобок, а затем ожидает строку ввода.
- Синтаксические ошибки, вызванные некорректным кодом, распознаются интерпретатором до исполнения программы.
- Ошибки исполнения, вызванные исключениями, распознаются интерпретатором во время исполнения программы.
- Смысловые ошибки, вызванные неожиданными результатами, не распознаются интерпретатором.

# 2 Выполнение операций

*Эта глава знакомит нас с операторами языка Python и демонстрирует их работу.*

- **Арифметические действия**
- **Присваивание значений**
- **Сравнение величин**
- **Оценочная логика**
- **Проверка условий**
- **Определение приоритетов**
- **Преобразование типов данных**
- **Манипуляции с битами**
- **Заключение**

# Арифметические действия

Основные операторы, используемые при программировании на языке Python, а также выполняемые ими операции, представлены в таблице ниже.

| Оператор | Операция |
|---|---|
| + | Сложение |
| - | Вычитание |
| * | Умножение |
| / | Деление |
| % | Деление по модулю |
| // | Целочисленное деление |
| ** | Возведение в степень |

Операторы сложения, вычитания, умножения и деления не должны вызывать каких-либо сложностей. Они работают так, как вы, наверное, и ожидали. Стоит, однако, обращать внимание, что при использовании нескольких операторов рекомендуется группировать выражения с помощью скобок — операции внутри скобок выполняются первыми. Например, выражение

```
a = b * c - d % e / f
```

может быть не совсем понятно с точки зрения порядка вычислений. Но его разрешается уточнить, записав в виде:

```
a = ( b * c ) - ( ( d % e ) / f )
```

Оператор % (деление по модулю) делит одно число на другое и возвращает остаток от деления. Он очень полезен для определения четности или нечетности числа.

Оператор // (целочисленное деление) работает так же, как и обычное деление, /, но отбрасывает результат после десятичной точки.

Оператор ** (возведение в степень) возводит первый операнд в степень второго операнда.



**Совет**

Значения, используемые вместе с операторами для формирования выражений, называются операндами — например, операндами в выражении 2 + 3 являются числа 2 и 3.

1. Начните новую программу с инициализации двух переменных с целочисленными значениями.

```
a = 8

b = 2
```

2. Затем выведите результат сложения переменных.

```
print( 'Addition:\t' , a , '+' , b , '=' , a + b )
```

3. Теперь отобразите результат вычитания переменных.

```
print( 'Subtraction:\t' , a , '-' , b , '=' , a - b )
```

4. Затем отобразите результат умножения переменных.

```
print( 'Multiplication:\t' , a , 'x' , b , '=' , a * b )
```

5. Отобразите результат деления переменных как с плавающей точкой, так и целочисленного.

```
print( 'Division:\t' , a , '÷' , b , '=' , a / b )

print( 'Floor Division:\t' , a , '÷' , b , '=' , a // b )
```

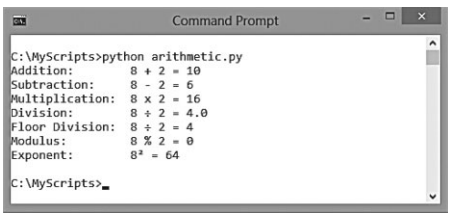
6. Затем выведите остаток от деления одной величины на другую.

```
print( 'Modulus:\t' , a , '%' , b , '=' , a % b )
```

7. Наконец отобразите результат возведения первого операнда в степень второго.

```
print( 'Exponent:\t ' , a , '² = ' , a ** b , sep = '' )
```

8. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу — вы увидите результат арифметических операций.





arithmetic.py

**Совет**

\t — это так называемая управляющая последовательность, которая добавляет невидимый символ табуляции для форматирования вывода.

**На заметку**

Вы можете использовать параметр sep для явного указания разделения между выводами. В этом примере нет разделения, так как указано два символа апострофа без пробела.

# Присваивание значений

Операторы, используемые при программировании на языке Python для присваивания значений переменным, перечислены в таблице ниже. Все они, за исключением простого присваивания, `=`, являются сокращенными формами от более длинных выражений. Для ясности в таблице представлены их эквиваленты.

| Оператор | Пример | Эквивалентная операция |
|---|---|---|
| `=` | `a = b` | `a = b` |
| `+=` | `a += b` | `a = ( a + b )` |
| `-=` | `a -= b` | `a = ( a – b )` |
| `*=` | `a *= b` | `a = ( a * b )` |
| `/=` | `a /= b` | `a = ( a / b )` |
| `%=` | `a %= b` | `a = ( a % b )` |
| `//=` | `a //= b` | `a = ( a // b )` |
| `**=` | `a **= b` | `a = ( a ** b )` |

В таблице выше переменной с именем `a` присваивается значение, которое содержится в переменной с именем `b`, и, таким образом, в переменной `a` будет храниться новое значение.

Оператор `+=` полезно использовать для добавления какого-то значения к существующему, хранящемуся в переменной `a`.

Оператор `+=` сначала добавляет к значению, содержащемуся в переменной, `b` значение, содержащееся в переменной `a`, а потом результат присваивает значению переменной `a`.

Все другие операторы работают по тому же принципу — сначала выполняют арифметическую операцию между двумя операндами, а затем присваивают первой переменной значение этого результата.

При использовании оператора `%=` первый операнд `a` делится на второй операнд `b`, и затем остаток от этого деления присваивается переменной `a`.



**На заметку**

Важно различать операции присваивания, `=`, и равенства, `==`, чтобы избежать недоразумений с логическим оператором равенства `==`.

1. Начните новую программу на Python, которая проинициализирует две переменные с помощью присваиваний им целых значений, и выведите эти присвоенные значения.


assign.py

```
a = 8

b = 4

print( 'Assign Values:\t\t' , 'a =' , a , '\tb =' , b )
```

2. Теперь присвойте новое значение первой переменной и выведите его на экран.

```
a += b

print( 'Add & Assign:\t\t' ,'a =' , a , '(8 += 4)' )
```

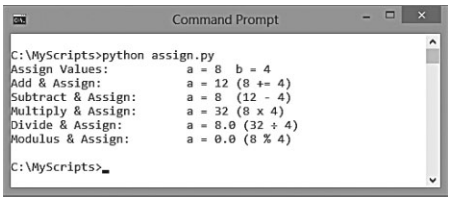
3. Проделайте то же самое, что и в предыдущем шаге, но теперь с использованием вычитания и умножения.

```
a -= b

print( 'Subtract & Assign:\t' , 'a =' , a , ' (12 - 4)' )

a *= b

print( 'Multiply & Assign:\t' , 'a =' , a , '(8 x 4)' )
```

4. Наконец с помощью деления и присваивания получите новое значение для первой переменной и выведите на экран результат, после чего используйте деление по модулю и присваивание с выводом результата.

```
a /= b

print( 'Divide & Assign:\t' , 'a =' , a , '(32 ÷ 4)' )

a %= b

print( 'Modulus & Assign:\t' , 'a =' , a , '(8 % 4)' )
```

5. Сохраните файл в вашем рабочем каталоге, затем откройте командную строку и запустите программу — вы увидите результат работы операции присваивания.




**Внимание**

В отличие от оператора присваивания, **=**, оператор равенства, **==**, сравнивает операнды. Он описывается в следующем разделе.

# Сравнение величин

Операторы, которые обычно используются при программировании на языке Python для сравнения двух операндов, представлены в таблице ниже.

| Оператор | Проверяемое условие |
|---|---|
| = = | Равенство |
| ! = | Неравенство |
| > | Больше |
| < | Меньше |
| > = | Больше либо равно |
| < = | Меньше либо равно |

Оператор равенства, **==**, производит сравнение двух операндов и возвращает **True** (Истина), если их значения равны, в противном случае возвращает **False** (Ложь). При этом если операндами являются числовые значения, и они одинаковые, то они равны, а если символы, то сравниваются их ASCII-коды.

Оператор неравенства, **!=**, наоборот, возвращает значение **True**, если оба операнда не равны, используя то же самое правило, как и оператор равенства, в противном случае возвращая **False**. Операторы равенства и неравенства полезны для выполнения условного ветвления в программе по результатам сравнения значений двух переменных.

Оператор «больше», **>**, сравнивает два операнда и возвращает **True**, если значение первого больше, чем значение второго, и наоборот, возвращает **False**, если значение первого равно значению второго или меньше его. Оператор «меньше», **<**, делает то же самое, но возвращает **True**, если первый оператор меньше. Эти два оператора часто используются для проверки значения счетчика операций в цикле.

Добавление оператора «равно», **=**, после оператора «больше», **>**, или «меньше», **<**, заставляет их возвращать значение **True**, если значения операндов совпадают.



**Совет**

Символы верхнего регистра от A до Z имеют коды ASCII 65–90, а символы нижнего регистра a–z — от 97 до 122.

1. Начните новую программу в Python с инициализации пяти переменных, которые будут использоваться для сравнения.

comparison.py

```
nil = 0

num = 0

max = 1

cap = 'A'

low = 'a'
```

2. Теперь добавьте инструкции для вывода результатов числового и символьного сравнения с помощью оператора равенства.

```
print( 'Equality :\t' , nil , '==' , num , nil == num )

print( 'Equality :\t' , cap , '==' , low , cap == low )
```

3. Теперь добавьте инструкцию для вывода результата сравнения с помощью оператора неравенства.

```
print( 'Inequality :\t' , nil , '!=' , max , nil != max )
```

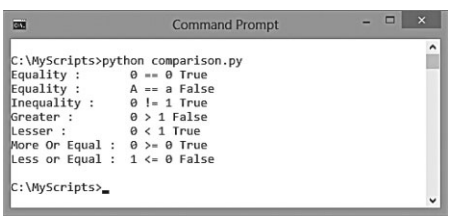
4. Теперь добавим инструкции, чтобы вывести результаты сравнения, выполненного операторами «больше» и «меньше».

```
print( 'Greater :\t' , nil , '>' , max , nil > max )

print( 'Lesser :\t' , nil , '<' , max , nil < max )
```

5. Наконец добавьте инструкцию для вывода результатов работы операторов сравнения, «больше либо равно», «меньше либо равно».

```
print( 'More Or Equal :\t' , nil , '>=' , num , nil >= num )

print( 'Less or Equal :\t' , max , '<=' , num , max <= num )
```

6. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу — вы увидите результаты работы операторов сравнения.




**На заметку**

Управляющая последовательность \t, представленная здесь, добавляет невидимый символ табуляции для форматирования вывода.

**На заметку**

Коды ASCII для символов `A` (65) и `a` (97) не равны — так что сравнение этих значений возвращает значение `False`.

# Оценочная логика

Логические операторы, используемые в программировании на Python, представлены в таблице ниже.

| Оператор | Операция |
|---|---|
| `and` | Логическое «И» |
| `or` | Логическое «ИЛИ» |
| `not` | Логическое «НЕ» |

Логические операторы работают с операндами, которые имеют значение логического (булева) типа, то есть `True` или `False`, либо со значениями, которые преобразуются в `True` или `False`.

Оператор «логическое И», `and`, оценивает два операнда и возвращает значение `True`, только если оба операнда сами имеют значение `True`, в противном случае возвращает значение `False`. Данный оператор обычно используется при условном ветвлении, когда направление работы программы определяется проверкой двух условий — если оба они верны, программа идет в определенном направлении, в противном случае — в другом.

В отличие от оператора `and`, которому необходимо, чтобы оба операнда имели значение `True`, оператор «логическое ИЛИ», `or`, оценивает два операнда и возвращает `True`, если хотя бы один из них сам возвращает значение `True`. В противном случае оператор `or` возвратит значение `False`. Это полезно использовать при программировании определенных действий в случае выполнения одного из двух проверяемых условий.

Оператор «логическое НЕ», `not`, является унарным и используется с одним операндом. Он возвращает противоположное значение от того, какое имел операнд. Так, если переменная `a` имела значение `True`, то `not a` возвратит значение `False`. Он может использоваться, например, для переключения значения переменной в последовательных итерациях цикла при помощи инструкции `a = not a`. Это значит, что на каждой итерации цикла логическое значение меняется на противоположное, подобно выключению и включению лампочки.

**На заметку**

Термин «булев» назван в честь английского математика Джорджа Буля, одного из основателей математической логики.

1. Начните новую программу в Python, проинициализировав две переменные с булевыми значениями для последующей их оценки.

```
a = True

b = False
```


logic.py

2. Теперь добавьте инструкции для того, чтобы вывести результаты работы оператора «логическое И».

```
print( 'AND Logic:' )

print( 'a and a =' , a and a )

print( 'a and b =' , a and b )

print( 'b and b =' , b and b )
```

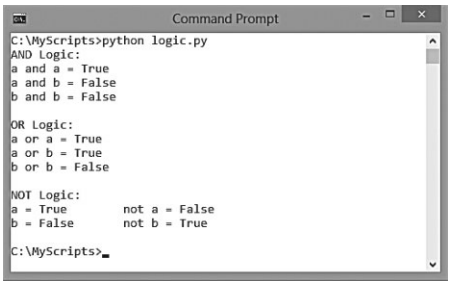
3. Теперь добавьте инструкции, чтобы отобразить вывод результатов работы оператора «логического ИЛИ».

```
print( '\nOR Logic:' )

print( 'a or a =' , a or a )

print( 'a or b =' , a or b )

print( 'b or b =' , b or b )
```

4. Наконец добавляем инструкции для вывода результатов работы «логического НЕ».

```
print( '\nNOT Logic:' )

print( 'a =' , a , '\tnot a =' , not a )

print( 'b =' , b , '\tnot b =' , not b )
```

5. Сохраните файл в рабочем каталоге, откройте командную строку и запустите программу. Вы увидите результаты логических операций.




**Совет**

При программировании на Python логические значения могут быть представлены числами: `True` — это `1`, `False` — это `0`.



**На заметку**

Обратите внимание, что выражение `False and False` возвращает `False`, а не `True`, наглядно демонстрируя непреложную истину «две лжи не станут правдой».

# Проверка условий

Во многих языках программирования, таких как, например, C++ или Java, существует так называемый **тернарный** оператор `?:`, который оценивает выражение на условие истинности, возвращая, в зависимости от результата оценки, одно из двух определенных значений. Тернарный оператор `?:` имеет следующий синтаксис:

```
( проверочное выражение ) ? если-истина-возвращаем-это : если-ложь-возвращаем-это
```

В языке Python роль тернарного оператора выполняет **условное выражение**, которое работает аналогичным способом и использует в своем синтаксисе ключевые слова `if` и `else`:

```
если-истина-возвращаем-это if ( проверочное-выражение ) else если-ложь-возвращаем-это
```

Несмотря на то, что синтаксис условного выражения может поначалу показаться непонятным, с ним стоит познакомиться, поскольку он является достаточно мощным инструментом для организации ветвлений в программах, используя минимальное количество кода. Например, чтобы организовать ветвление при условии, если значение переменной не равно единице, мы пишем следующее:

```
если-истина-выполняем-это if ( var != 1 ) else если-ложь-выполняем-это
```

Условное выражение может быть использовано, например, для присваивания максимального или минимального из значений двух переменных третьей. Например, для случая с минимальным значением пишем таким образом:

```
c = a if ( a < b ) else b
```

Выражение в скобках возвращает `True`, когда значение переменной `a` меньше, чем `b` — так что в этом случае меньшее значение присваивается переменной `c`.

Аналогично, заменив в проверочном выражении оператор «меньше» на оператор «больше», мы можем присвоить переменной `c` значение большей переменной `b`. Еще одна распространенная область применения условного оператора включает использование оператора деления по модулю, `%`, для определения того, является число четным или нечетным:

```
если-истина(нечетное)-выполняем-это if ( var % 2 != 0 ) else если-ложь(четное)-
выполняем-это
```

**На заметку**

Условное выражение имеет три операнда — проверочное выражение и два возможных возвращаемых значения.

Таким образом, если результат от деления значения переменной на два дает остаток, то число нечетное, а если остатка нет, то четное. Того же эффекта можно достичь, используя проверочное выражение ( `var % 2 == 1` ), но предпочтительней все же проверять на неравенство — так как два отличающихся значения найти легче, чем два идентичных.

1. Начните новую программу на Python с инициализации двух переменных, которые будем оценивать, целочисленными значениями.


condition.py

```
a = 1

b = 2
```

2. Теперь добавьте инструкции, чтобы вывести результаты проверки условий, описывающие первую переменную.

```
print( '\nVariable a Is :' , 'One' if ( a == 1 ) else 'Not One' )

print( 'Variable a Is :' , 'Even' if ( a % 2 == 0 ) else 'Odd' )
```

3. Затем добавьте инструкции для вывода результатов проверки логического выражения, описывающего значение второй переменной.

```
print( '\nVariable b Is :' , 'One' if ( b == 1 ) else 'Not One' )

print( 'Variable b Is :' , 'Even' if ( b % 2 == 0 ) else 'Odd' )
```

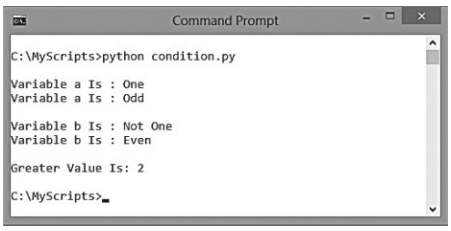
4. Добавьте инструкцию для присваивания его результата новой переменной.

```
max = a if ( a > b ) else b
```

5. Наконец, добавьте инструкцию для вывода присвоенного результата, определяющего большую из двух величин.

```
print( '\nGreater Value Is:' , max )
```

6. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу — вы увидите результаты операций с условными выражениями.




**Внимание**

Вы можете обнаружить, что некоторые программисты Python не любят условные выражения, поскольку их синтаксис противоречит принципам читаемости кода.

# Определение приоритетов

Приоритет операторов определяет порядок, которому интерпретатор Python следует при оценке выражений. Например, в выражении `3 * 8 + 4` порядок действий по умолчанию определяет, что умножение выполняется первым, так что результат равен 28 (24 + 4).

В таблице ниже перечисляются операторы в порядке убывания приоритета. Те из них, что находятся в строках выше, имеют более высокий приоритет. Приоритет операторов, находящихся на одной строке в таблице, определяется правилом «слева направо».

**На заметку**

Оператор умножения, *, выше в таблице, чем оператор сложения, поэтому умножение вычисляется перед сложением.

**Совет**

Операторы идентичности, побитовые операторы, а также операторы вхождения будут представлены дальше — здесь они включены в таблицу для полноты сведений.

| Оператор | Описание |
|---|---|
| ** | Возведение в степень |
| +<br>-<br>~ | Положительное значение<br>Отрицательное значение<br>Побитовое отрицание |
| *<br>/<br>//<br>% | Умножение<br>Деление<br>Целочисленное деление<br>Деление по модулю |
| +<br>- | Сложение<br>Вычитание |
| \| | Побитовое ИЛИ |
| ^ | Побитовое исключающее ИЛИ |
| & | Побитовое И |
| >><br><< | Побитовый сдвиг вправо<br>Побитовый сдвиг влево |
| >, < =, <, < =, = =, != | Сравнение |
| =, %=, /=, //=, -=, +=, *=, **= | Присваивание |
| is, is not | Идентичность |
| in, not in | Вхождение |
| not | Логическое отрицание |
| and | Логическое И |
| or | Логическое ИЛИ |

1. Начните новую программу на Python, проинициализировав три переменные целочисленными значениями для последующего сравнения приоритетов.

precedence.py

```
a = 2

b = 4

c = 8
```

2. Теперь добавьте инструкции для вывода результатов вычислений с порядком действий по умолчанию и явным указанием приоритета операции сложения.

```
print( '\nDefault Order:\t', a, '*', c,'+', b, '=', a * c + b )

print( 'Forced Order:\t', a, '* (', c,'+', b, ') =', a * ( c + b ) )
```

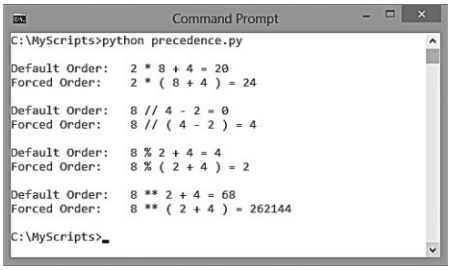
3. Затем добавьте инструкции для вывода результата вычислений с порядком действий по умолчанию и явным указанием приоритета операции вычитания.

```
print( '\nDefault Order:\t', c, '//', b, '-', a, '=', c // b - a )

print( 'Forced Order:\t', c, '// (', b,'-', a, ') =', c // ( b - a ) )
```

4. Наконец добавьте инструкции для вывода результатов вычислений с порядком операций по умолчанию, а затем с добавлением приоритета операции сложения перед делением по модулю, а также перед вычислением степени.

```
print( '\nDefault Order:\t', c, '%', a, '+', b, '=', c % a + b )

print( 'Forced Order:\t', c, '% (', a, '+', b, ') =', c % ( a + b ) )

print( '\nDefault Order:\t', c, '**', a, '+', b, '=', c ** a + b )

print( 'Forced Order:\t', c, '** (', a, '+', b, ') =', c ** ( a + b ) )
```

5. Сохраните файл в рабочем каталоге, откройте командную строку и запустите программу — вы увидите результаты работы операторов с порядком по умолчанию и с измененным порядком приоритетов.




**На заметку**

В отличие от оператора деления (/) оператор деления нацело (//) отбрасывает дробную часть числа.

**Внимание**

Никогда не полагайтесь на порядок операций по умолчанию — старайтесь всегда использовать для ясности скобки в ваших выражениях.

# Преобразование типов данных

Переменные в языке Python могут хранить данные любого типа, и очень важно различать эти типы для того, чтобы избежать ошибок при обработке данных в программе. Типов данных несколько, но пока мы рассмотрим основные из них: строковые (**str** (string)), целочисленные (**int** (integer)), с плавающей точкой (**float** (floating-point)).

Очень важно различать типы данных, особенно при присваивании переменным значений, используя пользовательский ввод, так как по умолчанию в нем хранится строковый тип данных. Строковые величины не могут быть использованы для арифметических выражений, и попытка сложить два строковых значения просто объединяет эти строки, а не использует операции над числами. Например, `'8' + '4' = '84'`.

Но, к счастью, любые типы данных, хранящиеся в переменных, могут быть легко преобразованы (приведены) к другим типам с помощью использования функций языка Python. Значение, которое требуется преобразовать, указывается в функции внутри скобок, идущих сразу после имени функции. Приведение строковых величин к целым позволяет затем использовать их уже для арифметических выражений, например, `8 + 4 = 12`.

Встроенные функции преобразования типов данных в Python возвращают новый объект, представляющий преобразованную величину. Наиболее часто используемые из этих функций представлены в таблице ниже.

**Внимание**

Преобразование типа данных с плавающей точкой (`float`) в целочисленный тип данных (`int`) отбрасывает десятичную часть числа.

| Функция | Описание |
|---|---|
| `int( x )` | Преобразует ***x*** в целое число |
| `float( x )` | Преобразует ***x*** в число с плавающей точкой |
| `str( x )` | Преобразует ***x*** в строковое представление |
| `chr( x )` | Преобразует целое ***x*** в символ |
| `unichr( x )` | Преобразует целое ***x*** в символ Юникода (Unicode) |
| `ord( x )` | Преобразует символ ***x*** в соответствующее ему целое число |
| `hex( x )` | Преобразует целое ***x*** в шестнадцатеричную строку |
| `oct( x )` | Преобразует целое ***x*** в восьмеричную строку |

1. Начните новую программу на Python, проинициализировав две переменные числовыми значениями из пользовательского ввода.


cast.py

```
a = input( 'Enter A Number: ' )

b = input( 'Now Enter Another Number: ' )
```

2. Теперь добавьте инструкции для сложения двух переменных и для вывода полученного результата, соответствующего ему типа данных, а также результата объединения строк.

```
sum = a + b

print( '\nData Type sum :' , sum , type( sum ) )
```

3. Затем добавьте инструкции для сложения двух приведенных величин и вывода результата, а также соответствующего ему типа данных — вы увидите целочисленное значение суммы.

```
sum = int( a ) + int( b )

print( 'Data Type sum :' , sum , type( sum ) )
```

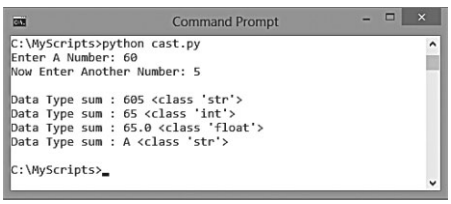
4. Добавьте инструкции для приведения типа переменной и отображения результата и его типа данных. Вы получите вывод общей суммы в формате с плавающей точкой.

```
sum = float( sum )

print( 'Data Type sum :' , sum , type( sum ) )
```

5. Наконец добавьте инструкции для приведения целочисленного представления величины и для вывода результата и его типа данных — вы получите строку символов.

```
sum = chr( int( sum ) )

print( 'Data Type sum :' , sum , type( sum ) )
```

6. Сохраните файл в рабочем каталоге, откройте командную строку и запустите программу. Вы увидите результат приведения различных типов данных.




**На заметку**

Число 65 — это ASCII-код для символа `A` в верхнем регистре.

# Манипуляции с битами

В компьютерной терминологии каждый байт состоит из восьми битов, который, в свою очередь, может содержать либо 1, либо 0 для хранения двоичного представления десятичных чисел от 0 до 255. Каждый бит является компонентом десятичного числа, если он содержит единицу. Компоненты распределены справа налево от наименее (Least Significant Bit, LSB) до наиболее значащих битов (Most Significant Bit, MSB). В примере ниже двоичное число `00110010` представляет десятичное число 50 (2+16+32):

| № бита | 8 MSB | 7 | 6 | 5 | 4 | 3 | 2 | 1LSB |
|---|---|---|---|---|---|---|---|---|
| **Десятичное** | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| **Двоичное** | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |

В языке Python можно работать с отдельными частями байта, используя так называемые **побитовые** операторы, представленные и описанные в таблице ниже.

| Оператор | Название | Операция с двоичными числами |
|---|---|---|
| \| | ИЛИ | Возвращает 1 в каждый бит, где один из сравниваемых битов имел значение 1<br>Пример: `1010 \| 0101 = 1111` |
| & | И | Возвращает 1 в каждый бит, где оба сравниваемых бита имели значение 1<br>Пример: `1010 & 1100 = 1000` |
| ~ | НЕ | Возвращает 1 в каждый бит, где ни один из двух сравниваемых битов не имел значение 1<br>Пример: `1010 ~ 0011 = 0100` |
| ^ | Исключающее ИЛИ | Возвращает 1 в каждый бит, где только один из двух сравниваемых битов имел значение 1<br>Пример: `1010 ^ 0100 = 1110` |
| << | Сдвиг влево | Сдвигает влево биты на указанное количество позиций<br>Пример: `0010 << 2 = 1000` |
| >> | Сдвиг вправо | Сдвигает вправо биты на указанное количество позиций<br>Пример: `1000 >> 2 = 0010` |

**На заметку**

Хотя многие программисты Python и не используют побитовые операторы, все же полезно знать их и понимать, как их можно применить в программе**.**

**Совет**

Каждая половина байта известна под названием **тетрада** и содержит 4 бита. Двоичные числа, описанные в данной таблице, представляют значения, хранящиеся в тетраде.

Поскольку программирование для устройств с ограниченными ресурсами не очень распространено, побитовые операторы редко используются программистами, но они могут быть полезны. Например, оператор «исключающее ИЛИ» позволяет вам поменять значение двух переменных между собой без использования третьей.


bitwise.py

1. Начните новую программу на Python с инициализации двух переменных целочисленными значениями и выведите эти первоначальные значения.

```
a = 10

b = 5

print( 'a =' , a , '\tb = ' , b )
```

2. Затем добавьте инструкцию, изменяющую десятичное значение первой переменной с помощью побитового оператора.

```
# 1010 ^ 0101 = 1111 (десятичное 15)

a = a ^ b
```

3. Теперь добавьте инструкцию, изменяющую десятичное значение второй переменной с помощью той же побитовой операции.

```
# 1111 ^ 0101 = 1010 (десятичное 10)

b = a ^ b
```

4. Добавьте инструкцию, изменяющую десятичное значение первой переменной еще одной побитовой операцией.

```
# 1111 ^ 1010 = 0101 (десятичное 5)

a = a ^ b
```

5. Наконец добавьте инструкцию для вывода измененных значений.

```
print( 'a =' , a , '\tb = ' , b )
```

6. Сохраните файл в рабочем каталоге, откройте командную строку и запустите программу — вы увидите результаты побитовых операций.


**Внимание**

Не путайте побитовые операторы с логическими. Первые сравнивают двоичные числа, в то время как вторые оценивают логические значения.

# Заключение

- Арифметические операторы могут формировать выражения с двумя операндами, организуя операции сложения `+`, вычитания `-`, умножения `*`, деления `/`, целочисленного деления `//`, деления по модулю `%` и возведения в степень `**`.
- Оператор присваивания `=` можно комбинировать с арифметическими операторами, чтобы производить арифметические вычисления и сразу же присваивать их результаты.
- Операторы сравнения формируют выражения, сравнивающие два операнда. Например, равенство `==`, неравенство `!=`, а также больше `>`, меньше `<`, больше либо равно `>=`, меньше либо равно `<=`.
- «Логическое И», `and`, и «логическое ИЛИ», `or`, являются операторами, формирующими выражение для оценки двух операндов и возвращающими булево значение `True` (Истина) или `False` (Ложь).
- Оператор «логическое НЕ», `not`, возвращает обратное булево значение единственного операнда.
- Условное выражение `if-else` оценивает заданное условие и возвращает один из двух операндов в зависимости от результата оценки.
- Операции в выражениях выполняются в соответствии с правилами приоритета, если явно не используются дополнительные скобки.
- Тип данных переменной может быть преобразован к другому типу с помощью встроенных в Python функций `int()`, `float()` и `str()`.
- Встроенная функция `type()` определяет, к какому типу данных принадлежит указанная переменная.
- Побитовые операторы «ИЛИ», `|`, «И», `&`, «НЕ», `~`, и «исключающее ИЛИ», `^`, сравнивают два бита и возвращают соответствующее значение, в то время как операторы «сдвиг влево», `<<`, и «сдвиг вправо», `>>`, производят сдвиг указанного количества значащих битов в соответствующем направлении.

# 3 Конструирование инструкций

*В этой главе описывается, как с помощью создаваемых вами инструкций можно оценивать выражения для определения направления работы программы на языке Python.*

- **Списки**
- **Работа со списками**
- **Неизменяемые списки**
- **Элементы ассоциативного списка**
- **Ветвление с помощью условного оператора**
- **Цикл while**
- **Обход элементов в цикле**
- **Выход из цикла**
- **Заключение**

# Списки

В языке Python любой переменной должно быть присвоено начальное значение (она обязана быть проинициализирована), иначе интерпретатор выдаст сообщение об ошибке (переменная не определена).

Можно проинициализировать несколько переменных одним начальным значением в одной инструкции с использованием последовательности операторов присваивания. Например, чтобы одновременно присвоить одно значение трем переменным, мы пишем:

```
a = b = c = 10
```

С другой стороны, несколько переменных можно проинициализировать различными начальными значениями в одном-единственном операторе присваивания, используя запятые как разделители. Например, для одновременного присваивания различных значений трем переменным, мы пишем:

```
a , b , c = 1 , 2 , 3
```

В отличие от обычных переменных, которые содержат единственный элемент данных, в Python существует так называемый список, в котором может храниться несколько элементов данных. Данные хранятся последовательно в «элементах» списка, которые индексируются числовыми значениями, начиная с нуля. То есть первое значение сохраняется в элементе 0, второе — в элементе 1 и так далее.

Список создается так же, как и другая переменная, но инициализируется с помощью присваивания в виде разделенного запятой списка значений, заключенных в квадратные скобки, например создание списка с именем `nums` выглядит так:

```
nums = [0 , 1 , 2 , 3 , 4 , 5]
```

Обратиться к отдельному элементу списка можно, используя имя списка и следующий за ним индекс элемента в квадратных скобках. Это значит, что `nums[1]` обращается ко второму элементу указанного выше списка (а не к первому, так как нумерация начинается с нуля).

| | [0] | [1] | [2] |
|---|---|---|---|
| [0] | 1 | 2 | 3 |
| [1] | 4 | 5 | 6 |

Списки могут иметь более чем один индекс, то есть быть многомерными. Списки, содержащие три и более индекса, мало распространены, но, например, двухмерные очень часто используются для хранения пар координат X, Y.

Список строковых величин можно также рассматривать как многомерный, так как каждая строка сама по себе является списком символов. Следовательно, к каждому символу можно обратиться по его числовому индексу внутри определенной строки.

1. Начните новую программу на Python, проинициализировав список из трех элементов, содержащий строковые значения.

```
quarter = [ 'January' , 'February' , 'March' ]
```

2. Затем добавьте инструкции для отдельного вывода значений, содержащихся в каждом элементе списка.

```
print( 'First Month :' , quarter[0] )

print( 'Second Month :' , quarter[1] )

print( 'Third Month :' , quarter[2] )
```

3. Теперь добавьте инструкцию для создания многомерного списка из двух элементов, каждый из которых сам по себе является списком из трех элементов, содержащих целые значения.

```
coords = [ [ 1 , 2 , 3 ] , [ 4 , 5 , 6 ] ]
```

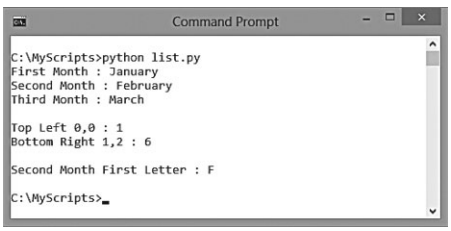
4. Затем добавьте инструкции для отображения значений, содержащихся в двух определенных элементах внутреннего списка.

```
print( '\nTop Left 0,0 :' , coords[0][0] )

print( 'Bottom Right 1,2 :' , coords[1][2] )
```

5. Наконец добавьте инструкцию для вывода только одного символа строковой переменной.

```
print( '\nSecond Month First Letter :' , quarter[1][0] )
```

6. Сохраните файл в рабочем каталоге, откройте командную строку и запустите программу — вы увидите вывод значений элементов списка.





list.py

**Совет**

Строковые индексы могут быть и отрицательными — -1 будет указывать на последний символ, начиная справа.

**На заметку**

Структуры с использованием циклов, которые будут представлены позднее в этой главе, очень часто используют итерации с элементами списка.

# Работа со списками

Списки, содержащие множественные элементы данных, широко используются в программировании на Python. Для работы с ними существует множество так называемых методов, к которым можно обращаться через **точечную** запись.

| Метод списка | Описание |
|---|---|
| *list*.append(*x*) | Добавляет элемент *x* в конец списка |
| *list*.extend(*L*) | Добавляет все элементы списка *L* в конец списка |
| *list*.insert(*i*,*x*) | Вставляет элемент *x* в позицию перед индексом *i* в списке |
| *list*.remove(*x*) | Удаляет первый элемент *x* из списка |
| *list*.pop(*i*) | Удаляет элемент с индексом *i* и возвращает его |
| *list*.index(*x*) | Возвращает индекс первого элемента *x* в списке |
| *list*.count(*x*) | Возвращает количество вхождений элемента *x* в список |
| *list*.sort() | Сортирует элементы списка по возрастанию |
| *list*.reverse() | Обращает порядок следования элементов |

**Совет**

Для списков, содержащих как числовые, так и строковые значения, метод sort() возвращает отсортированный список, где расположены сначала числовые значения, а затем строковые, например 1, 2, 3, A, B, C.

В языке Python существует полезная функция len(*L*), возвращающая размер списка *L*, то есть общее количество элементов, содержащихся в списке. Подобно методам index() и count() в данном случае возвращаемое значение является числовым и не может быть напрямую объединено с текстом для вывода текстовой строки.

Однако с помощью функции str(*n*) числовые величины можно привести к строковому представлению, чтобы позже использовать их для сложения с другими строками. Аналогично с помощью функции str(*L*) можно возвратить строковое представление всего списка. В обоих случаях нужно помнить, что первоначальные версии остаются неизменными, а возвращаемое представление — это только их копия.

**Совет**

В Python существует функция int(*s*), которая возвращает числовое представление строковой величины *s*.

Отдельные элементы списка могут быть удалены с помощью указания их индекса в качестве параметра для функции del(*i*). Можно удалить одиночный элемент, указав номер его позиции *i*, либо набор элементов, используя запись *i1*:*i2*, которая указывает диапазон индексов первого и последнего удаляемых элементов. Это значит, что удалятся все элементы с индексами от *i1* до *i2*, исключая последний.

1. Начните новую программу на Python, проинициализировав два списка из трех элементов каждый, содержащие строковые значения.

pop.py

```
basket = [ 'Apple' , 'Bun' , 'Cola' ]

crate = [ 'Egg' , 'Fig' , 'Grape' ]
```

2. Теперь добавьте инструкции для вывода содержимого первого списка и его длины.

```
print( 'Basket List:' , basket )

print( 'Basket Elements:' , len( basket ) )
```

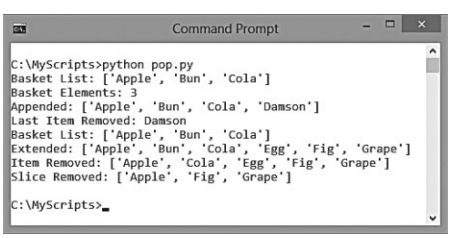
3. Напишите инструкции для добавления элемента в список, отображения всех элементов, затем удаления последнего элемента и повторного вывода всех элементов.

```
basket.append( 'Damson' )

print( 'Appended:' , basket )

print( 'Last Item Removed:' , basket.pop() )

print( 'Basket List:' , basket )
```

4. Наконец добавьте инструкции для расширения первого списка элементами второго, вывода всех элементов первого списка, затем удаления элементов и вывода элементов первого списка еще раз.

```
basket.extend( crate )

print( 'Extended:' , basket )

del basket[1]

print( 'Item Removed:' , basket )

del basket[1:3]

print( 'Slice Removed:' , basket )
```

5. Сохраните файл в рабочем каталоге, откройте командную строку и запустите программу. Вы увидите результат обработки списков.

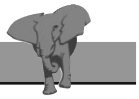



**На заметку**

Последний индексный номер в диапазоне для удаления указывает, на каком элементе остановиться, но сам элемент с этим номером не удаляется.

# Неизменяемые списки

## Кортеж



**На заметку**

Подобно спискам, элементы кортежа индексируются, начиная с нуля.

Значения элементов обычного списка могут меняться программистом по мере выполнения программы. Но список может быть создан также с фиксированными (неизменяемыми) значениями, которые остаются постоянными на протяжении всей программы. Неизменяемый список в языке Python называется **кортежом** и создается присваиванием значений, разделенных запятой и стоящих в скобках. Такой процесс присваивания называется **упаковкой кортежа**:

```
colors-tuple = ( 'Red' , 'Green' , 'Red' , 'Blue' , 'Red' )
```

**Внимание**

Для распаковки кортежа количество переменных должно быть равно количеству элементов кортежа.

К отдельному элементу кортежа можно обращаться, используя запись имени кортежа и индекса в квадратных скобках. Все значения, хранящиеся внутри кортежа, могут быть присвоены отдельным переменным. Такой процесс называется **распаковкой последовательности**:

```
a , b , c , d , e = colors-tuple
```

## Множество

**Совет**

В документации по языку Python вы можете найти дополнительную информацию о других методах.

В обычном списке элементы могут повторяться, как и в кортеже, описанном чуть выше, но существует возможность создавать такой список, где нет повторяющихся элементов. Неизменяемый список уникальных значений называется **множеством** и создается он присвоением значений через запятую и заключением их в фигурные скобки:

```
phonetic-set = { 'Alpha' , 'Bravo' , 'Charlie' }
```

Нельзя обратиться к отдельному элементу множества, используя имя множества и квадратные скобки, содержащие индекс. Вместо этого у множеств существуют методы для работы с ними.

| Метод множества | Описание |
|---|---|
| `set.add(x)` | Добавляет элемент *x* в множество |
| `set.update(x,y,z)` | Добавляет несколько элементов в множество |
| `set.copy()` | Возвращает копию множества |
| `set.pop()` | Удаляет один элемент из множества случайным образом |
| `set.discard( i )` | Удаляет из множества элемент с порядковым номером *i* |
| `set1.intersection(set2)` | Возвращает элементы, принадлежащие обоим множествам |
| `set1.difference(set2)` | Возвращает элемента из множества *set1*, которых нет в *set2* |

Для того чтобы определить принадлежность той или иной структуры данных к классу списков, применяется функция `type()` Python, а чтобы найти значение в этом списке, используется встроенный оператор вхождения `in`.

1. Начните новую программу на Python, проинициализировав кортеж, затем выведите его содержимое, длину и тип.


set.py

```
zoo = ( 'Kangaroo' , 'Leopard' , 'Moose' )

print( 'Tuple:' , zoo , '\tLength:' , len( zoo ) )

print( type( zoo ) )
```

2. Теперь проинициализируйте множество, затем добавьте в него еще один элемент и потом выведите его содержимое, длину и тип.

```
bag = { 'Red' , 'Green' , 'Blue' }

bag.add( 'Yellow' )

print( '\nSet:' , bag , '\tLength' , len( bag ) )

print( type( bag ) )
```

3. Добавьте инструкции для поиска двух значений в множестве.

```
print( '\nIs Green In bag Set?:' , 'Green' in bag )

print( 'Is Orange In bag Set?:' , 'Orange' in bag )
```

4. Наконец проинициализируйте второе множество, также выведите его содержимое, длину, а потом общие элементы для обоих множеств.

```
box = { 'Red' , 'Purple' , 'Yellow' }

print( '\nSet:' , box , '\t\tLength' , len( box ) )

print( 'Common To Both Sets:' , bag.intersection( box ) )
```

5. Сохраните файл в рабочем каталоге, откройте командную строку и запустите программу — вы увидите результаты работы с элементами кортежа и множества.


**Совет**

Множество может быть создано также, используя конструктор `set()`, в котором в скобках заключен список, а неизменяемое множество — с помощью конструктора `frozenset()`.

# Элементы ассоциативного списка

В языке программирования Python **словарем** называется некоторый контейнер, который может содержать несколько элементов данных в виде набора пар *ключ*:*значение*. В отличие от обычного списка, к элементам которого можно обратиться по их числовому индексу, к значениям в словаре можно обращаться по связанному с ним ключу. Ключ должен быть уникальным в пределах этого словаря и является, как правило, строкой, хотя числовые значения также иногда используются.

**Совет**

В других языках программирования список часто называется **массивом**, а словарь — **ассоциативным массивом**.

Создание словаря — это присваивание переменной с выбранным вами именем структуры в фигурных скобках, в которой через запятую перечислены строковые пары ключ:значение. Строки должны быть заключены в кавычки, а между ключом и связанным с ним значением обязан стоять символ двоеточия.

Пара ключ:значение может быть удалена из словаря. Для этого нужно указать после ключевого слова `del` имя словаря и имя ключа в квадратных скобках. И наоборот, с помощью присвоения значения новому ключу можно добавить пару ключ:значение в имеющийся словарь.

Для словарей в языке Python существует метод `keys()`, с помощью которого можно вернуть список всех ключей словаря в случайном порядке, используя суффиксную запись к имени словаря. Если вам нужно отсортировать ключи в алфавитно-цифровом порядке, просто заключите эту инструкцию в скобки, включив его в функцию Python `sorted()`.

С помощью оператора `in` можно определять, содержит ли словарь нужный ключ. Для этого используется синтаксис *ключ* `in` *словарь*. Поиск будет выдавать логическое значение `True`, если ключ найден в указанном словаре, в противном случае появится значение `False`.

**Совет**

Данные очень часто представлены в виде пар ключ:значение. Например, когда вы заполняете веб-форму, текстовое значение, которое вы вводите в поле ввода, обычно связано с именем этого текстового поля, как с ключом.

В заключение резюмируем, что различными типами контейнеров данных, доступных при программировании на языке Python, являются:

- **переменная** — хранит одиночное значение;
- **список** — хранит несколько значений, упорядоченных по индексам;
- **кортеж** — хранит несколько фиксированных значений в определенной последовательности;
- **множество** — хранит несколько уникальных значений в неупорядоченном наборе;
- **словарь** — хранит несколько неупорядоченных пар ключ:значение.

1. Начните новую программу на Python, проинициализировав словарь, а затем выведя содержащиеся в нем пары ключ:значение.


dict.py

```
dict = { 'name' : 'Bob' , 'ref' : 'Python' , 'sys' : ''Win' }

print( 'Dictionary:' , dict )
```

2. Теперь выведите на экран единичное значение, указав на него ссылку по ключу.

```
print( '\nReference:' , dict[ 'ref' ] )
```

3. Затем выведите все ключи, содержащиеся в словаре.

```
print( '\nKeys:' , dict.keys() )
```

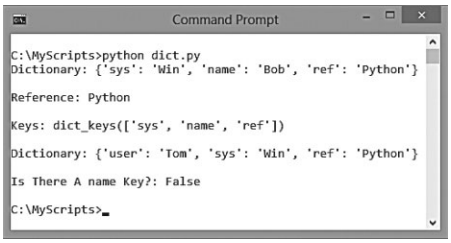
4. Удалите одну пару из словаря, а затем добавьте новую и выведите новое содержимое словаря.

```
del dict[ 'name' ]

dict[ 'user' ] = 'Tom'

print( '\nDictionary:' , dict )
```

5. Запустите поиск определенного ключа и выведите результаты этого поиска.

```
print( '\nIs There A name Key?:' ,'name' in dict )
```

6. Сохраните файл в рабочем каталоге, откройте командную строку и запустите программу — вы увидите результаты работы с ключами и значениями.




**Внимание**

Обратите внимание, что для того, чтобы избежать преждевременного завершения строки, использовалась управляющая последовательность внутри кавычек.

# Ветвление с помощью условного оператора

В языке Python оператор, заданный ключевым словом **if**, осуществляет проверку условия, которое оценивает заданное выражение на предмет значений **True** или **False**. Это позволяет программе продолжать действия в различных направлениях в зависимости от результата выполнения этой проверки. Данная процедура известна как **условное ветвление**.

**Внимание**

Отступы в коде очень важны в языке Python, поскольку они определяют блоки кода для интерпретатора — в других языках программирования для этого используются фигурные скобки.

Проверяемое выражение должно заканчиваться двоеточием. Затем на отдельных строках должны идти инструкции, которые выполняются при условии успешности проверки. Каждая из этих строк должна быть отделена отступом от базовой строки со словом **if**. Размер отступа не так важен, но он должен быть одинаков для каждой из этих строк. Таким образом, синтаксис выглядит так:

```
if проверочное-выражение :
выполняемая-инструкция-когда-проверочное-выражение-истинно
выполняемая-инструкция-когда-проверочное-выражение-истинно
```

С помощью ключевого слова **else** можно изменить данную конструкцию, добавив инструкции для выполнения в случае, если проверка не была успешна. Ключевое слово **else** ставится после инструкций, выполняемых при успешной проверке, и должно заканчиваться двоеточием, а также располагаться строго под ключевым словом **if**, а соответствующие ему инструкции опять же должны быть отделены отступами. Таким образом, синтаксис выглядит следующим образом:

```
if проверочное-выражение :
выполняемая-инструкция-когда-проверочное-выражение-истинно
выполняемая-инструкция-когда-проверочное-выражение-истинно
else :
выполняемая-инструкция-когда-проверочное-выражение-ложно
выполняемая-инструкция-когда-проверочное-выражение-ложно
```

**На заметку**

Конструкция **if**: **elif**: **else**: в языке Python является эквивалентом инструкций **switch** или **case**, которые можно встретить в других языках программирования.

После блока проверки **if** может использоваться еще ключевое слово **elif** (else if), которое предполагает осуществление альтернативной проверки и выполнение соответствующих инструкций. Это ключевое слово должно быть привязано к слову **if**, завершать его тоже нужно двоеточием, и его инструкции также должны выделяться отступами. Затем, чтобы завершить альтернативную проверку, может добавляться финальное ключевое слово **else** на случай, если проверка не пройдена. Синтаксис для полной конструкции **if**: **elif**: **else** выглядит таким образом:

```
if проверочное-выражение-1 :
выполняемая-инструкция-когда-проверочное-выражение-1-истинно
```

```
выполняемая-инструкция-когда-проверочное-выражение-1-истинно
elif проверочное-выражение-2 :
выполняемая-инструкция-когда-проверочное-выражение-2-истинно
выполняемая-инструкция-когда-проверочное-выражение-2-истинно
else :
выполняемая-инструкция-когда-проверочные-выражения-ложны
выполняемая-инструкция-когда-проверочные-выражения-ложны
```

1. Начните новую программу на Python, проинициализировав переменную, содержащую целочисленное значение, при помощи пользовательского ввода.


if.py

```
num = int( input( 'Please Enter A Number: ' ) )
```

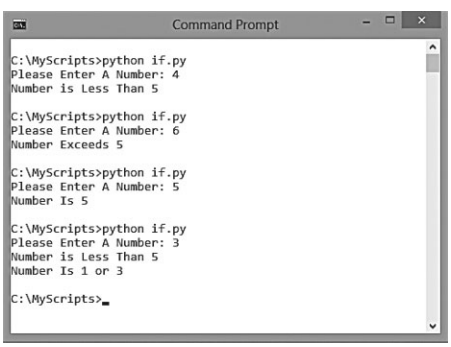
2. Теперь выполните проверку переменной и выведите соответствующий результат.

```
if num > 5 :

print( 'Number Exceeds 5' )

elif num < 5 :

print( 'Number is Less Than 5' )

else :

print( 'Number Is 5' )
```

3. Затем выполните проверку переменой еще раз, используя два выражения и выводя результат только в случае успеха.

```
if num > 7 and num < 9 :

print( 'Number is 8' )

if num == 1 or num == 3 :

print( 'Number Is 1 or 3' )
```

4. Сохраните файл в рабочем каталоге, откройте командную строку и запустите программу — вы увидите результат выполнения условного ветвления.




**На заметку**

Как мы помним, при пользовательском вводе переменная имеет строковый тип и для арифметических сравнений должна быть приведена к целочисленному типу с помощью функции приведения `int()`.

**Совет**

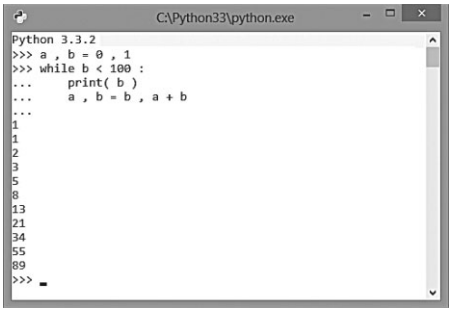
Ключевое слово `and` предполагает, что результат оценки равен `True`, только если обе проверки успешны, в то время как ключевое слово `or` дает в результате `True`, когда успешна хотя бы одна из проверок.

# Цикл `while`

Циклом является кусок кода в программе, который автоматически повторяется. Одно полное исполнение инструкций внутри цикла называется **итерацией** или **проходом**. Размер цикла контролируется проверочным условием, создаваемым внутри цикла. Цикл продолжается, пока проверочное выражение равно `True`, и заканчивается в той точке, когда оно становится равным `False`.

В программировании на Python циклы создаются с помощью ключевого слова `while`. После него следует проверочное выражение, а затем символ двоеточия. Ниже должны идти инструкции, выполняемые при успешной проверке выражения. Каждая строка обязана иметь один и тот же отступ от строки с ключевым словом `while`. Этот блок должен включать в каком-либо месте инструкцию, изменяющую значение проверочного выражения на противоположное — иначе цикл будет бесконечным.

При работе в интерактивном режиме отступы блоков кода на языке Python также должны соблюдаться — как в этом примере, который генерирует последовательность чисел Фибоначчи с помощью цикла `while`:




Циклы могут быть вложенными один в другой, при этом на каждой итерации наружного цикла уже выполнены все итерации внутреннего цикла. Для удобства можно проинициализировать перед каждым определением цикла так называемую переменную-«счетчик», задав ей первоначальное значение и увеличивая его на каждой итерации, а также включив данную переменную в проверочное выражение. Цикл будет завершаться при неудачной проверке выражения.

**На заметку**

В отличие от других ключевых слов языка Python, `True` и `False` начинаются с символа в верхнем регистре.

**Совет**

Интерпретатор предлагает знак многоточия в качестве подсказки, когда он ожидает дальнейшие инструкции. Чтобы продолжить работать с отступами и инструкциями, нажмите клавишу **Tab**, а затем нажимайте **Enter** для продолжения. Чтобы закончить работу с операторами, нажмите **Enter** сразу.

1. Начните новую программу на Python, проинициализировав переменную-счетчик и определив наружный цикл, используя эту переменную в проверочном выражении.

while.py

```
i = 1

while i < 4 :
```

2. Затем добавьте инструкции с отступами для вывода значения счетчика и приращения его значения на каждой итерации цикла.

```
    print( '\nOuter Loop Iteration:' , i )

    i += 1
```

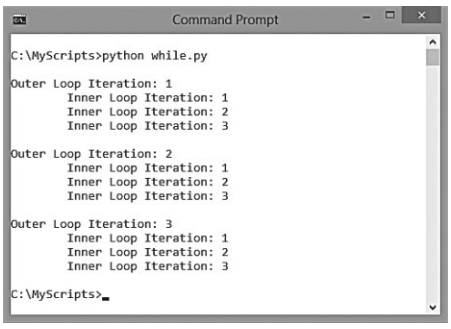
3. Теперь (по-прежнему с отступами) проинициализируйте вторую переменную-счетчик и определите внутренний цикл, используя эту переменную в проверочном выражении.

```
    j = 1

    while j < 4 :
```

4. Наконец добавьте с дальнейшим отступом инструкции для вывода значения счетчика, а также для приращения его величины на каждой итерации.

```
        print( '\tInner Loop Iteration:' , j )

        j += 1
```

5. Сохраните файл в рабочем каталоге, откройте командную строку и запустите программу — вы увидите результаты вывода на каждой итерации цикла.




**Совет**

Вывод внутреннего цикла будет отделяться отступом от вывода наружного цикла с помощью управляющей последовательности `\t`.

**Совет**

Оператор присваивания, `+=`, используемый в строке `i += 1` — это просто сокращенная запись выражения `i = i+1`. Вы можете также использовать сокращения `*=`, `/=` и `-=` в записи для присваивания значений переменным.

# Обход элементов в цикле

В языке Python для перебора элементов любого указанного списка используется оператор `for in`. После `in` указывается имя списка. Инструкция должна заканчиваться символом двоеточия, после него идут инструкции, которые выполняются на каждой итерации цикла. Например:

```
for элемент in имя-списка :
выполняемые-инструкции-на-каждой-итерации
выполняемые-инструкции-на-каждой-итерации
```

**На заметку**

С помощью функции `range()` можно получить как убывающую, так и возрастающую последовательность.

С помощью инструкции `for in` можно обходить в цикле элементы списка, кортежа, множества, а также ключи словаря. Поскольку строка — это не что иное, как список символов, каждый символ строки можно обходить в цикле, используя инструкцию `for in`.

С помощью цикла `for in` вы можете обходить элементы любого списка или символы строки в том порядке, в котором они появляются, но нельзя указать явно количество итераций цикла, условие остановки, либо размер шага итераций. Однако вы можете воспользоваться функцией языка `range()`, чтобы сгенерировать последовательность чисел, используемых для итераций. Эта функция генерирует последовательность, начинающуюся с нуля и кончающуюся числом в скобках, не включая его. Например, `range(5)` генерирует последовательность 0,1,2,3,4.

**Внимание**

В отличие от других языков, в языке Python в цикле `for` нельзя использовать размер шага и указывать конечный элемент.

Вы можете указать в качестве параметров функции `range()` два числа, разделенных запятой, — начальную и конечную величину. Например, `range(1,5)` генерирует последовательность 1,2,3,4. Также можно использовать еще один вариант функции, которая принимает три параметра, разделенных запятой — начальная величина, конечная величина и шаг. Например, `range(1,14,4)` генерирует последовательность 1,5,9,13.

Существует полезная функция `enumerate()`, с помощью которой, указав в качестве параметра имя списка, вы можете вывести все индексы и связанные с ними значения.

Допускается делать обход нескольких списков одновременно. Для этого вам нужно использовать специальную функцию `zip()`, указав в качестве параметров через запятую имена списков, и на выходе вы получите попарно значения элемента с одним и тем же индексом через запятую.

При обходе элементов словаря вы можете вывести пары ключ:значение, используя метод словаря `items()` и указав после ключевого слова `for` два параметра — один для имени ключа, другой для его значения.

1. Начните новую программу на Python, проинициализировав сначала список, потом кортеж, затем словарь.


for.py

```
chars = [ 'A' , 'B', 'C' ]

fruit = ( 'Apple' , 'Banana' , 'Cherry' )

dict = { 'name' : 'Mike' , 'ref' : 'Python' , 'sys' : 'Win' }
```

2. Теперь добавьте инструкции для вывода значений всех элементов списка.

```
print( '\nElements:\t' , end = ' ' )

for item in chars :

print( item , end = ' ' )
```

3. Затем добавьте инструкции для вывода значения всех элементов списка, а также их индексов.

```
print( '\nEnumerated:\t' , end = ' ' )

for item in enumerate( chars ) :

print( item , end = ' ' )
```

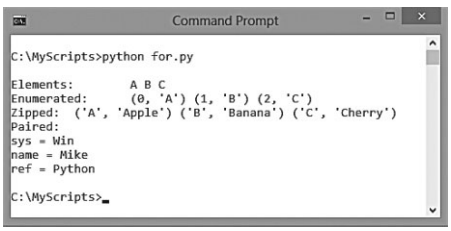
4. Теперь добавьте инструкции для вывода всех элементов списка и кортежа.

```
print( '\nZipped:\t' , end = ' ' )

for item in zip( chars , fruit ) :

print( item , end = ' ' )
```

5. Наконец добавьте инструкции для вывода всех имен ключей словаря и связанных значений элементов.

```
print( '\nPaired:' )

for key , value in dict.items() :

print( key , '=' , value )
```

6. Сохраните файл в рабочем каталоге, откройте командную строку и запустите вашу программу — вы увидите результаты обхода в цикле элементов разных структур.




**Совет**

При программировании на Python любая структура, содержащая множественные элементы, которые можно обойти в цикле, считается **итерируемой**.

# Выход из цикла

Для того чтобы принудительно выйти из цикла, когда выполняется какое-то условие, используется ключевое слово **break**. Оно располагается внутри инструкции цикла после проверяемого выражения. Когда проверка возвращает значение **True**, цикл немедленно заканчивается, и программа передает управление следующей инструкции. Например, если `break` стоит во вложенном внутреннем цикле, то управление будет передаваться следующей итерации наружного цикла.


**nest.py**

1. Начните новую программу на Python с инструкции, создающей цикл, который выполняется три раза.

```
for i in range( 1, 4 ) :
```

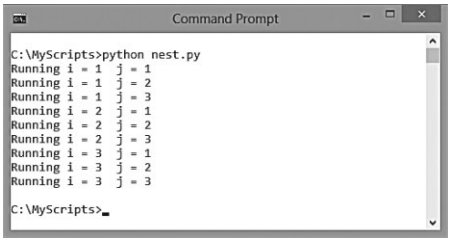
2. Добавьте инструкцию с использованием отступа, в котором создается вложенный внутренний цикл, запускаемый также три раза.

```
    for j in range( 1, 4 ) :
```

3. Теперь во внутреннем цикле, сделав еще один отступ, добавьте вывод значений счетчиков (как внутреннего цикла, так и наружного) для каждого прохода внутреннего цикла.

```
        print( 'Running i=' , i , 'j=' , j )
```

4. Сохраните файл в рабочем каталоге, откройте командную строку и запустите программу — вы увидите вывод значений счетчика в каждой итерации цикла.



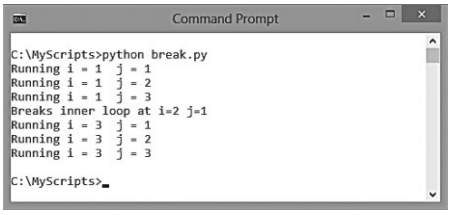



**Совет**

Сравните этот пример с примером для цикла **while** ранее в этой главе.

5. Теперь добавьте инструкцию **break** в самое начало блока инструкций внутреннего цикла, чтобы организовать выход из него — затем сохраните файл и запустите программу еще раз.

```
if i == 2 and j== 1 :
          print( 'Breaks inner loop at i=2 j=1' )
break
```





break.py

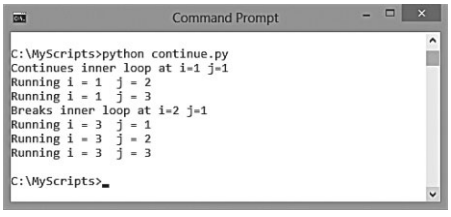
**На заметку**

В данном случае инструкция **break** заставляет пропустить все три итерации внутреннего цикла на втором проходе наружного.

Иногда нужно пропустить одну из итераций цикла, если выполняется какое-либо условие. Для этого используется ключевое слово **continue**, которое располагается внутри блока инструкций цикла и предваряется проверочным выражением. Когда результат проверки возвращает значение **True**, данная итерация заканчивается и программа переходит к следующей.

6. Добавьте ключевое слово **continue** в самое начало блока внутреннего цикла, для того чтобы пропустить первую итерацию внутреннего цикла, затем сохраните файл и запустите программу заново.

```
if i == 1 and j == 1 :

print( 'Continues inner loop at i=1 j=1' )

continue
```





continue.py

**На заметку**

В данном случае инструкция **continue** просто дает возможность пропустить первую итерацию внутреннего цикла при первом проходе наружного цикла.

# Заключение

- В языке Python имеется возможность проинициализировать сразу несколько переменных в одной инструкции, используя множественные присваивания.
- Списки представляют собой переменные, которые могут хранить множественные элементы данных в пронумерованных элементах, индекс которых начинается с нуля.
- К данным, хранящимся в элементе списка можно обратиться, используя имя списка и индекс элемента в квадратных скобках.
- Функция `len()` возвращает длину указанного списка.
- Кортеж в языке Python — это неизменяемый список, элементы которого можно присвоить отдельным переменным с помощью так называемой распаковки последовательности.
- К данным, хранящимся в элементе кортежа, можно обращаться, используя имя кортежа и номер индекса в квадратных скобках.
- Множество в языке Python — это упорядоченный набор уникальных элементов, значения которых могут быть обработаны с помощью методов. К данным, хранящимся в множестве, нельзя обращаться по индексу элемента.
- Словарь в языке Python представляет собой список пар ключ:значение, в которых каждый ключ должен быть уникален.
- К данным, хранящимся в элементе словаря, можно обращаться, используя имя словаря и ключ в квадратных скобках.
- Ключевое слово `if` языка Python осуществляет проверку условного выражения на равенство логическим значениям `True` или `False`.
- С помощью ключевых слов `if`, `else` и `elif` можно организовать условное ветвление программ.
- Цикл `while` повторяется до тех пор, пока проверочное выражение не возвратит значение `False`.
- С помощью цикла `for in` можно перебирать элементы указанного списка, либо строки.
- Функция `range()` генерирует числовую последовательность, которую можно использовать для указания длины цикла `for in`.
- Ключевые слова `break` и `continue` используются для прерывания итераций цикла.

# 4 Определение функций

*В этой главе демонстрируется, как создавать функции, которые можно вызывать в программе для выполнения набора инструкций.*

- **Область видимости переменных**
- **Подстановка аргументов**
- **Возвращение значений**
- **Использование обратного вызова**
- **Добавление заполнителей**
- **Генераторы в Python**
- **Обработка исключений**
- **Отладка с помощью инструкции assert**
- **Заключение**

# Область видимости переменных

В предыдущих примерах данной книги использовались встроенные функции языка Python, такие, как, например, `print()`. Однако в большинстве программ на языке Python может содержаться значительное число пользовательских функций, вызываемых по мере необходимости.

**На заметку**

Инструкции в теле функции должны быть отделены отступом от строки определения самой функции, чтобы интерпретатор мог распознать блок.

Пользовательскую функцию можно создать, используя ключевое слово `def` (definition), после которого следует выбранное вами имя функции и скобки. В качестве имени для своей функции программист может выбирать любой идентификатор за исключением ключевых слов языка Python, а также существующих имен встроенных функций. Строка с определением функции обязана заканчиваться символом двоеточия. Инструкции, которые должны выполняться при вызове функции (тело функции), располагаются на строках ниже с использованием отступа. Синтаксис определения функции выглядит следующим образом:

```
def имя-функции ( ) :
исполняемое-выражение
исполняемое-выражение
```

После того как выполнены инструкции в теле функции, программа передает управление в ту точку, которая следует за вызовом этой функции. Этот принцип модульности, обеспечивающий изоляцию частей программы, которые должны выполняться периодически, очень полезен в программировании.



**Внимание**

По возможности избегайте использования глобальных переменных для предотвращения конфликтов. Рекомендуется использовать локальные переменные, если это допустимо.

При создании пользовательских функций необходимо понимать принцип доступности переменных в программе (**область видимости** переменных).

- К переменным, создаваемым вне функции, можно обращаться из инструкций внутри функций — они являются **глобальными**.
- К переменным, создаваемым внутри функций, нельзя обращаться извне — они имеют **локальную** область видимости.

Ограниченная доступность локальных переменных означает, что переменные с одним и тем же именем без каких-либо последствий могут появляться в различных функциях.

Если вы хотите, чтобы к локальной переменной был доступ из любого места, ее нужно сначала объявить с использованием ключевого слова `global`, после которого следует имя переменной. После этого ей мож-

но присваивать значение сколько угодно раз, и оно будет доступно из любого места программы. В тех случаях, когда две переменных — глобальная и локальная — имеют одно имя, функция будет использовать локальную версию.

1. Начните новую программу на Python с инициализации переменной.


scope.py

```
global_var = 1
```

2. Теперь создайте функцию с именем `my_vars` для вывода значения, содержащегося в глобальной переменной.

```
def my_vars() :

print( 'Global Variable:' , global_var )
```

3. Затем, используя отступ, добавьте к функции блок, где проинициализируйте локальную переменную и выведите значение, которое она содержит.

```
local_var = 2

print( 'Local variable:' , local_var )
```

4. Теперь добавьте инструкции в блок функций, не забывая про отступы, и создайте принудительно глобальную переменную, присвоив ей начальное значение.

```
global inner_var

inner_var = 3
```

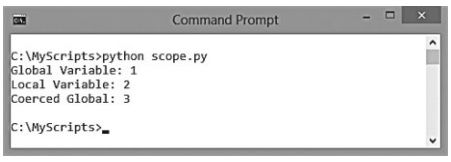
5. После этого добавьте вызов данной функции для выполнения инструкций, которые она содержит.

```
my_vars()
```

6. Наконец добавьте инструкцию для вывода значений, содержащихся в глобальной переменной.

```
print( 'Coerced Global:' , inner_var )
```

7. Сохраните файл в рабочем каталоге, откройте командную строку и запустите программу — вы увидите результаты работы пользовательской функции, выводящей значения переменных.




**Совет**

Переменные, которые не являются глобальными, но встречаются в некоторых наружных областях, можно использовать, если объявить их с помощью ключевого слова `nonlocal`.

# Подстановка аргументов

При определении пользовательской функции в языке Python вы можете также указывать необязательный параметр (так называемый **аргумент**). После этого аргументу можно передать значение, указав его в скобках при вызове функции, и тогда она станет использовать переданное ей значение, ссылаясь на имя аргумента. Например, определение функции, принимающей аргумент для вывода на печать, выглядит следующим образом:

```
def echo( user ) :

print( 'User:' , user )
```

**На заметку**

Правила именования аргументов функций те же самые, что и для переменных и функций.

Теперь в вызове этой функции нужно указать значение, которое передается аргументу в скобках для того, чтобы вывести его на печать:

```
echo( 'Mike' )
```

В функцию можно передавать как один, так и несколько аргументов (параметров), при этом значения в скобках разделяются запятой:

```
def echo( user , lang , sys ) :

print('User:' , user , 'Language:' , lang , 'Platform:' , sys )
```

При вызове функции, определение которой содержит аргументы, нужно включать то же самое количество передаваемых значений. Например, для вызова функции `echo`, определенной чуть выше, мы запишем:

```
echo( 'Mike' , 'Python' , 'Windows' )
```

При вызове функции передаваемые значения должны идти в том же порядке, что и аргументы, если только вызов не происходит подобным образом:

```
echo( lang = 'Python' , user = 'Mike' , sys = 'Windows' )
```

**Совет**

Для большей ясности давайте по возможности аргументам названия, схожие с названиями передаваемых им значений.

Существует возможность при определении функции указывать заранее значение аргументов по умолчанию. Оно станет использоваться, если в вызове функции аргументу не передано соответствующее значение, а в случае передачи будет перезаписано тем значением, которое указано в вызове:

```
def echo( user , lang , sys = 'Linux' ) :

print('User:' , user , 'Language:' , lang , 'Platform:' , sys )
```

Это означает, что вы можете вызывать функцию, передав ей меньше значений, чем задано при ее определении, и при этом вы будете исполь-

зовать значение аргументов, заданное по умолчанию, или же станете передавать столько же значений, сколько указано аргументов, и при этом перезаписывать значения тех аргументов, которые заданы по умолчанию.

1. Начните новую программу на Python с определения функции, принимающей три аргумента и выводящей на печать их значения.


args.py

```
def echo( user , lang , sys ) :

print( 'User:', user, 'Language:', lang, 'Platform:', sys )
```

2. Теперь вызовите функцию, передав строковые значения ее аргументам в том порядке, в котором они определены.

```
echo( 'Mike' , 'Python' , 'Windows' )
```

3. Вызовите функцию заново, передав ей строковые значения путем указания имен аргументов.

```
echo( lang = 'Python' , sys = 'Mac OS' , user = 'Anne' )
```

4. Теперь объявите еще одну функцию, принимающую два аргумента со значениями, заданными по умолчанию, которая также будет выводить свои аргументы.

```
def mirror( user = 'Carole' , lang = 'Python' ) :

print( '\nUser:' , user , 'Language:' , lang )
```

5. Теперь добавьте инструкции с вызовом второй функции, сначала используя значение аргументов по умолчанию, а затем переписывая их.

```
mirror()

mirror( lang = 'Java' )

mirror( user = 'Tony' )

mirror( 'Susan' , 'C++' )
```

6. Сохраните файл в рабочем каталоге, откройте командную строку и запустите вашу программу — вы увидите результат работы функции, выводящей значение своих аргументов.

# Возвращение значений

Как и встроенная в Python функция **str()**, которая возвращает числовое представление значения указанного в ней аргумента, пользовательская функция также может возвращать значения тому оператору, который ее вызвал. Это выполняется с помощью ключевого слова **return** с указанием после него возвращаемого значения. Например, чтобы возвратить значение суммы двух складываемых аргументов, пишем:

```
def sum( a , b ) :

return a + b
```

Возвращенный результат можно присвоить переменной с помощью инструкции вызова функции и впоследствии использовать в программе, например:

```
total = sum( 8 , 4 )

print( 'Eight Plus Four Is:' , total )
```

Либо он может быть использован непосредственно «на лету»:

```
print( 'Eight Plus Four Is:' , sum( 8 , 4 ) )
```

Как правило, инструкция **return** появляется в самом конце блока функции и возвращает окончательный результат всех вычислений, производимых инструкциями в теле функции.

**На заметку**

Вы можете указывать значение аргументов по умолчанию в определении функции.

Однако он может находиться и в начале блока инструкций определенной функции, прерывая исполнение всех последующих инструкций этого блока. Тогда исполнение программы в вызывающем функцию операторе немедленно останавливается. Опять же, инструкции **return** можно указать значение, которое она будет возвращать оператору, вызвавшему функцию, либо не указывать его, и в этом случае будет возвращаться значение **None**. Обычно данный прием используется, чтобы пропустить исполнение инструкций функции после того, как определенное проверочное условие не выполнилось. Как, например, в данном коде, где значение принимаемого аргумента меньше указанного числа:

```
def sum( a , b ) :

if a < 5 :

return

return a + b
```

В данном случае, когда первый переданный аргумент меньше пяти, функция будет возвращать значение **None**, а последняя инструкция не станет исполняться.

При выполнении в функции арифметических действий очень полезно проверять введенные пользователем значения на предмет наличия чисел с помощью встроенной функции `isdigit()`.

1. Начните новую программу на Python с инициализации переменной, задаваемой с помощью пользовательского ввода.

   ```
   num = input( 'Enter An Integer:' )
   ```

2. Затем добавьте определение функции, которая принимает единственный аргумент.

   ```
   def square( num ) :
   ```

3. Теперь добавьте в блок функции инструкцию для проверки переданного значения на предмет того, является ли оно числом, и для прерывания дальнейшего исполнения функции по результатам проверки.

   ```
   if not num.isdigit() :

   return 'Invalid Entry'
   ```

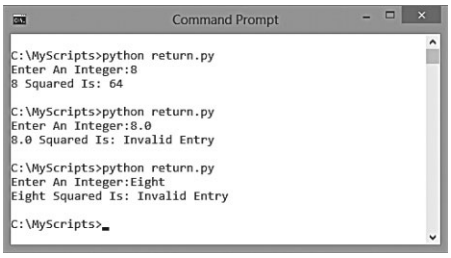
4. Затем добавьте инструкции для приведения типов с помощью функции `int`, а также возвращение результата возведения значения аргумента в квадрат.

   ```
   num = int( num )

   return num * num
   ```

5. Наконец добавьте инструкцию для вывода строки и возвращенного с помощью вызова функции значения.

   ```
   print( num , 'Squared Is:' , square( num ) )
   ```

6. Сохраните файл в рабочем каталоге, откройте командную строку и запустите вашу программу — вы увидите результат работы функции, отображающей возвращаемые значения.





return.py



**Внимание**

Помните, что пользовательский ввод работает со строками, так что для арифметических операций не забывайте использовать приведение типов к `int` или `float`.

# Использование обратного вызова

Как мы уже узнали, с помощью ключевого слова **def** можно создавать именованную функцию, а затем использовать ее имя для вызова в любом месте программы. Кроме того, именованная функция может возвращать вызвавшему ее оператору некоторое значение.

В языке Python существует возможность создавать функции без имени (анонимные), используя ключевое слово **lambda**. Анонимная функция может содержать только одно выражение, которое должно всегда возвращать значение.

В отличие от созданной с помощью ключевого слова **def** обычной функции **lambda**-функция возвращает объект, который разрешается присвоить переменной. Впоследствии она может быть использована для того, чтобы обратиться к функции (обратный вызов) в любом месте программы и исполнить блок выражений, которые содержит функция.

Таким образом, конструкция **lambda** позволяет программистам использовать альтернативный синтаксис для создания функции. Например:

```
def square( x ) :

return x ** 2
```

можно в более лаконичной форме записать так:

```
square = lambda x : x ** 2
```

В обоих случаях вызов **square(5)** возвратит результат 25, передав целое число 5 в качестве аргумента функции. Отметим, что аргумент после ключевого слова **lambda** стоит без скобок и выражение в данной функции не требует ключевого слова **return**, поскольку **lambda**-функция в любом случае возвращает значение.

**Совет**

Однострочные **lambda**-функции часто используются для программирования кнопок в программах с графическим пользовательским интерфейсом.

Поскольку ключевое слово **lambda** предлагает альтернативный способ для создания функции, он часто используется для встраивания функции в любое место кода. Например, обратные вызовы очень часто программируются в виде однострочных **lambda**-выражений, встроенных непосредственно в список аргументов вместо обычного определения при помощи **def** и вызова по имени:

```
def function_1 : исполняемые-выражения

def function_2 : исполняемые-выражения

callbacks = [ function_1 , function_2 ]
```

Такую запись можно выразить более кратко:

```
callbacks = [ lambda : выражение , lambda : выражение ]
```

1. Начните новую программу на Python, определив функции, которые возвращают переданный аргумент, возведенный в различную степень.


lambda.py

```
def function_1( x ) : return x ** 2

def function_2( x ) : return x ** 3

def function_3( x ) : return x ** 4
```

2. Добавьте инструкцию со списком обратных вызовов каждой из функций, указав ссылки на их имена.

```
callbacks = [ function_1 , function_2 , function_3 ]
```

3. Теперь выведите заголовок, а также результат передачи значений каждой из трех именованных функций.

```
print( '\nNamed Functions:' )

for function in callbacks : print( 'Result:' , function( 3 ) )
```

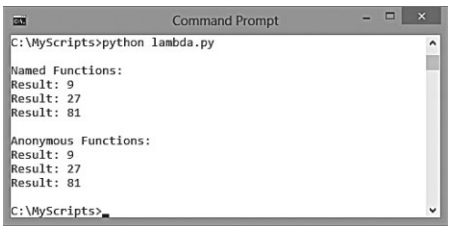
4. Затем добавьте инструкцию для создания списка вызовов анонимных функций, которые возвращают первый переданный аргумент, возведенный в различную степень.

```
callbacks = \

[ lambda x : x ** 2 , lambda x : x ** 3 , lambda x : x ** 4 ]
```

5. Наконец выведите заголовок, а также результат передачи значения каждой из трех анонимных функций.

```
print( '\nAnonymous Functions:' )

for function in callbacks : print( 'Result:' , function( 3 ) )
```

6. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу — вы увидите результаты, которые возвращают обычные и анонимные функции.




**Совет**

Определения функций, которые содержат только одну инструкцию, могут быть записаны в одну строку, как вы видите слева.

**Совет**

Для того чтобы продолжить код на следующей строке, вы можете использовать символ \.

# Добавление заполнителей

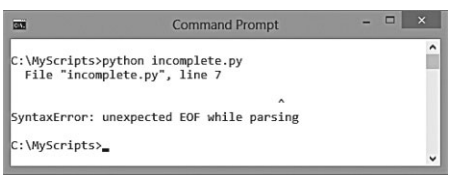
В языке Python существует ключевое слово **pass**, которое используется в качестве временной заглушки (заполнителя) и может быть добавлено в те места кода, куда впоследствии надо вставить какие-нибудь строки. Ключевое слово **pass** является «пустым», то есть не делает ничего. Данная возможность позволяет запускать еще не завершенные программы в целях отладки и исправления синтаксиса.


incomplete.py

1. Начните новую программу на Python, проинициализировав переменную логическим значением, добавив незавершенный условный оператор.

```
bool = True

if bool :

print( 'Python In Easy Steps' )

else :

# Сюда добавляем операторы.
```

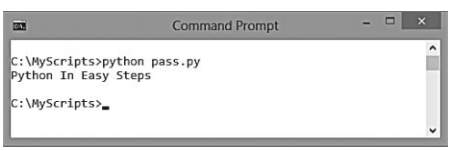
2. Сохраните файл, откройте командную строку, запустите программу — вы увидите, как интерпретатор выдает ошибку.





pass.py

3. Замените строку с комментарием ключевым словом **pass**, затем сохраните файл, запустите программу снова — вы увидите, что теперь интерпретатор не находит никакой ошибки.




При работе с циклами следует различать ключевые слова **pass** и **continue**. Первый из них позволяет интерпретатору обрабатывать все следующие за ним инструкции в текущей итерации цикла, в то время как второй пропускает все последующие инструкции и передает управление в новую итерацию.

1. Начните новую программу на Python с инициализации переменной строковым значением.

```
title = '\nPython In Easy Steps\n'
```

2. Теперь добавьте цикл, который выводит каждый символ строки.

```
for char in title : print( char , end = ' ' )
```

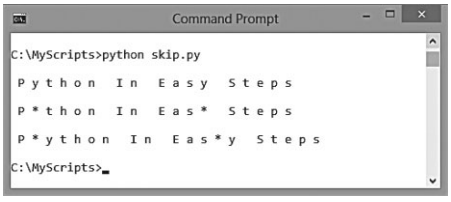
3. Затем добавьте еще один цикл, который выводит символ строки, но заменяет символ **y** и затем переходит к следующей итерации.

```
for char in title :

if char == 'y' :

print( '*' , end = ' ' )

continue

print( char , end = ' ' )
```

4. Наконец добавьте следующий цикл, который выводит каждый символ строки, но еще добавляет ***** перед каждым символом **y**.

```
for char in title :

if char == 'y' :

print( '*' , end = ' ' )

continue

print( char , end = ' ' )
```

5. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу — вы увидите различный вывод как результат применения ключевых слов **pass** и **continue** в цикле.





skip.py



**На заметку**

В циклах ключевое слово **continue** выполняет переход на следующую итерацию, в то время как ключевое слово **pass** передает управление следующей инструкции той же итерации.

# Генераторы в Python

Когда вызывается какая-либо функция, то выполняются инструкции, содержащиеся в ее определении, а также может быть возвращено любое значение переменной, указанной после ключевого слово **return**. После завершения работы функции управление передается вызвавшему ее оператору, а состояние функции при этом не сохраняется. При следующем вызове функции она будет обрабатывать те же инструкции от начала до конца еще раз.

В языке Python существует специальная функция-генератор, которая возвращает объект, а не значение. При этом она сохраняет состояние своего последнего вызова и при следующем вызове продолжает работу с той же точки.

Функции-генераторы определяются так же, как и обычные функции, но они дополнительно содержат «генераторную» инструкцию. Она начинается с ключевого слова **yield** и определяет объект-генератор, который возвращается оператору, вызвавшему функцию. Когда генераторная инструкция исполняется, состояние объекта-генератора «замораживается» и сохраняется. Объект, возвращаемый генераторной инструкцией, может быть присвоен переменной. С помощью встроенной функции **next()** можно, передав ей имя этой переменной, продолжить выполнение функции с той самой точки заморозки.

Повторный вызов генератора с помощью функции **next()** продолжает исполнение функции до тех пор, пока не вызовется исключение. Его можно избежать, поместив генераторную инструкцию внутрь бесконечного цикла. Например, чтобы сгенерировать значения с приращением на каждом новом вызове, запишем:

```
def incrementer() :
i = 1
while True :
yield i
i += 1
inc = incrementer()
print( next( inc ) )
print( next( inc ) )
print( next( inc ) )
```

Эти последовательные вызовы функции выведут целые значения **1**, **2** и **3**.

**Внимание**

При изменении условного выражения в данном цикле на `while i < 3` третий вызов функции приведет к генерации исключения `StopIteration`.

Наиболее эффективным будет размещать объект-генератор в цикле, осуществляющем последовательные итерации по значениям.

1. Начните новую программу на Python, определив функцию, которая начинается с инициализации двух переменных одним целым числом.


yield.py

```
def fibonacci_generator() :

a = b = 1
```

2. Теперь добавьте в тело функции бесконечный цикл (не забывая про отступ) для генерации суммы двух предыдущих значений.

```
while True :

yield a

a , b = b , a + b
```

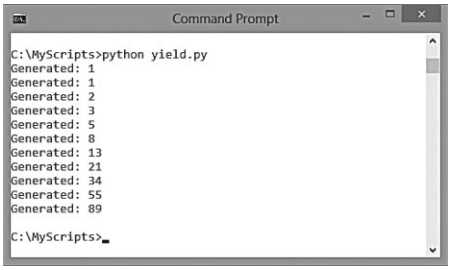
3. Присвойте переменной возвращенный объект-генератор.

```
fib = fibonacci_generator()
```

4. Наконец добавьте цикл для последовательного вызова функции-генератора и вывода ее значения на каждой итерации.

```
for i in fib :

if i > 100 :

break

else :

print( 'Generated:' , i )
```

5. Сохраните файл в рабочем каталоге, откройте командную строку и запустите вашу программу — вы увидите сгенерированные в цикле возрастающие значения.




**На заметку**

Здесь переменные инициализируются одновременно одним значением.

**Совет**

Для того чтобы проверить, является ли объект генератором, можно также использовать встроенную функцию `type()`.

# Обработка исключений

Те части программы на языке Python, в которых возможно появление ошибок, например работающие с пользовательским вводом, разрешается заключать в специальные блоки **try-except**, с помощью которых можно обрабатывать «исключительные ситуации». Инструкции, которые способны вызвать ошибки при выполнении, группируются в блок **try**, а те команды, которым предстоит обрабатывать эти ошибки, — в последующий блок **except**. После него может стоять необязательный блок **finally**, где содержатся инструкции, выполняющиеся после того, как все исключения будут обработаны.

**Совет**

Более подробную информацию о встроенных исключениях смотрите на странице **docs.python.org/3/library/exceptions.html**.

В языке Python встречается много различных типов встроенных исключений, таких как, например, **NameError**, которые происходят, если имя переменной не найдено; исключение **IndexError**, вызываемое при попытке обращения к несуществующему индексу элемента списка; а также **ValueError**, которое появляется, когда во встроенную операцию или функцию передается аргумент, имеющий несоответствующее значение.

Каждое исключение возвращает сообщение с описательной информацией, которое можно использовать, присвоив его какой-нибудь переменной с помощью ключевого слова **as**, чтобы отобразить природу происхождения этого исключения.




try.py

1. Начните новую программу на Python, проинициализировав переменную со строковым значением.

```
title = 'Python In Easy Steps'
```

2. Теперь добавьте блок инструкции **try**, в котором производится попытка вывести значение переменной, но ее имя, указанное в выражении, некорректно.

```
try :
print( titel )
```

3. Теперь добавьте блок инструкции **except** для вывода сообщения об ошибке.

```
except NameError as msg :
print( msg )
```

4. Сохраните файл в рабочем каталоге, откройте командную строку и запустите вашу программу — вы увидите результат обработки ошибки.

Можно обрабатывать множественные исключения, для этого внутри блока **except** в скобках нужно указать через запятую типы этих исключений:

```
except ( NameError , IndexError ) as msg :

print( msg )
```

Вы можете также создавать пользовательские типы исключений. Для этого используется ключевое слово **raise**, после которого указывается тип исключения и описательное сообщение, придуманное пользователем.

1. Начните новую программу на Python с инициализации целочисленной переменной.


raise.py

```
day = 32
```

2. Теперь добавьте блок инструкции **try**, в котором проверяется значение переменной, затем указывается исключение и соответствующее сообщение для пользователя.

```
try :

if day > 31 :

raise ValueError( 'Invalid Day Number' )

# Сюда добавляем операторы.
```

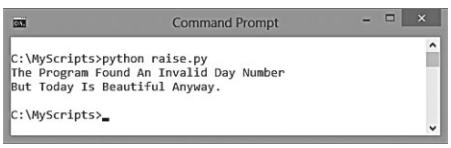
3. Затем добавьте блок инструкции **except** для вывода сообщения в случае появления исключения ValueError.

```
except ValueError as msg :

print( 'The Program found An' , msg )
```

4. Теперь добавьте блок инструкции **finally** для вывода сообщения после успешной обработки исключения.

```
finally :

print( 'But Today Is Beautiful Anyway.' )
```

5. Сохраните файл в рабочем каталоге, откройте командную строку и запустите вашу программу — вы увидите результат обработки вызываемого исключения.



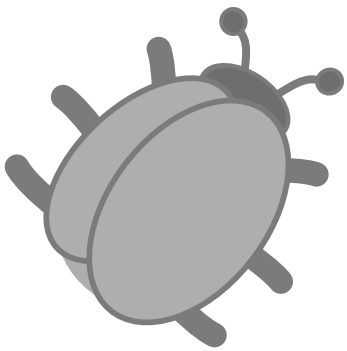

**На заметку**

Инструкции в блоке **try** исполняются до тех пор, пока не вызовется исключение.

# Отладка с помощью инструкции assert

Для поиска ошибок в вашем коде при отладке программы полезно использовать такой прием, как комментирование, когда в начало строки добавляется символ **#**, и таким образом одна или несколько строк превращаются в комментарии. В результате интерпретатор Python опускает выполнение этих закомментированных строк, локализуя проблему в программе. Например, когда вы подозреваете, что есть какая-то ошибка в присваивании значения переменной, то ее можно исключить следующим образом:

```
# elem = elem / 2
```



Если после этого программа выполняется без ошибок, то очевидно, что проблема находилась в закомментированной строке. Еще одним полезным средством для отладки является использование инструкции **assert**. Она проверяет указанное тестовое выражение на предмет равенства значениям **True** или **False** и выдает ошибку **AssertionError**, если проверка не пройдена. Она может также включать описательное сообщение, имея следующий синтаксис:

**assert** *проверочное-выражение* , *описательное-сообщение*

Когда проверка выражения не выполнена, интерпретатор сообщает об ошибке **AssertionError** и прерывает исполнение программы. В случае же успешной проверки, инструкция **assert** не делает ничего, и исполнение программы продолжается.

Использование механизма проверки с помощью инструкции **assert** очень полезно для документирования ваших программ.

**Совет**

Используя опцию **-O** при запуске программы, вы можете отключить проверку интерпретатором инструкции **assert**. Например, **python -O assert.py**.

## Assert или исключение

На первый взгляд работа инструкции **assert** может быть похожа на обработку исключений, но важно помнить об их различиях.

- **Исключения** предлагают способ для обработки ошибок, которые могут происходить на этапе исполнения.
- Инструмент **AssertionErrors** обеспечивает программистов средствами для нахождения ошибок во время работы над программой.

Инструкции **assert**, как правило, удаляются из окончательных версий программы после того, как отладка завершена, в то время как инструкции **except** остаются в программе и обрабатывают ошибки исполнения.

1. Начните новую программу на Python, проинициализировав список, состоящий из нескольких строковых переменных.

```
chars = [ 'Alpha' , 'Beta' , 'Gamma' , 'Delta' , 'Epsilon' ]
```

2. Затем определите функцию, принимающую единственный аргумент.

```
def display( elem ) :
```

3. Теперь добавьте инструкции в тело функции для проверки переданного значения аргумента на целое число и отображения элемента списка с соответствующим порядковым номером. (Не забывайте про отступы в теле функции).

```
assert type( elem ) is int , 'Argument Must Be Integer!'

print( 'List Element' , elem , '=' , chars[ elem ] )
```

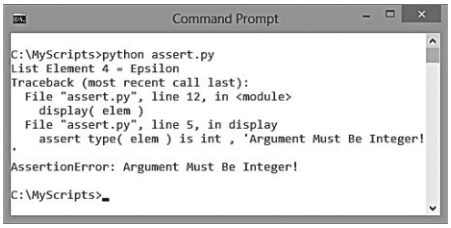
4. Затем проинициализируйте переменную целочисленным значением и вызовите функцию, передав ей значение этой переменной в качестве аргумента.

```
elem = 4

display( elem )
```

5. Теперь измените значение переменной и вызовите функцию еще раз, передав ей в качестве аргумента это новое значение.

```
elem = elem / 2

display( elem )
```

6. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу — вы увидите ваше сообщение, которое сопровождает ошибку **AssertionError**.





**assert.py**



**На заметку**

В данном случае **AssertionError** происходит в результате того, что операция деления возвращает не целочисленную величину, а значение с плавающей точкой.

# Заключение

- Функция определяется с помощью ключевого слова `def` и содержит блок инструкций, выделенных отступом, которые исполняются при вызове данной функции.
- К переменным с глобальной областью видимости можно обращаться из любого места программы, в то время как к локальным переменным — только внутри тех функций, где они объявлены.
- При определении функции ее аргументы объявляются в виде списка, разделенного запятой, находящегося внутри скобок.
- При вызове функции ей должны передаваться параметры для каждого ее аргумента, кроме случаев, когда в качестве аргументов используются значения по умолчанию, описанные при объявлении функции.
- Функция может включать в себя инструкцию `return`, которая возвращает значение оператору, вызвавшему функцию.
- С помощью ключевого слова `lambda` можно создавать анонимные функции, содержащие одно выражение и возвращающие объект функции.
- Очень часто используются функции обратного вызова, представляющие собой `lambda`-выражения, встроенные непосредственно в список аргументов в вызывающем функцию операторе.
- С помощью ключевого слова `pass` можно добавлять заполнители (заглушки) в места программы, куда впоследствии планируется поставить какую-либо инструкцию.
- Если в блоке функции появляется инструкция, использующая ключевое слово `yield`, создается генераторная функция.
- Генераторная функция сохраняет свое состояние с момента последнего вызова, а также возвращает объект-генератор вызвавшему ее оператору.
- Для продолжения исполнения генераторной функции с того места, где она была заморожена, используется встроенная функция `next()`.
- Ошибки или исключительные ситуации, которые появляются в процессе запуска программы, могут быть обработаны с помощью заключения инструкций в блоки `try-except`.
- Можно использовать необязательную инструкцию `finally` для определения инструкций, которые будут исполняться после обработки исключений.
- С помощью ключевого слова `assert` можно добавлять в программу отладочный код, который станет сообщать об ошибках разработки.

# 5 Импорт модулей

*В этой главе описывается, как использовать модули Python в ваших программах.*

- **Хранение функций**
- **Принадлежность имен функций**
- **Системные запросы**
- **Математические операции**
- **Вычисления с десятичными дробями**
- **Работа со временем**
- **Запуск таймера**
- **Шаблоны соответствий**
- **Заключение**

# Хранение функций

Когда вы один раз определили какую-либо функцию, ее можно сохранить в одном или нескольких отдельных файлах, а затем использовать в других программах без дополнительного копирования в каждую из них. Такой файл, хранящий определение функции, называется **модулем**, а именем модуля является имя соответствующего файла без расширения *.py*.

Функции, хранящиеся в модуле, можно сделать доступными в любой другой программе на Python с помощью так называемого **импортирования** модуля, используя ключевое слово `import` и стоящее после него имя нужного модуля. Обычно инструкции, содержащие импорт модулей, ставят в начало программы, хотя это и необязательно.

Любую импортированную в программу функцию затем можно вызвать, используя **суффиксную** (или **точечную**) запись, а именно *`имя-модуля.имя-функции`*. Например, чтобы вызвать функцию `steps` из импортированного модуля с именем `ineasy`, достаточно набрать `ineasy.steps()`.

Когда хранящиеся в модуле функции включают в себя какие-либо аргументы, очень часто полезным будет назначать этим аргументам значения по умолчанию при определении функции. Это сделает функцию более универсальной, так как при вызове ее из какого-либо места программы указание аргументов становится необязательным.


`cat.py`

1. Начните новый модуль на Python, определив функцию, в которой для вывода используется значение аргумента по умолчанию.

```
def purr( pet = 'A Cat' ) :

print( pet , 'Says MEOW!' )
```

2. Теперь добавьте еще два определения функций, в которых также задаются значения их аргументов по умолчанию для использования при выводе.

```
def lick( pet = 'A Cat' ) :

print( pet , 'Drinks Milk' )

def nap( pet = 'A Cat' ) :

print( pet , 'Sleeps By The Fire' )
```

3. Сохраните файл под именем *cat.py*, таким образом, ваш модуль будет называться `cat`.

4. Начните новую программу на Python, в которой в самом начале поставьте инструкцию импорта модуля **cat**.

kitty.py

```
import cat
```

5. Теперь добавьте вызовы всех трех функций без подстановки аргумента.

```
cat.purr()

cat.lick()

cat.nap()
```

6. Затем добавьте еще по одному вызову функций, передав при этом каждой из них по одному аргументу, и сохраните файл.

```
cat.purr( 'Kitty' )

cat.lick( 'Kitty' )

cat.nap( 'Kitty' )
```

7. Начните другую программу, сделав доступными функции модуля **cat** еще раз.

tiger.py

```
import cat
```

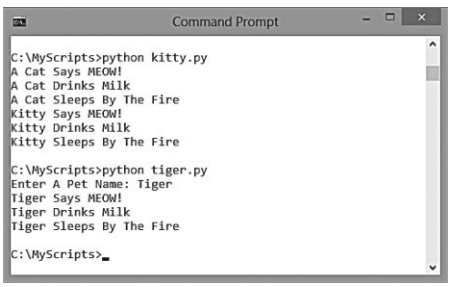
8. Теперь запросите пользователя ввести имя для переменной, которая перезапишет значение аргументов функции по умолчанию.

```
pet = input( 'Enter A Pet Name: ' )
```

9. Наконец, вызовите все три функции еще раз, передав в качестве аргумента значение переменной, определенное пользователем.

```
cat.purr( pet )

cat.lick( pet )

cat.nap( pet )
```

10. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите эти программы. Вы увидите результат вывода с использованием данных из импортированных модулей.




**На заметку**

При импорте модуля вы можете создавать его псевдоним, используя ключевые слова `import as`. Например, команда `import cat as tom` позволит вам использовать все функции модуля `cat` под именем `tom`.

# Принадлежность имен функций

Каждый модуль в языке Python и каждая программа имеют собственную **таблицу символов**, которая используется для всех функций, определенных в данном модуле или программе. Это позволяет избегать конфликтов в том случае, если в одну программу импортированы два модуля, в которых имеются функции с одинаковыми именами.

**На заметку**

При импорте отдельных имен функций имена модулей не импортируются — поэтому их нельзя использовать в качестве префикса.

Когда вы импортируете в программу какой-то модуль с помощью инструкции `import`, то таблица символов этого модуля не добавляется в текущую таблицу символов программы — туда помещается только имя импортируемого модуля. Поэтому, когда вам нужно вызывать функции из этого модуля, следует использовать в имени функций префиксное имя данного модуля. Например, если в программе вы импортируете функцию `steps` из модуля с именем `ineasy`, а также еще одну функцию `steps` из другого модуля под именем `dance`, то затем без особых проблем вы можете вызывать эти функции с помощью записи `ineasy.steps()` и `dance.steps()`.

Как правило, во избежание конфликтов предпочтительнее импортировать имена модулей и вызывать принадлежащие им функции так, как было сказано выше, с помощью записи через префикс имени модуля. Но вы можете также импортировать отдельные имена функций с помощью инструкции `from import`. Имя модуля при этом указывается после ключевого слова `from`, а имена импортируемых функций — в виде разделенного запятой списка после ключевого слова `import`. Еще существует способ импортирования всех имен функций в таблицу символов программы. Для этого используется шаблон `*` после ключевого слова `import`. В этом случае все импортированные функции можно будет вызывать без префикса имени модуля.


dog.py

1. Начните новый модуль на Python с определения функции, в которой задано значение ее аргумента по умолчанию.

```
def bark( pet = 'A Dog' ) :

print( pet , 'Says WOOF!' )
```

2. Теперь добавьте еще два определения функций, в которых также заданы значения по умолчанию для их аргументов.

```
def lick( pet = 'A Dog' ) :

print( pet , 'Drinks water' )

def nap( pet = 'A Dog' ) :

print( pet , ' Sleeps In The Sun' )
```

3. Сохраните файл под именем *dog.py*, таким образом, модуль будет иметь имя `dog`.

4. Начните новую программу на Python, включив в нее инструкцию, которая сделает доступными функции модуля `dog`.

```
from dog import bark , lick , nap
```

5. Теперь вызовите каждую из функций без подстановки аргументов.

```
bark()

lick()

nap()
```

6. Вызовите все функции заново, передав начальное значение аргумента для каждой, и сохраните файл.

```
bark( 'Pooch' )

lick( 'Pooch' )

nap( 'Pooch' )
```

7. Начните другую программу на Python, сделав доступными функции модуля `dog`.

```
from dog import *
```

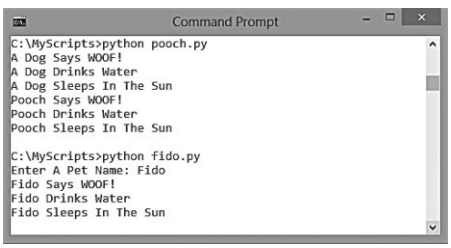
8. Теперь добавьте запрос пользователю ввести имя переменной, которую будете использовать для перезаписи значения аргумента по умолчанию.

```
pet = input( 'Enter A Pet Name: ' )
```

9. Наконец вызовите все функции заново, передав каждому из аргументов значение, определенное пользователем.

```
bark( pet )

lick( pet )

nap( pet )
```

10. Сохраните файл в вашем рабочем каталоге, затем откройте командную строку и запустите эти программы — вы увидите результаты работы импортированных функций.





pooch.py


fido.py



**Совет**

В больших программах вы можете импортировать одни модули в другие, организуя модульную иерархию.

# Системные запросы

В Python включены модули **sys** и **keyword**, которые применяются для организации доступа к некоторым переменным и функциям, взаимодействующим с самим интерпретатором Python. Модуль **keyword** содержит список всех ключевых слов языка Python, содержащихся в его атрибуте **kwlist**, а также обеспечивает метод **iskeyword()** для определения, является ли слово зарезервированным.

Вы можете изучить большинство функций модуля **sys** — также как всего интерпретатора Python — используя встроенную систему помощи интерактивного режима. Для этого наберите **help()** в строке подсказки после символов **>>>** для запуска системы помощи, а затем в строке подсказки **help >** наберите **sys**.

Возможно, что из большинства атрибутов модуля **sys** вам могут быть полезны те, что содержат номер интерпретатора, расположение его в вашей сети, а также список каталогов, где интерпретатор осуществляет поиск.




**system.py**

1. Начните новую программу на Python, импортировав модули **sys** и **keyword**, сделав доступными их функции.

```
import sys , keyword
```

2. Затем добавьте инструкцию для вывода информации о версии установленного интерпретатора Python.

```
print( 'Python Version:' , sys.version )
```

3. Теперь добавьте инструкцию, которая выводит действительное расположение интерпретатора Python в вашей операционной системе.

```
print( 'Python Interpreter Location:' , sys.executable )
```

4. Добавьте инструкции для вывода списка всех каталогов, среди которых интерпретатор Python производит поиск исполняемых файлов.

```
print( 'Python Module Search Path: ' )

for dir in sys.path :

print( dir )
```

5. Наконец, добавьте инструкции для вывода списка ключевых слов языка Python.

```
print( 'Python Keywords: ' )

for word in keyword.kwlist :

print( word )
```

6. Сохраните файл в вашем рабочем каталоге, затем откройте командную строку и запустите программу — вы увидите подробную информацию об установленной в вашей системе версии интерпретатора Python.


**На заметку**

Первым элементом в списке каталогов поиска является ваш текущий каталог — таким образом, интерпретатор Python автоматически найдет любой файл внутри этого каталога либо в его подкаталогах.

**Совет**

Потратьте некоторое время для изучения системы помощи в интерактивном режиме Python.

# Математические операции

В языке Python существует модуль **math**, методы которого вы можете использовать для работы с математическими операциями.

Например, методы **math.ceil()** и **math.floor()** позволяют осуществлять округление значений с плавающей точкой, указанных в качестве параметров этим методам, до ближайшего целого — **math.ceil()** округляет вверх, **math.floor()** — вниз. Несмотря на то что данное значение имеет нулевую дробную часть, оно на самом деле имеет тип **float**, а не **int**.

**На заметку**

Используя встроенные функции **float()** и **str()**, вы можете приводить целочисленные значения к типам **float** и **string**.

Метод **math.pow()**, принимающий два аргумента, используется для возведения одного аргумента в степень другого. Метод **math.sqrt()**, требующий единственный аргумент, возвращает квадратный корень из указанной в его скобках величины. Оба этих метода возвращают числовое значение типа **float**.

Модули также могут осуществлять вычисление тригонометрических функций, например **math.sin()**, **math.cosin()** и **math.tan()**.

Для того чтобы работать с псевдослучайными числами, вы можете также импортировать в программу модуль **random** языка Python.

Метод **random.random()** генерирует одно число с плавающей точкой от нуля до 1.0. Возможно, больший интерес будет представлять метод **random.sample()**, который генерирует список элементов, случайно выбранных из последовательности. Этот метод требует два аргумента для указания последовательности, из которой выбирать, а также размера генерируемого списка. В качестве первого аргумента для метода **random.sample()** можно использовать функцию **range()**, которая, как вы помните, возвращает последовательность чисел. Таким образом, из этой последовательности метод **random.sample()** будет создавать список с неповторяющимися элементами.


**maths.py**

1. Начните новую программу на Python, импортировав модули **math** и **random**, чтобы сделать доступными их функции.

```
import math , random
```

2. Теперь добавьте инструкции для вывода двух округленных значений.

```
print( 'Rounded Up 9.5:' , math.ceil( 9.5 ) )

print( 'Rounded Down 9.5:' , math.floor( 9.5 ) )
```

3. Затем добавьте инструкцию с инициализацией целочисленной переменной.

```
num = 4
```

4. Добавьте инструкции для вывода значения квадрата, а также квадратного корня из значения заданной переменной.

```
print( num , 'Squared:' , math.pow( num , 2 ) )

print( num , 'Square Root:' , math.sqrt( num ) )
```

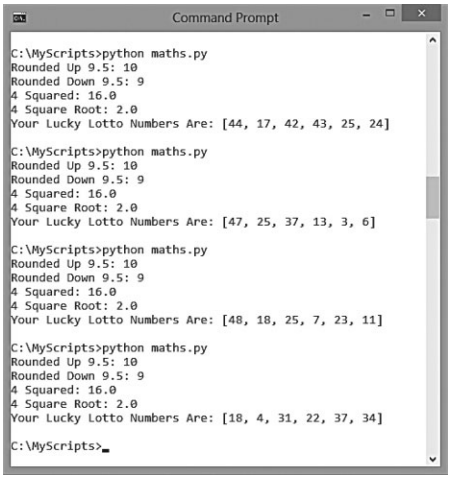
5. Добавьте инструкцию, которая генерирует случайный список из шести уникальных чисел в диапазоне от 1 до 49.

```
nums = random.sample( range( 1, 49 ) , 6 )
```

6. Наконец добавьте инструкцию для вывода случайного списка.

```
print( 'Your Lucky Lotto Numbers Are:' , nums )
```

7. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу. Вы увидите результат работы функций модуля `math`.




**На заметку**

Все используемые здесь методы возвращают числа с плавающей точкой (типа `float`).

**Совет**

Список, генерируемый методом `random.sample()` создает только копию первоначальной последовательности, но не работает с ее элементами.

# Вычисления с десятичными дробями

При использовании в программах арифметических вычислений с числами с плавающей точкой могут возникать погрешности. Это обусловлено округлением десятичных дробей.


inaccurate.py

1. Начните новую программу на Python с инициализации двух переменных с плавающей точкой.

```
item = 0.70

rate = 1.05
```

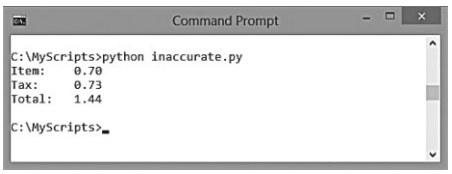
2. Теперь проинициализируйте еще две переменные с помощью арифметических операций над первыми двумя.

```
tax = item * rate

total = item + tax
```

3. Затем добавьте инструкции для форматированного вывода значений переменных, использующего два знака после запятой для десятичных дробей.

```
print( 'Item:\t' , '%.2f' % item )

print( 'Tax:\t' , '%.2f' % tax )

print( 'Total:\t' , '%.2f' % total )
```

4. Сохраните файл в рабочем каталоге, откройте командную строку и запустите программу. Вы увидите вывод результата сложения с погрешностью.





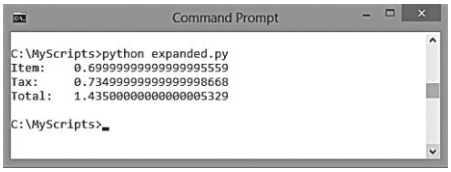
expanded.py

5. Для понимания этой проблемы отредактируйте все три инструкции таким образом, чтобы выводить не два, а двадцать знаков после запятой. Затем запустите измененную программу.

```
print( 'Item:\t' , '%.20f' % item )

print( 'Tax:\t' , '%.20f' % tax )

print( 'Total:\t' , '%.20f' % total )
```

**Совет**

В данном случае форматирование значений переменных использует метод подстановки строки для отображения количества десятичных знаков — это описано более подробно в следующей главе.




Очевидно, что значение **tax** немного меньше, чем 0,735, поэтому оно округлено до 0,73. А значение переменной **total** чуть выше, чем 1,435, поэтому было округлено до 1,44, и это предопределило результат сложения.

Чтобы избежать таких ошибок при выполнении арифметических операций над числами с плавающими точками, в языке Python используется модуль **decimal**. С помощью находящегося в нем объекта **Decimal()** значения представляются более точно.

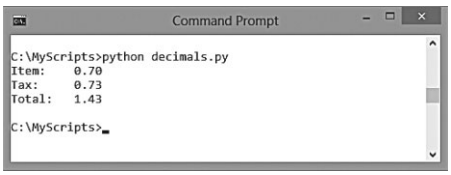
6. Добавьте в программу инструкцию импорта для подключения модуля **decimal**, сделав доступными все его функции.

```
from decimal import *
```

7. Теперь отредактируйте первые две инструкции присваивания следующим образом:

```
item = Decimal( 0.70 )

rate = Decimal( 1.05 )
```

8. Сохраните изменения, запустите отредактированную программу — вы увидите, что значения обеих переменных (**tax** и **total**) будут округлены вниз, и результат сложения окажется верным.




**На заметку**

Эта проблема существует не только в языке Python — в Java, например, есть класс **BigDecimal**, который решает ее примерно таким же способом, как модуль **Decimal** в Python.


**decimals.py**

**Совет**

Везде, где требуется большая точность в вычислениях, например в финансовых расчетах, используйте объект **Decimal()**.

# Работа со временем

Для того чтобы в программах на Python работать с системными временем и датой, может быть подключен модуль `datetime`. Он содержит объект `datetime` с атрибутами `year`, `month`, `day`, `hour`, `minute`, `second`, `microsecond`.

Объект `datetime` содержит метод `today()`, который присваивает атрибутам объекта значение текущей даты и времени и возвращает их в виде кортежа. Он также содержит метод `getattr()`, который требует два аргумента, указывающих имя объекта и атрибут, который нужно получить. Альтернативным способом для доступа к атрибуту может служить точечная запись вида `datetime.year`.

Все значения в объекте `datetime` хранятся в виде числовых величин, но могут быть преобразованы в их текстовые эквиваленты с помощью метода `strftime()`. Данный метод требует передачи единственного строкового аргумента (так называемой **директивы**), указывающего, какую часть кортежа и в каком формате возвратить. Список возможных директив представлен в таблице ниже.

**Совет**

Поскольку объект `datetime` находится в модуле с тем же именем, простое импортирование модуля означает, что к нему нужно будет обращаться так: `datetime.datetime`. Использование инструкции `from datetime import *` позволит упростить запись до `datetime`.

| Директива | Возвращаемое значение |
|---|---|
| `%A` | Полное название дня недели (`%a` — для сокращенного) |
| `%B` | Полное название месяца (`%b` — для сокращенного) |
| `%c` | Дата и время (локальные) |
| `%d` | Порядковый номер дня в месяце от 1 до 31 |
| `%f` | Количество микросекунд от 0 до 999999 |
| `%H` | Десятичное представление часа от 0 до 23 (для 24-часового вида) |
| `%I` | Десятичное представление часа от 1 до 12 (для 12-часового вида) |
| `%j` | Порядковый номер дня в году от 0 до 366 |
| `%m` | Порядковый номер месяца от 1 до 12 |
| `%M` | Десятичное представление минут от 0 до 59 |
| `%p` | Обозначение AM (до полудня) или PM (после полудня) |
| `%S` | Десятичное представление секунд от 0 до 59 |
| `%w` | Порядковый номер дня в неделе от 0 (воскресенье) до 6 |
| `%W` | Порядковый номер недели в году от 0 до 53 |
| `%X` | Локальное время (`%x` — локальная дата) |
| `%Y` | Полное десятичное представление года от 0001 до 9999 (`%y` — для краткого представления от 00 до 99) |
| `%z` | Смещение часового пояса от UTC в виде +ЧЧММ или –ЧЧММ |
| `%Z` | Название часового пояса |

1. Начните новую программу на Python, импортировав модуль **datetime**, чтобы сделать доступными его функции.

```
from datetime import *
```

2. Теперь создайте объект **datetime** и присвойте его атрибутам текущее значение времени, затем выведите содержимое.

```
today = datetime.today()

print( 'Today Is:' , today )
```

3. Добавьте цикл для вывода значения каждого атрибута отдельно.

```
for attr in \

['year','month','day','hour','minute','second','microsecond' ] :

print( attr , ':\t' , getattr( today , attr ) )
```

4. Теперь добавьте инструкцию для вывода значения времени, используя точечную запись.

```
print( ' Time:' , today.hour , ':' , today.minute , sep = '')
```

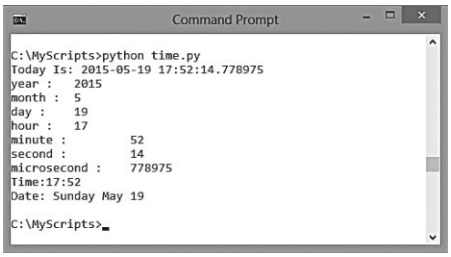
5. Затем присвойте переменным **day** и **month** форматированное значение.

```
day = today.strftime( '%A' )

month = today.strftime( '%B' )
```

6. Наконец, добавьте инструкцию для вывода форматированной даты.

```
print( 'Date:' , day , month , today.day )
```

7. Сохраните файл в вашем рабочем каталоге, затем откройте командную строку и запустите программу — вы увидите результат вывода значений даты и времени.





today.py

**Совет**

Обратите внимание, что при написании данного цикла использовался символ \, который позволяет переносить инструкции на следующую строку.

**Совет**

Для присваивания новых значений атрибутам объекта **datetime** можно использовать метод **replace()**, например **today = today.replace(year=2015)**.

# Запуск таймера

Если знать время начала и конца какого-либо события, то можно вычислить его длительность по разнице двух значений времени. Для того чтобы использовать различные функции, связанные с системным временем, в программу на Python можно импортировать модуль `time`.

Обычно текущее системное время рассчитывается как количество секунд, прошедших с 00 часов 1 января 1970 года (так называемая эра Unix). Если вы вызовете метод `time()` модуля `time`, то он возвратит текущее время в секундах, прошедшее с вышеуказанного момента начала эры Unix, в виде числа с плавающей точкой.

Значение, возвращаемое методом `time()`, может быть преобразовано в объект `struct_time` при помощи методов `gmtime()` и `localtime()`. Данный объект содержит атрибуты `tm_year`, `tm_mon`, `tm_mday`, `tm_hour`, `tm_ min`, `tm_sec`, `tm_wday`, `tm_yday`, `tm_yday` и `tm_isdst`, к которым можно обратиться, используя точечную запись. Например, `struct.tm_wday`.

Все, что находится в объекте `struct_time`, хранится в числовом виде и может быть преобразовано в текстовый эквивалент при помощи метода `strftime()`. Данный метод требует наличия аргумента, который представляет собой директиву, определяющую формат, после которой следует имя объекта. Возможные директивы для объекта `datetime` перечислены в таблице чуть ранее в этой главе. Например, для дня недели запись будет выглядеть так: `strftime( '%A' , struct )`.

Модуль `time` предоставляет очень полезный метод `sleep()`, который можно использовать для того, чтобы организовывать паузы в выполнении программы. Аргумент этого метода определяет количество времени в секундах, на которое необходимо сделать задержку.

**На заметку**

Метод `gmtime()` преобразует время, выраженное в секундах, с начала эры Unix в объект `struct_time` с флагом DST, всегда равным нулю (флаг дневного времени), в то время как `localtime()` выполняет преобразование в объект `struct_time` в соответствии с вашим локальным временем.


`timer.py`

1. Начните новую программу на Python, сделав доступными функции импортированного модуля `time`.

```
from time import *
```

2. Затем проинициализируйте переменную, относящуюся к числовому типу с плавающей точкой, и задайте ей значение времени, прошедшего с начала эры Unix.

```
start_timer = time()
```

3. Теперь добавьте инструкцию для создания объекта `struct_time`, используя эту переменную.

```
struct = localtime( start_timer )
```

4. После чего выведите сообщение о том, что с текущего момента времени стартует таймер обратного отсчета.

```
print( '\nStarting Countdown At:' , strftime( '%X' , struct ) )
```

5. Добавьте цикл с инициализацией и выводом переменной-счетчика, который на каждой итерации уменьшается на единицу, и односекундной паузой.

```
i = 10

while i > -1 :

print( i )

i -= 1

sleep( 1 )
```

6. Теперь проинициализируйте вторую переменную, являющуюся числом с плавающей точкой, которая представляет время, прошедшее с начала эры Unix.

```
end_timer = time()
```

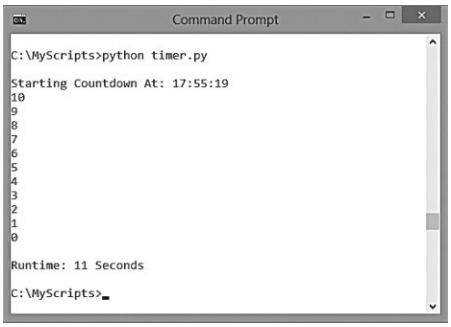
7. Затем проинициализируйте переменную, которая будет представлять результат округления разности двух значений времени.

```
difference = round( end_timer - start_timer )
```

8. Наконец добавьте инструкцию для вывода времени выполнения цикла обратного отсчета.

```
print( '\nRuntime:' , difference , 'Seconds' )
```

9. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу — вы увидите числа обратного отсчета, выводящиеся через 1 секунду, и общее время выполнения.




**Совет**

В качестве аргумента методу `sleep()` можно выбирать число с плавающей точкой для указания более точного времени паузы.

**Внимание**

Не путайте метод `time.strftime()`, использовавшийся здесь, с методом `datetime.strftime()` из предыдущего примера.

# Шаблоны соответствий

В языке Python существует мощное средство для поиска и замены текста при помощи шаблонов — так называемые **регулярные выражения**. Для их использования необходимо импортировать модуль `re`.

**Внимание**

Тема регулярных выражений достаточно обширна и выходит за рамки данной книги — здесь мы только кратко знакомимся с ними.

Шаблон регулярного выражения может состоять целиком из **символьных литералов**, описывающих строку символов, соответствующих какому-либо тексту. Например, регулярное выражение `wind` найдет соответствие в строке `windows`. В общем случае шаблон регулярного выражения состоит из набора символьных литералов, а также следующих метасимволов.

| Метасимвол | Соответствие | Пример |
|---|---|---|
| . | Любой символ | `py..on` |
| ^ | Начало строки | `^py` |
| $ | Конец строки | `...on$` |
| * | Повторение фрагмента ноль или более раз | `py*` |
| + | Повторение фрагмента один или более раз | `py+` |
| ? | Фрагмент либо присутствует, либо отсутствует | `py?` |
| { } | Множественное повторение | `a{ 3 }` |
| [ ] | Класс символов | `[ a-z ]` |
| \ | Специальная последовательность | `\s` |
| \| | Фрагмент слева или фрагмент справа | `a \| b` |
| ( ) | Группировка выражений | `( ... )` |

**На заметку**

Диапазон символов `[a-z]` соответствует только строчным латинским буквам, а диапазон `[a-z0-9]` включает еще и цифры.

Комбинация литералов и метасимволов, образующая регулярное выражение, при помощи метода `re.compile()` компилируется в объект шаблона. У этого объекта существуют методы для различных операций, например при помощи метода `match()` можно проверять соответствие шаблону строки, переданной методу в качестве аргумента.

Метод `match()` возвращает значение `None`, если совпадений с шаблоном не найдено, а в случае успешного поиска возвращается объект, содержащий информацию о соответствии строки шаблону.

Полученный в результате объект содержит методы `start()` и `end()`, возвращающие соответственно позиции начала и конца совпадения, а также метод `group()`, который возвратит всю строку соответствия.

1. Начните новую программу на Python, сделав доступными функции импортированного модуля **re** для работы с регулярными выражениями.


regex.py

```
from re import *
```

2. Затем проинициализируйте переменную объектом, содержащим регулярное выражение.

```
pattern = \
compile( '(^|\s)[-a-z0-9_.]+@([-a-z0-9]+\.)+[a-z]{2,6}(\s|$)' )
```

3. Теперь определите функцию, в которой запрашивается ввод строки пользователем и осуществляется проверка этой строки на соответствие шаблону.

```
def get_address() :
address = input( 'Enter Your Email Address: ' )
is_valid = pattern.match( address )
```

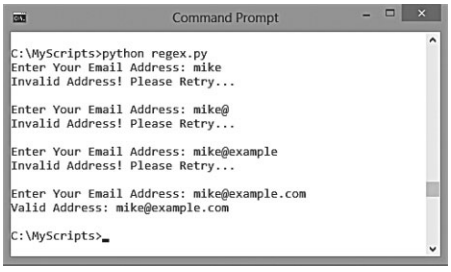
4. После этого добавьте инструкции для вывода соответствующего сообщения, описывающего результат проверки (не забывайте про отступы).

```
if is_valid :
print( 'Valid Address:' , is_valid.group() )
else :
print( 'Invalid Address! Please Retry...\n' )
```

5. Наконец добавьте вызов функции, созданной на третьем шаге.

```
get_address()
```

6. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу — вы увидите, что проверку проходит только тот адрес электронной почты, что введен корректно.




**Совет**

Более подробную информацию об использовании регулярных выражений в Python вы можете найти в соответствующем разделе документации на странице **docs.python.org/3/library/re.html**.

# Заключение

- Функции в Python можно сохранять в модулях, которые имеют такое же имя, что и соответствующий файл, но без расширения *.py*.
- Функцию любого модуля можно сделать доступной в программе при помощи инструкции **import**, после чего к ней можно обращаться, используя точечную запись вида **имя-модуля.имя-функции**.
- Для того чтобы при вызове функции не указывать имя модуля, можно воспользоваться вариантом импортирования **from имя-модуля import**.
- Модуль **sys** содержит атрибуты, которые хранят информацию о номере версии интерпретатора Python, месте его установки в системе, а также пути для поиска исполняемых файлов.
- Атрибут **kwlist** модуля **keyword** содержит список всех ключевых слов языка Python.
- В модуле **math** содержатся методы для выполнения математических вычислений, такие как **math.ceil()** и **math.floor()**.
- Модуль **random** предоставляет методы для работы со случайными числами, а именно метод **random()**, генерирующий псевдослучайные числа, и метод **sample()**, с помощью которого можно получить список из отобранных в случайном порядке элементов.
- Объект **Decimal()**, предоставляемый модулем **decimal**, используется для точного представления чисел с плавающей точкой и рекомендуется применять в финансовых вычислениях.
- Для работы с системным временем модуль **datetime** предоставляет такой объект, как **datetime**, содержащий атрибуты **year**, **month**, **day**, **hour**, **minute**, **second**, **microsecond**, к которым можно обращаться как при помощи точечной записи, так и используя метод **getattr()**.
- Метод **strftime()**, используя директивы, возвращает форматированную часть объекта **datetime**.
- Метод **time()** модуля **time** возвращает текущее время в секундах, прошедшее с момента начала эры Unix.
- Методы **gmtime()** и **localtime()** возвращают объект **struct_time**, имеющий атрибуты, в которых содержатся компоненты даты и времени.
- В модуле **re** существует метод **compile()**, создающий шаблоны регулярных выражений, а также метод **match()** для проверки соответствия строки указанному шаблону.

# 6 Строки и работа с файлами

*В данной главе демонстрируется, каким образом работать с переменными строкового типа и текстовыми файлами при программировании на Python.*

- **Работа со строками**
- **Форматирование строк**
- **Модификация строк**
- **Преобразование строк**
- **Доступ к файлам**
- **Чтение и запись файлов**
- **Изменение текстового файла**
- **Консервация данных**
- **Заключение**

# Работа со строками

В программах на языке Python для работы с переменными строкового типа используются различные операторы, представленные в таблице ниже.

| Оператор | Описание | Пример |
|---|---|---|
| + | Конкатенация (объединение) строк | 'Hello' + 'Mike' |
| * | Повторение строки указанное число раз | 'Hello' * 2 |
| [ ] | Выбор символа по указанному индексу | 'Hello' [0] |
| [ : ] | Извлечение среза по указанному диапазону индексов | 'Hello' [ 0 : 4 ] |
| in | Проверка вхождения — возвращает True, если символ или подстрока присутствует в строке | 'H' in 'Hello' |
| not in | Обратная операция — возвращает True, если символ или подстрока в строке отсутствует | 'h' not in 'Hello' |
| r/R | «Сырая строка» — подавление экранирующей последовательности | print( r'\n' ) |
| ''' ''' | Строка документации — для описания модуля, функции, класса или метода | def sum( a,b ) : ''' Add Args ''' |

**Внимание**

Операции проверки вхождения чувствительны к регистру, поэтому запись 'A' in 'abc' не даст положительного результата.

Операторы извлечения срезов [ ] и [:] представляют строку в виде простого списка, содержащего отдельные символы в качестве элементов, к которым можно обращаться по их порядковому номеру.

Аналогично операторы вхождения **in** и **not in** выполняют поиск в строке, работая с ее символами как с элементами списка.

Оператор «сырая строка», **r** (или **R**), должен располагаться непосредственно перед открывающими кавычками для подавления управляющих символов в строке. Используется обычно в случаях, когда строка содержит символ \.

**Внимание**

Оператор извлечения среза [:] возвращает строку до символа, чей порядковый номер указан последним в диапазоне, не включая его.

Так называемая **строка документации** представляет собой многострочный комментарий, описывающий модуль, функцию, класс или метод. В этих компонентах программы он ставится в самом начале и должен заключаться в три одинарные кавычки.

Строка документации во время выполнения программы доступна в виде специального атрибута **__doc__** объекта, к которому можно обра-

титься, используя точечную запись. Как правило, все модули, функции и классы содержат данную строку документации.

1. Начните новую программу на Python, определив простейшую функцию, включающую строку документации.


manipulate.py

```
def display( s ) :

'''Выводим значение аргумента.'''

print( s )
```

2. Затем добавьте инструкцию для вывода описания функции.

```
display( display.__doc__ )
```

3. Теперь добавьте инструкцию для вывода неформатированной («сырой») строки, которая содержит символ \.

```
display( r'C:\Program Files')
```

4. После этого добавьте инструкцию для вывода объединенной строки, включающей управляющий символ и знак пробела.

```
display( '\nHello' + ' Python' )
```

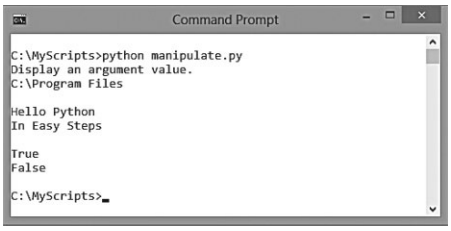
5. Затем добавьте инструкцию, выводящую срез указанной строки в соответствии с диапазоном номеров элементов.

```
display( 'Python In Easy Steps\n' [ 7 : ] )
```

6. Наконец отобразите результат поиска символов `p` и `P` в указанной строке

```
display( 'p' in 'Python' )

display( 'P' in 'Python' )
```

7. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу — вы увидите результат обработки строк.




**На заметку**

Не забывайте заключать строковые значения в одинарные или двойные кавычки.

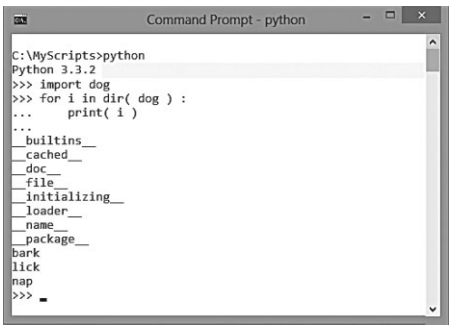
**Совет**

Если при извлечении среза опущен начальный индекс диапазона, то предполагается, что он равен нулю, а при отсутствии конечного индекса берется значение длины строки.

# Форматирование строк

В языке Python существует встроенная функция `dir()`, которая может быть использована для получения имен всех функций и переменных, определенных в модуле.

При этом имя нужного модуля указывается ей в скобках в качестве параметра. Для этих целей можно воспользоваться интерактивным режимом, импортировать требуемый модуль и вызвать функцию `dir()`. Ниже представлен пример, в котором проверяется модуль `dog`, созданный в предыдущей главе:




Имена, которые начинаются и заканчиваются символом двойного подчеркивания, являются зарезервированными объектами языка Python, а все остальные — определенными программистом. При помощи функций `dir()` можно также получить список имен функций и переменных, определенных по умолчанию в модуле `__builtins__` таких как, например, функция `print()` и объект `str`.

Объект `str` определяет несколько очень полезных методов для форматирования строк, включая метод `format()`, который производит подстановки. Строка, которую необходимо отформатировать методом `format()`, может содержать текстовые поля и поля для замены, куда будет подставляться текст из списка элементов, разделенных запятой. Каждое поле замены обозначается парой фигурных скобок {}. Внутри фигурных скобок может стоять порядковый номер заменяемого элемента, в соответствии с которым будут происходить подстановки из списка.

Строки можно также форматировать с помощью оператора замены `%s`, как в языке C. Данный оператор будет помечать места в строке, куда будет вставляться текст из упорядоченного списка значений.

**Совет**

Обратите внимание, что после вызова функция `dir()` среди других имен появляется также атрибут `__doc__`, который мы рассмотрели в предыдущем примере.

**На заметку**

Не путайте объект `str`, описываемый здесь, с функцией `str()`, которая преобразует переменные к строковому типу.

1. Начните новую программу на Python, проинициализировав переменную форматированной строкой.

```
snack = '{} and {}'.format( 'Burger' , 'Fries' )
```

2. Затем выведите значение переменной, чтобы увидеть текст, подставленный в нее в указанном порядке.

```
print( '\nReplaced:' , snack )
```

3. Теперь присвойте этой же переменной значение строки, отформатированной по-другому (с использованием индексов)

```
snack = '{1} and {0}'.format( 'Burger' , 'Fries')
```

4. Теперь выведите значение переменной заново, чтобы увидеть, что теперь строки подставляются в соответствии с указанным порядком индексов.

```
print( 'Replaced:' , snack )
```

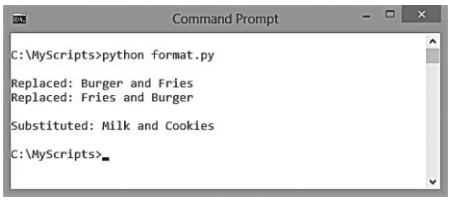
5. Присвойте переменной еще одно значение строки.

```
snack = '%s and %s' % ( 'Milk' , 'Cookies')
```

6. Наконец выведите значение переменной еще раз, чтобы увидеть, что строки подставились в нужном порядке.

```
print( '\nSubstituted:' , snack )
```

7. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу — вы увидите результат форматирования строк.





`format.py`

**Внимание**

Вокруг индексов в полях замены недопустимы символы пробела.

**Совет**

Для других типов данных при форматировании используйте `%d` для целых чисел, `%c` — для символов, `%f` — для чисел с плавающей точкой.

**Внимание**

Символ пробела не является алфавитно-цифровым, поэтому при проверке, включающей пробел строки, функция `isalnum()` возвратит значение `false`.

# Модификация строк

Объект `str` имеет очень много полезных методов для проверки содержимого строк, а также их модификации. К этим методам можно обращаться, используя точечную запись. Наиболее распространенные методы модификации строк, а также их краткое описание, представлены в таблице ниже.

| Метод | Описание |
|---|---|
| `capitalize( )` | Переводит первый символ строки в верхний регистр, а все остальные — в нижний |
| `title( )` | Переводит первую букву каждого слова в верхний регистр, а все остальные — в нижний |
| `upper( )`<br>`lower( )`<br>`swapcase( )` | Преобразование строки к верхнему регистру, к нижнему регистру, и смена регистра на противоположный |
| `join( seq )` | Сборка строки из списка *seq* с добавлением разделителя |
| `lstrip( )`<br>`rstrip ( )`<br>`strip( )` | Удаление пробелов в начале строки, в конце строки, в начале и конце |
| `replace( old , new )` | Замена всех подстрок *old* на подстроку *new* |
| `ljust( w , c )`<br>`rjust( w , c )` | Подгоняет строку под ширину в *w* символов, добавляя справа или слева символы *c* |
| `center( w , c )` | Центрирует строку, подгоняя под ширину в *w* символов, добавляя справа и слева символы *c* (по умолчанию добавляется пробел) |
| `count( sub )` | Возвращает количество вхождений подстроки *sub* |
| `find( sub )` | Возвращает номер первого вхождения подстроки *sub* или `-1`, если подстрока не найдена |
| `startswith( sub )`<br>`endswith( sub )` | Возвращает `True`, если подстрока *sub* найдена в начале строки (в конце строки), и `False` — в противном случае |
| `isalpha( )`<br>`isnumeric( )`<br>`isalnum( )` | Возвращает `True`, если все символы строки являются только буквами, только цифрами, только буквами или цифрами, и `False` — в противном случае |
| `islower( )`<br>`isupper( )`<br>`istitle( )` | Возвращает `True`, если все символы строки в нижнем регистре, в верхнем регистре, все первые буквы слов в верхнем регистре, и `False` — в противном случае |
| `isspace( )` | Возвращает `True`, если строка пустая (то есть в ней содержатся пробелы, символы табуляции, символы новой строки), в противном случае возвращается `False` |
| `isdigit( )`<br>`isdecimal( )` | Возвращает `True`, если строка содержит только цифры, только десятичные числа, в противном случае возвращается `False` |

1. Начните новую программу на Python, проинициализировав строковую переменную, содержащую символы в нижнем регистре и пробелы.


modify.py

```
string = 'python in easy steps'
```

2. Теперь выведите получившиеся в результате работы функций **capitalized**, **titled** и **centered** строки.

```
print( '\nCapitalized:\t' , string.capitalize() )

print( '\nTitled:\t\t' , string.title() )

print( '\nCentered:\t' , string.center( 30 , '*' ) )
```

3. Затем выведите строки со всеми символами в верхнем регистре, а также объединенными с символами **.

```
print( '\nUppercase:\t' , string.upper() )

print( '\nJoined:\t\t' , string.join( '**' ) )
```

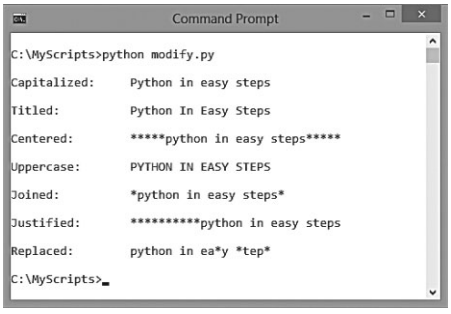
4. Теперь выведите новую строку, которая дополнена символами * слева.

```
print( '\nJustified:\t' ,string.rjust( 30 , '*' ) )
```

5. Наконец выведите строку, в которой все символы **s** заменены на *.

```
print( '\nReplaced:\t' , string.replace( 's' , '*' )
```

6. Сохраните файл в вашем рабочем каталоге, откройте командную строку, запустите программу — вы увидите вывод модифицированных строк.




**На заметку**

При использовании метода `rjust()` выровненная по правому краю строка заполняется символами слева, а при использовании `ljust()` — выровненная по левому краю строка заполняется символами справа.

# Преобразование строк

До версии 3.0 строковые символы в языке Python хранились в соответствии с их числовыми кодами формата ASCII в диапазоне от 0 до 127, представляя только символы латинского алфавита без символов с ударением. Например, символ нижнего регистра `a` имел числовой код `97`. В действительности каждый байт памяти компьютера позволяет сохранять значения в диапазоне от 0 до 255, но этого все равно недостаточно для представления всех символов с ударением, а также символов нелатинского алфавита. Например, общее количество символов с ударениями, используемых в западноевропейском языке, а также в кириллице, больше, чем 127, и, следовательно, они не могут быть представлены диапазоном от 128 до 255. В последних версиях языка Python данное ограничение успешно преодолено с помощью использования кодовых страниц в формате ЮНИКОД (Unicode). Тем самым все символы и алфавиты представлены числовым диапазоном от 0 до 1114111. Символы, которые находятся выше диапазона ASCII, могут требовать для хранения два байта, например шестнадцатеричное значение `0xC3 0xB6` для символа `ö`.

**Совет**

Термин ASCII означает аббревиатуру для American Standard Code for Information Interchange (Американский стандартный код для обмена информацией).

Для преобразования символов в кодировку Unicode существует метод `encode()` объекта `str`, а для обратного действия — метод `decode()`.

В модуле `unicodedata` существует полезный метод `name()`, который отображает имя каждого символа в формате Unicode. Таким образом, нелатинские, а также символы с ударениями можно получить, используя их Unicode-имя, либо с помощью преобразования их шестнадцатеричного представления.


`unicode.py`

1. Начните новую программу на Python, проинициализировав переменную, содержащую строку с не-ASCII-символами, а затем выведите ее значение, тип данных и длину.

```
s = 'Röd'

print( '\nRed String:' , s )

print( 'Type:' , type( s ) , '\tLength:' , len( s ) )
```

2. Теперь преобразуйте строку с помощью метода `encode`, а затем заново выведите ее значение, тип данных и длину.

```
s = s.encode( 'utf-8' )

print( '\nEncoded String:' , s )

print( 'Type:' , type( s ) , '\tLength:' , len( s ) )
```

3. Произведите обратное преобразование с помощью метода **decode**, затем еще раз выведите значение, тип данных и длину строки — чтобы отобразить шестнадцатеричный код не-ASCII-символа.

```
s = s.decode( 'utf-8' )

print( '\nDecoded String:' , s )

print( 'Type:' , type( s ) , '\tLength:' , len( s ) )
```

4. Добавьте инструкции с импортом функций модуля **unicodedata** и получите Unicode-имя каждого символа в строке, создав цикл.

```
import unicodedata

for i in range( len( s ) ) :

print( s[ i ] , unicodedata.name( s[ i ] ) , sep = ' : ' )
```

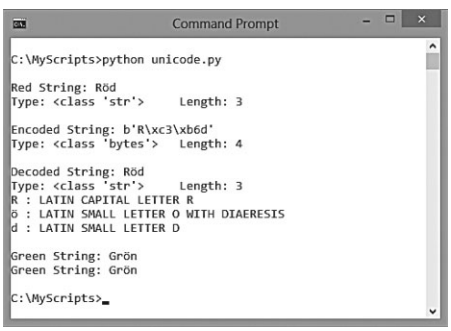
5. Теперь добавьте инструкции, присваивающие переменной новое значение, которое включает шестнадцатеричный код не-ASCII-символа, а затем выведите преобразованную строку.

```
s = b'Gr\xc3\xb6n'

print( '\nGreen String:' , s.decode( 'utf-8' ) )
```

6. Наконец добавьте инструкции с присваиванием переменной нового значения, которое включает Unicode-имя для не-ASCII-символа, а затем выведите строку.

```
s = 'Gr\N{LATIN SMALL LETTER O WITH DIAERESIS}n'

print( 'Green String:' , s )
```

7. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу — чтобы увидеть преобразованные строки и имена Unicode-символов.




**На заметку**

Строка, содержащая байтовые адреса, должна начинаться с префикса **b** для обозначения, что это не строковый, а байтовый литерал.

**Внимание**

В данном примере Unicode-имя — это содержащиеся между фигурных скобок символы верхнего регистра, перед которыми стоит префикс **\N**.

# Доступ к файлам

Если с помощью функции `dir()` проверить модуль `__builtins__`, то можно увидеть, что данный модуль содержит объект `file`, который определяет несколько методов для работы с файлами системы, включая такие методы, как `open()`, `read()`, `write()` и `close()`.

Перед тем как что-то читать из файла или записывать в него, первым делом его следует открыть, используя метод `open()`. Данный метод требует два строковых аргумента для указания имени и пути расположения файла в системе, а также одного из спецификаторов режима открытия файла.

**Внимание**

Аргументы, представляющие собой режим открытия файла, являются строковыми величинами, поэтому они должны быть заключены в кавычки.

| Файловый режим | Операция |
|---|---|
| `r` | Открыть для чтения существующий файл |
| `w` | Открыть существующий файл для записи. Создает новый файл, если он не существует, или открывает существующий файл и стирает все его содержимое |
| `a` | Режим добавления текста. Открывает или создает текстовый файл для записи в конец файла |
| `r+` | Открыть текстовый файл для чтения или записи |
| `w+` | Открыть текстовый файл для записи или чтения |
| `a+` | Открыть или создать текстовый файл для чтения или записи в конец файла |
| Если после любого из перечисленных выше режимов файлов добавлен символ `b`, то операция будет относиться не к текстовому, а к двоичному файлу. Например, `rb` или `w+b`. | |

**Совет**

Можно также использовать метод `readlines()`, который возвращает список всех строк.

После того как вы открыли файл и у вас появился объект `file`, с помощью свойств последнего вы можете получить различные подробности, относящиеся к данному файлу.

| Свойство | Описание |
|---|---|
| `name` | Имя открываемого файла |
| `mode` | Режим открытия файла |
| `closed` | Возвращает `True`, если файл был закрыт, и `False` — если нет |
| `readable( )` | Логическая величина, определяющая, установлено ли на файл разрешение по чтению (`True` или `False`) |
| `writable( )` | Логическая величина, определяющая, установлено ли на файл разрешение по записи (`True` или `False`) |

1. Начните новую программу на Python с создания файлового объекта, которым будет являться новый текстовый файл с именем *example.txt* для записи содержимого в него.


access.py

```
file = open( 'example.txt' , 'w' )
```

2. Теперь добавьте инструкции для вывода имени файла и режима его открытия.

```
print( 'File Name:' , file.name )

print( 'File Open Mode:' , file.mode )
```

3. После этого добавьте инструкции для вывода разрешений на доступ к файлу.

```
print( 'Readable:' , file.readable() )

print( 'Writable:' , file.writable() )
```

4. Теперь определите функцию, задающую состояние файла.

```
def get_status( f ) :

if ( f.closed != False ) :

return 'Closed'

else :

return 'Open'
```

5. Наконец добавьте инструкции для вывода текущего состояния файла, затем закройте файл и выведите его состояние еще раз.

```
print( 'File Status:' , get_status( file ) )

file.close()

print( '\nFile Status:' , get_status( file ) )\
```

6. Сохраните файл в вашем рабочем каталоге, затем откройте командную строку и запустите программу — вы увидите результат открытия файла, записи в него и последующего закрытия.


**На заметку**

Если ваша программа пытается открыть несуществующий файл в режиме `r`, то интерпретатор сообщит об ошибке.

# Чтение и запись файлов

После того как файл был успешно открыт, в зависимости от указания режима при вызове метода `open()`, с ним можно производить различные действия, такие как чтение, добавление содержимого и запись. После выполнения этих действий файл должен быть закрыт с помощью метода `close()`.

Как вы и, вероятно, ожидали, метод `read()` возвращает все содержимое файла, а метод `write()` добавляет содержимое к файлу.

Вы можете быстро и эффективно читать все содержимое файла, используя циклы и итерации по строкам.


`file.py`

1. Начните новую программу на Python, проинициализировав переменную, содержащую объединенные строки, а также символы новой строки.

```
poem = 'I never saw a man who looked\n'

poem += 'With such a wistful eye\n'

poem += 'Upon that little tent of blue\n'

poem += 'Which prisoners call the sky\n'
```

2. Далее добавьте инструкцию для создания файлового объекта для нового текстового файла с именем *poem.txt* для записи в него содержимого.

```
file = open( 'poem.txt' , 'w' )
```

3. Теперь добавьте инструкции для записи строки, содержащейся в переменной, в текстовый файл, а затем закройте этот файл.

```
file.write( poem )

file.close()
```

**Внимание**

Операция записи в существующий файл автоматически перезапишет все его содержимое.

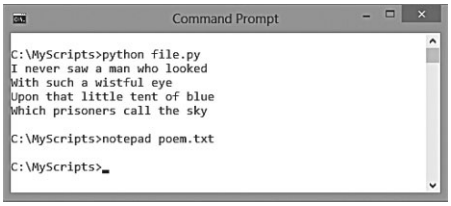
4. Затем добавьте инструкцию, создающую файловый объект для существующего текстового файла *poem.txt* для выполнения чтения из него.

```
file = open( 'poem.txt' , 'r' )
```

5. Теперь добавьте инструкции для вывода содержимого текстового файла, а затем закройте его.

```
for line in file :

print( line , end = '' )

file.close()
```

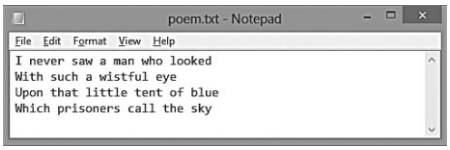
6. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу — вы увидите результат создания файла и вывода его содержимого.




**Совет**

Обратите внимание, что в строках содержится управляющий символ новой строки `\n`, и поэтому в самой функции `print()` подавляется перевод строки.

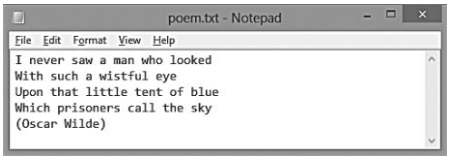
7. Запустите текстовый редактор Блокнот (Notepad), для того чтобы подтвердить, что новый текстовый файл существует, и посмотреть его содержимое, записанное с помощью программы.




8. Теперь добавьте инструкции в конец программы для добавления в текстовый файл подписи, затем сохраните файл еще раз.

```
file = open( 'poem.txt' , 'a' )

file.write( '(Oscar Wilde)' )

file.close()
```

9. Запустите программу заново, чтобы перезаписать текстовый файл, а затем просмотрите его содержимое в текстовом редакторе Блокнот (Notepad). Вы увидите, что теперь в конец содержимого оригинального текста добавлена подпись.




**Совет**

Можете использовать также метод файлового объекта `readlines()`, который возвращает список всех строк в файле — каждая строка является одним элементом списка.

# Изменение текстового файла

Метод `read()` файлового объекта по умолчанию читает все содержимое файла с начала до конца, то есть с нулевого индекса до индекса, соответствующего последнему символу файла. Дополнительно данный метод может принимать целочисленный аргумент, указывающий ему, сколько символов нужно прочитать из файла.

Существует также способ доступа к файлу из произвольной позиции. Для этого используется метод `seek()`, для него указывается позиция, с которой следует начинать читать или записывать. Данный метод принимает целочисленный аргумент, определяющий, на сколько символов от начала файла нужно сместиться, чтобы начать определенное действие.

Текущую позицию внутри файла можно получить в любое время с помощью вызова метода файлового объекта `tell()`, который возвращает целочисленное значение, соответствующее порядковому номеру текущего символа текстового файла.

При работе с файловыми объектами в Python хорошей практикой является использование ключевого слова `with` для группировки в блок инструкций, работающих с файлом. Использование данного приема позволит вам, во-первых, убедиться в том, что файл корректно закрылся после выполнения с ним операций, даже если возникли исключения, а, во-вторых, синтаксис будет гораздо короче, чем при использовании аналогичных блоков `try-except`.


update.py

1. Начните новую программу на Python с присваивания строковой переменной значения, содержащего текст, который нужно будет записать в файл.

```
text = 'The political slogan "Workers Of The World Unite!"

is from The Communist Manifesto.'
```

2. Теперь добавьте инструкции для записи строки текста в файл и отображения текущего состояния файла, используя блок `with`.

```
with open( 'update.txt' , 'w' ) as file :

file.write( text )

print( '\nFile Now Closed?:' , file.closed )
```

3. Затем после кодового блока `with` без отступа добавьте инструкцию для вывода нового состояния файла.

```
print( 'File Now Closed?:' , file.closed )
```

4. Теперь заново откройте файл и отобразите его содержимое, чтобы убедиться, что он теперь содержит всю текстовую строку.

```
with open( 'update.txt' , 'r+' ) as file :

text = file.read()

print( '\nString:' , text )
```

5. После добавьте инструкции для вывода текущей позиции в файле, затем сместите текущую позицию и отобразите ее значение заново.

```
print( '\nPosition In File Now:' , file.tell() )

position = file.seek( 33 )

print( 'Position In File Now:' , file.tell() )
```

6. Теперь добавьте инструкцию для перезаписи текста с текущей позиции в файле.

```
file.write( 'All Lands' )
```

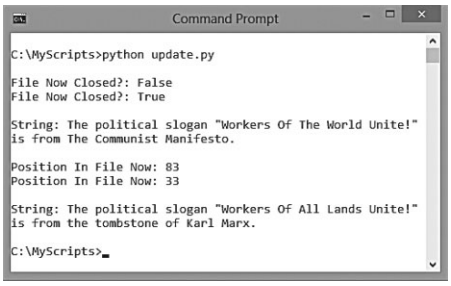
7. Добавьте инструкции для изменения текущей позиции в файле еще раз, а также перезаписи текста с этой новой позиции.

```
file.seek( 59 )

file.write( 'the tombstone of Karl Marx.' )
```

8. Наконец добавьте инструкции для возврата в начальную позицию файла и отображения всего его обновленного содержимого.

```
file.seek( 0 )

text = file.read()

print( '\nString:' , text )
```

9. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу — вы увидите результат изменений текстового файла.




**Совет**

Метод `seek()` способен принимать второй необязательный аргумент, значениями которого могут являться `0`,`1` или `2`, указывающие, откуда производить отсчет символов — от начала, от текущего символа либо от конца текста соответственно. Ноль является значением по умолчанию.

**На заметку**

Как и в случае со строками, первый символ в файле имеет индекс не один, а ноль.

# Консервация данных

Как мы увидели из предыдущего примера, строковые данные в языке Python легко могут быть сохранены в текстовые файлы с помощью определенных методов. Другие типы данных, такие как числа, списки или словари можно также сохранять в текстовых файлах, но для этого требуется сначала преобразовать их в строки. А возвращение этих данных к их первоначальному типу будет требовать еще одного преобразования. Самый простой способ для достижения того, чтобы любой объект данных при этом оставался неизменным, предлагает модуль **pickle** языка Python.

Процесс консервации (или упаковки) объектов сохраняет строковое представление объекта, которое впоследствии может быть «распаковано» к его первоначальному состоянию. Это является наиболее распространенной процедурой программирования на Python.

Для того чтобы сохранить в файл объект, который нужно преобразовать, используется метод **dump()** модуля **pickle**, которому указываются в качестве аргументов объект и файл. Для последующего восстановления этого объекта из файла используется метод **load()**, принимающий в качестве единственного аргумента имя файла.

Поскольку для хранения файлов не требуется, чтобы они были записаны в человеко-читаемом виде, то наиболее эффективно хранить файл в двоичном виде.

Если программе нужно будет проверить существование сохраняемого файла, то для этого в языке Python используется модуль **os**, предлагающий объект **path**, у которого есть метод **isfile()**, возвращающий значение **True**, если указанный ему в скобках файл найден.


**data.py**

1. Начните новую программу на Python, сделав доступными методы модулей **pickle** и **os**.

```
import pickle , os
```

2. Теперь добавьте проверку существования указанного файла данных.

```
if not os.path.isfile( 'pickle.dat' ) :
```

3. Затем добавьте инструкцию для создания списка из двух элементов для случая, когда указанный файл не найден.

```
data = [ 0 , 1 ]
```

4. Теперь добавьте инструкции, запрашивающие ввод значений для каждого из элементов списка.

```
data[ 0 ] = input( 'Enter Topic: ' )

data[ 1 ] = input( 'Enter Series: ' )
```

5. Создайте двоичный файл, открываемый по записи.

```
file = open( 'pickle.dat' , 'wb' )
```

6. Произведите запись значений переменных из списка в виде данных в двоичный файл.

```
pickle.dump( data , file )
```

7. После записи файла не забудьте его закрыть.

```
file.close()
```

8. Теперь добавьте инструкцию в альтернативную ветку на случай, если открываемый файл существует, чтобы считать из него указанные данные.

```
else :

file = open( 'pickle.dat' , 'rb' )
```

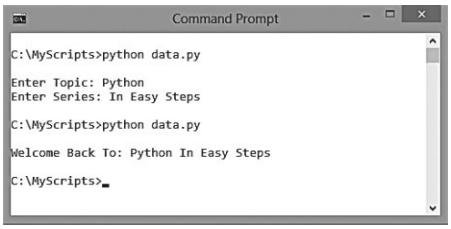
9. Добавьте инструкции для загрузки данных, хранящихся в этом существующем файле, в переменную, а затем закройте файл.

```
data = pickle.load( file )

file.close()
```

10. Наконец добавьте инструкцию для вывода восстановленных данных.

```
print( 'Welcome Back To:' + data[0] + ',' + data[1] )
```

11. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу — вы увидите, как пользовательский ввод сохраняется в файл и затем вызывается.




**Совет**

Консервация в Python является стандартным способом для создания объектов, которые затем могут быть использованы в других программах.

**На заметку**

Несмотря на то, что в данном примере используются две строки, при консервации данных могут обрабатываться почти любые типы объектов Python.

# Заключение

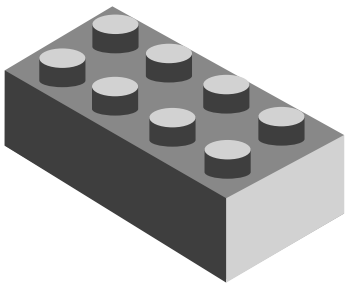
- Обработку строк в Python можно производить с помощью операторов конкатенации, `+`, извлечения среза, `[]`, вхождения, `in` и `not in`.
- Существует специальный атрибут `__doc__`, содержащий строку документации, описывающую модуль, функцию, класс или метод.
- Для проверки имен функций или переменных, определенных в модуле, может быть использована встроенная функция `dir()`.
- Модуль `__builtins__` содержит функции и переменные, которые доступны по умолчанию, например, функцию `print()`.
- Объект `str` содержит метод `format()` для форматирования строк, а также многочисленные методы для изменения строк, например `capitalize()`.
- По умолчанию при преобразовании символов используется кодировка Unicode, но можно также использовать методы `encode()` и `decode()` объекта `str`.
- Отобразить Unicode-имя каждого из символов поможет вам метод `name()` модуля `unicodedata`.
- Объект `file` содержит методы для работы с файлами, такие как `open()`, `read()`, `write()` и `close()`, а также параметры, описывающие свойства файла.
- Для метода `open()` нужно указывать два строковых аргумента — имя файла и режим открытия файла, такой как, например, `'r'` для чтения из файла.
- С помощью метода `seek()` может быть указана позиция, с которой начинать читать или записывать в файл, а метод `tell()` сообщает об этой текущей позиции.
- Для группировки инструкций, работающих с файловыми операциями, и автоматического закрытия открытого файла полезно использовать ключевое слово `with`.
- Так называемый процесс консервации объектов сохраняет строковое представление объекта, который впоследствии может быть распакован в свое первоначальное состояние.
- Метод `dump()` объекта `pickle` требует передачи двух аргументов для указания объекта преобразования и имени файла, в котором сохранять данные.
- Сохраненные данные объекта могут быть вызваны с помощью метода `load()` объекта `pickle` с указанием имени файла.

# 7 Объектное программирование

*В данной главе рассматривается применение объектно ориентированного программирования в языке Python.*

- **Инкапсуляция данных**
- **Создание экземпляров объектов**
- **Доступ к атрибутам класса**
- **Встроенные атрибуты**
- **Сборка мусора**
- **Наследование свойств**
- **Переопределение основных методов**
- **Реализация полиморфизма**
- **Заключение**

# Инкапсуляция данных



В языке Python **класс** — это тип, описывающий набор свойств, которые характеризуют объект. Каждый класс имеет структуру данных, которая может содержать как функции, так и переменные, характеризующие объект.

Членами класса могут быть функции, называемые **методами**, а также переменные (объявленные внутри структуры класса), называемые **атрибутами**.

К членам класса можно обращаться в программе с помощью точечной записи, поставив соответствующий суффикс после имени класса, применив следующий синтаксис: ***имя-класса.имя-метода***() или ***имя-класса.имя-атрибута***.

Объявление класса начинается с ключевого слова **class**, после которого следует указываемое программистом имя класса (при выборе имени придерживаются обычного правила именования языка Python, но начинают с букв в верхнем регистре), затем ставится символ двоеточия.

Далее с отступом следуют инструкции, определяющие строку документации класса, объявление переменных — атрибутов класса и определение методов класса. Таким образом, синтаксис блока класса выглядит следующим образом:

```
Class ClassName :
''' строка-документации-класса '''
объявление-переменных-класса
определение-методов-класса
```

Объявление класса, в котором определяются его атрибуты и методы, является образцом, из которого могут быть произведены рабочие копии (**экземпляры**) класса.

**Совет**

Имена классов в Python принято начинать с прописной буквы, а имена объектов — со строчной.

Все переменные, объявленные внутри определения методов, известны как **переменные экземпляра** и доступны только локально в том методе, в котором они были объявлены — к ним нельзя обратиться извне структуры класса.

Как правило, переменные экземпляра содержат данные, передаваемые вызывающим оператором во время создания экземпляра класса. Поскольку эти данные доступны только локально (как говорится «для внутреннего пользования») они фактически скрыты от остальной части программы. Данный прием, называемый **инкапсуляцией** данных, гарантирует, что данные надежно хранятся в структуре класса, и явля-

ется первым принципом объектно ориентированного программирования (ООП).

Ко всем свойствам класса можно обратиться локально, используя точечную запись с префиксом **self** , например, атрибут с именем **sound** можно вызвать так: **self.sound**. Кроме того, все определения методов должны содержать **self** в качестве первого аргумента, то есть метод с именем **talk** выглядит как **talk( self )**.

При создании экземпляра класса автоматически вызывается специальный метод **__init__( self )**. Если необходимо передать еще значения для инициализации его атрибутов, то в скобки могут быть добавлены последующие аргументы.

Полное объявление класса в Python может выглядеть, например, так:

```
class Critter :
''' Базовый класс для всех живых существ. '''
count = 0
def __init__( self , chat ) :
self.sound = chat
        Critter.count += 1
def talk( self ) :
return self.sound
```

Давайте рассмотрим компоненты класса, представленные в этом примере.

- Переменная **count** является переменной класса, чье целочисленное значение доступно всем экземплярам данного класса. К ней можно обратиться с помощью записи **Critter.count** как внутри, так и извне класса.
- Первый из методов **__init__()** — метод инициализации, который вызывается автоматически во время создания экземпляра класса.
- Метод **__init__()** в данном случае инициализирует переменную экземпляра **sound** значением, переданным из аргумента **chat**, и увеличивает значение переменной класса **count** при каждом создании экземпляра этого класса.
- Второй метод **talk()** объявлен как обычная функция, за исключением автоматически указываемого аргумента **self** — других значений для передачи из вызываемого оператора не требуется.
- В данном случае метод **talk()** возвращает значение, инкапсулированное в переменную экземпляра **sound**.

**На заметку**

К строке документации определенного класса можно обратиться через специальный атрибут **__doc__** следующим образом: **Имякласса.__doc__**.

**Совет**

Поскольку программный класс не может в точности эмулировать подлинный объект, цель заключается в том, чтобы инкапсулировать все соответствующие атрибуты и действия.

# Создание экземпляров объектов

**Экземпляр** класса, являющийся объектом, — это просто копия прототипа, создаваемая с помощью вызова конструктора класса с указанием требуемого числа аргументов внутри скобок при вызове. Аргументы данного вызова должны соответствовать указанным в определении метода `__init__()`, за исключением внутреннего аргумента `self`.

Экземпляр класса представляет собой объект, возвращаемый конструктором, и его можно присвоить переменной, используя синтаксис *`имя-экземпляра = ИмяКласса( аргументы )`*.

Для того чтобы обратиться к методам и атрибутам созданного экземпляра, можно использовать точечную запись, например, *`имя-экземпляра.имя-метода()`* или *`имя-экземпляра.имя-атрибута`*.

Обычно определяют некоторый базовый класс и его хранят в отдельном файле модуля Python. В дальнейшем его можно импортировать в другие программы и, таким образом, с легкостью создавать объекты-экземпляры из так называемого **мастер-прототипа** класса.

1. Начните новую программу на Python с объявления нового класса, содержащего описательную строку документации.

   ```
   class Bird :
   '''Базовый класс, определяющий свойства птиц.'''
   ```

   Bird.py

2. Затем добавьте инструкцию с объявлением и инициализацией переменной — атрибута класса целочисленным значением, равным нулю.

   ```
   count = 0
   ```

3. Теперь определите метод класса, инициализирующий переменную экземпляра и производящий приращение значения переменной класса.

   ```
   def __init__( self , chat ) :
   self.sound = chat
   Bird.count += 1
   ```

4. Наконец добавьте метод класса, при вызове которого возвращается значение переменной экземпляра. Затем сохраните класс в отдельном файле.

   ```
   def talk( self ) :
   return self.sound
   ```

**Совет**

Конструктор создает экземпляр класса и представляет собой не что иное, как простую запись имени класса, после которой следуют скобки, содержащие нужное количество аргументов.

**На заметку**

Вам не нужно передавать значение для аргумента `self`, поскольку он добавляется интерпретатором Python автоматически.

5. Начните еще одну программу на Python, сделав доступными свойства класса с помощью импорта, а затем выведите строку документации данного класса.

```
from Bird import *

print( '\nClass Instances Of:\n' , Bird.__doc__ )
```

6. Теперь добавьте инструкцию для создания экземпляра класса и передачи значения строкового аргумента в переменную экземпляра.

```
polly = Bird( 'Squawk, squawk!' )
```

7. После выведите значение переменной экземпляра и вызовите метод класса для отображения значения переменной класса.

```
print( '\nNumber Of Birds:' , polly.count )

print( 'Polly Says:' , polly.talk() )
```

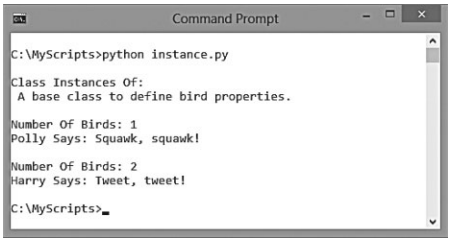
8. Создайте второй экземпляр класса, передав другое значение строкового аргумента переменной экземпляра.

```
harry = Bird( 'Tweet, tweet!' )
```

9. Наконец отобразите значение переменной экземпляра и вызовите метод класса для вывода значения переменной класса.

```
print( '\nNumber Of Birds:' , harry.count )

print( 'Harry Says:' , harry.talk() )
```

10. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу — вы увидите результат создания двух экземпляров класса `Bird`.





`instance.py`

Экземпляр класса `Bird` — `polly`

Экземпляр класса `Bird` — `harry`

**Внимание**

К переменной класса `count` можно также обратиться с помощью записи `Bird.count`, в то время как инкапсулированную переменную экземпляра `sound` можно получить только с помощью вызова метода экземпляра `talk()`.

# Доступ к атрибутам класса

Атрибуты созданного экземпляра класса можно добавлять, изменять или удалять в любое время, используя для доступа к ним точечную запись. Если построить инструкцию, в которой присвоить значение атрибуту, то можно изменить значение, содержащееся внутри существующего атрибута, либо создать новый с указанным именем и содержащий присвоенное значение:

```
имя-экземпляра.имя-атрибута = значение
del имя-экземпляра.имя-атрибута
```

**Внимание**

Имя атрибута, указанное в этих встроенных функциях, должно быть заключено в кавычки.

Альтернативным способом добавления, изменения либо удаления переменной экземпляра является использование встроенных функций Python:

- `getattr( имя-экземпляра , 'имя-атрибута' )` — возвращает значение атрибута экземпляра класса;
- `hasattr( имя-экземпляра , 'имя-атрибута' )` — возвращает `True`, если значение атрибута существует в экземпляре, в противном случае возвращает `False`;
- `setattr( имя-экземпляра , 'имя-атрибута' , значение )` — модифицирует существующее значение атрибута либо создает новый атрибут для экземпляра;
- `delattr( имя-экземпляра , 'имя-атрибута' )` — удаляет атрибут из экземпляра.

Имена атрибутов, автоматически предоставляемые интерпретатором Python, всегда содержат символ подчеркивания, чтобы обозначить «частный» характер этих атрибутов — их не следует изменять либо удалять. Таким же образом вы можете добавить свои собственные атрибуты, обозначив их как «частные», но не забывайте, что они, как и все другие, могут быть изменены.


address.py

1. Начните новую программу на Python, сделав доступными функции класса `Bird`, созданного в предыдущем примере.

```
from Bird import *
```

2. Затем создайте экземпляр класса, после чего добавьте, используя точечную запись, новый атрибут с присвоенным значением.

```
chick = Bird( 'Cheep, cheep!' )

chick.age = '1 week'
```

3. Теперь выведите значения, хранящиеся в обоих атрибутах экземпляра.

```
print( '\nChick Says:' , chick.talk() )

print( 'Chick Age:' , chick.age )
```

4. Измените новый атрибут с помощью точечной записи и выведите его новое значение.

```
chick.age = '2 weeks'

print( 'Chick Now:' , chick.age )
```

5. Опять измените новый атрибут, используя в этот раз встроенную функцию.

```
setattr( chick , 'age' , '3 weeks' )
```

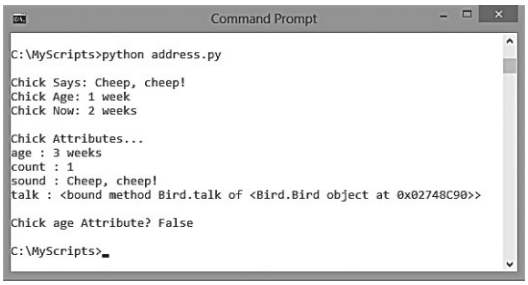
6. После этого выведите список всех атрибутов экземпляра, не являющихся частными, и соответствующие им значения, используя встроенную функцию.

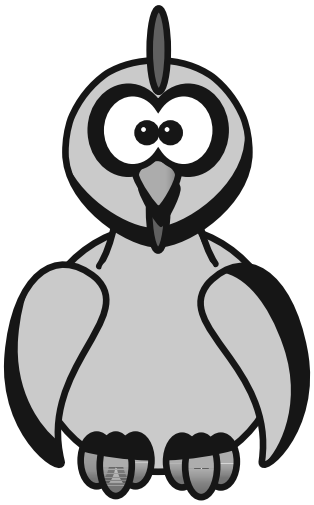
```
print( '\nChick Attributes...' )

for attrib in dir( chick ) :

if attrib[0] != '_' :

print( attrib , ':' , getattr( chick , attrib ) )
```

7. Наконец удалите новый атрибут и проверьте его отсутствие, используя встроенные функции.

```
delattr( chick , 'age' )

print( '\nChick age Attribute?' , hasattr( chick , 'age' ) )
```

8. Сохраните файл в вашем рабочем каталоге, затем откройте командную строку и запустите программу — вы увидите результат обращения к атрибутам экземпляра.



Экземпляр класса `Bird` — `chick`



**На заметку**

Данный цикл пропускает все атрибуты, чьи имена начинаются с символа подчеркивания, так что «частные» атрибуты не попадут в результирующий список.

# Встроенные атрибуты

В языке Python каждый класс автоматически создается с определенным набором встроенных «частных» атрибутов. Доступ к их значениям можно получить, используя точечную запись. Например, чтобы получить значение атрибута строки документации определенного класса, нужно записать *имя-класса*.__doc__.

Чтобы отобразить список всех встроенных атрибутов указанного класса, можно использовать функцию dir(), которой в качестве параметра указать имя класса, а затем проверить, начинается ли имя атрибута с символа подчеркивания.

Встроенный атрибут __dict__ является словарем, содержащим пары ключей и связанных с ними значений. Ключами здесь являются имена атрибутов, а значениями — соответствующие значения атрибута. Словарь базового класса включает в себя метод __init__(), а также все методы и атрибуты класса. Словарь экземпляра класса включает все атрибуты экземпляра.


builtin.py

1. Начните новую программу на Python, сделав доступными функции класса Bird, который был определен ранее в этой главе.

```
from Bird import *
```

2. Теперь добавьте инструкцию для создания экземпляра класса.

```
zola = Bird( 'Beep, beep!' )
```

3. Затем добавьте цикл для вывода значений всех встроенных атрибутов экземпляра.

```
print( '\nBuilt-in Instance Attributes...' )

for attrib in dir( zola ) :

if attrib[0] == '_' :

print( attrib )
```

**Совет**

Значения, хранящиеся в словаре, являются ссылками на область памяти, где хранятся функции.

4. Теперь добавьте цикл для вывода всех элементов из словаря класса.

```
print( '\nClass Dictionary...' )

for item in Bird.__dict__ :

print( item , ':' , Bird.__dict__[ item ] )
```

5. Наконец добавьте цикл для отображения всех элементов из словаря экземпляра.

```
print( '\nInstance Dictionary...' )

for item in zola.__dict__ :

print( item , ':' , zola.__dict__[ item ] )
```

6. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу — вы увидите список встроенных атрибутов.


При выводе словаря класса отображаются все атрибуты этого класса, а при выводе словаря экземпляра отображаются только атрибуты этого экземпляра — атрибуты класса наследуются экземпляром по умолчанию.

Класс `Bird` — экземпляр `zola`

**На заметку**

В этой программе сначала создается экземпляра класса, поэтому метод `__init__()` увеличивает счетчик `count` до вывода словаря экземпляра.

**Совет**

Атрибут `__weakref__` используется для автоматической сборки мусора (удаления так называемых слабых ссылок) в целях увеличения производительности программы.

# Сборка мусора

Когда создается объект — экземпляр класса, под него выделяется уникальный адрес в памяти, который можно посмотреть с помощью встроенной функции `id()`.Python автоматически выполняет «сборку мусора», чтобы освободить память, периодически удаляя ненужные объекты, такие как экземпляры класса, — в результате их адреса в памяти становятся свободными.

Всякий раз, когда объекту присваивается новое имя или он помещается в контейнер (например, в список), его «счетчик ссылок» возрастает. И наоборот, каждый раз, когда объект удаляется или выходит из области видимости, этот счетчик уменьшается. Когда счетчик обнуляется, объект разрешается подвергнуть обработке «сборщиком мусора».

Для уничтожения экземпляра класса можно еще воспользоваться, при необходимости, так называемым **деструктором** — методом `__del__()`, который явно освобождает занятое пространство памяти.

1. Начните новую программу на Python, объявив класс с методом инициализации, в котором создаются две переменные экземпляра, и выводом значения одной из этих переменных.


Songbird.py

```
class Songbird :

def __init__( self , name , song ) :

self.name = name

self.song = song

print( self.name , 'Is Born...' )
```

2. Затем добавьте метод, в котором просто выводятся значения обеих переменных.

```
def sing( self ) :

print( self.name , 'Sings:' , self.song )
```

3. Теперь добавьте метод-деструктор, подтверждающий уничтожение экземпляра класса, затем сохраните файл.

```
def __del__( self ) :

print( self.name , 'Flew Away!\n' )
```

4. Начните еще одну программу на Python, сделав доступными через импорт функции класса `Songbird`.


garbage.py

```
from Songbird import *
```

5. Затем создайте экземпляр класса, после чего выведите значения его атрибутов и его уникальный идентификатор.

```
bird_1 = Songbird( 'Koko' , 'Tweet, tweet!\n' )

print( bird_1.name , 'ID:' , id( bird_1 ) )

bird_1.sing()
```

6. Теперь удалите экземпляр, вызвав метод-деструктор.

```
del bird_1
```

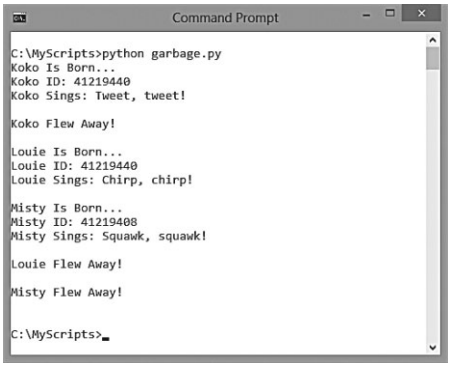
7. Создайте еще два экземпляра класса, после чего снова выведите значения атрибутов экземпляров и их уникальные идентификаторы.

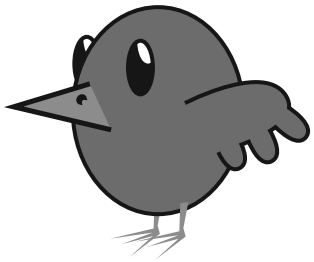
```
bird_2 = Songbird( 'Louie' , 'Chirp, chirp!\n' )

print( bird_2.name , 'ID:' , id( bird_2 ) )

bird_2.sing()

bird_3 = Songbird( 'Misty' , 'Squawk, squawk!\n' )

print( bird_3.name , 'ID:' , id( bird_3 ) )

bird_3.sing()
```

8. Наконец удалите эти экземпляры, вызвав их деструкторы.

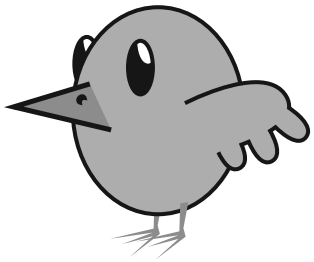
```
del bird_2

del bird_3
```

9. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу — вы увидите результат работы сборщика мусора, освобождающего память компьютера.

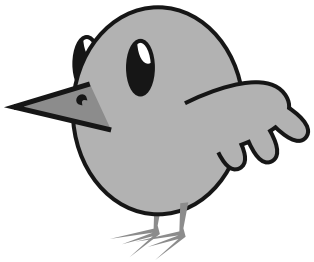


Класс `Songbird` — экземпляр `Koko`



Класс `Songbird` — экземпляр `Louie`

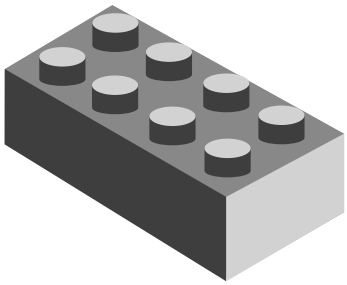


Класс `Songbird` — экземпляр `Misty`

**На заметку**

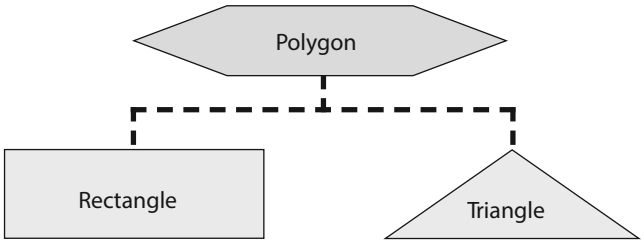
Обратите внимание, что второму экземпляру, созданному в данном примере, выделяется ячейка памяти, освободившаяся после удаления первого экземпляра.

# Наследование свойств



В языке Python класс может быть создан либо как совершенно новый (как в предыдущих примерах), либо как «производный» от существующего. Важно отметить, что производный класс наследует члены родительского (базового) класса, от которого он произошел, в дополнение к своим собственным членам.

Возможность наследовать члены из базового класса позволяет создавать производные классы, имеющие некоторые общие свойства, которые были определены для базового класса. Например, в базовом классе `Polygon` (Многоугольник) могут быть определены такие свойства, как ширина и высота, которые являются общими для всех многоугольников. Классы `Rectangle` (Прямоугольник) и `Triangle` (Треугольник) могут являться производными от класса `Polygon`, наследуя его свойства «ширина» и «высота», а также могут иметь свои собственные члены, определяющие уникальные свойства, присущие только им.




Модель наследования является исключительно мощным инструментом, а также вторым основным принципом объектно ориентированного программирования (ООП).

Чтобы объявить производный класс, нужно после его имени добавить скобки, в которых указать имя родительского класса.


Polygon.py

1. Создайте новую программу на Python, в которой объявляется базовый класс с двумя переменными класса и методом, в котором устанавливаются значения для этих переменных.

```
class Polygon :
width = 0
height = 0
def set_values( self , width , height ) :
Polygon.width = width
Polygon.height = height
```

2. Затем создайте новую программу с определением производного класса, содержащего метод, который возвращает значение нужных вам переменных класса.

```
from Polygon import *

class Rectangle( Polygon ) :

def area( self ) :

return self.width * self.height
```

Rectangle.py

3. Теперь создайте еще одну программу, в которой описывается производный класс и метод, возвращающий значение переменных класса.

```
from Polygon import *

class Triangle( Polygon ) :

def area( self ) :

return ( self.width * self.height ) / 2
```

Triangle.py

4. Сохраните три файла классов, затем начните новую программу, сделав доступными функции обоих производных классов.

```
from Rectangle import *

from Triangle import *
```

inherit.py

5. Затем создайте экземпляры каждого из производных классов.

```
rect = Rectangle()

trey = Triangle()
```

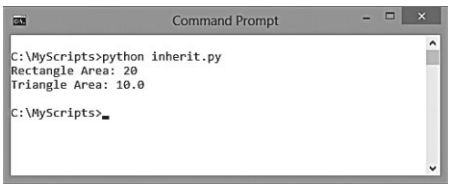
6. Теперь вызовите метод класса, унаследованный от базового, передав аргументы для присваивания значений переменным класса.

```
rect.set_values( 4 , 5 )

trey.set_values( 4 , 5 )
```

7. Наконец выведите результат работы с переменными, унаследованными от базового класса.

```
print( 'Rectangle Area:' , rect.area() )

print( 'Triangle Area:' , trey.area() )
```

8. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу — вы увидите результат использования наследуемых свойств.




**Совет**

Класс может быть объявлен как производный более чем от одного родительского. Для этого нужно в скобках перечислить через запятую имена базовых классов.

**Внимание**

Не путайте экземпляры класса и производные классы: экземпляр — это простая копия класса, в то время как производный класс — это новый класс, который наследует свойства родительского, от которого он произошел.

# Переопределение основных методов

Если в производном классе объявить метод, имеющий то же самое имя и то же число аргументов, что и какой-либо метод в родительском классе, то данный метод «переопределяется». При этом метод из базового класса становится как бы скрытым и недоступным до тех пор, пока не будет вызван явно с использованием имени базового класса.

Если в базовом методе определен какой-то аргумент по умолчанию, то он и будет использоваться при явном вызове метода базового класса, а для тех аргументов, у которых значения по умолчанию не определены, будут подставляться аргументы из переопределенных методов.


Person.py

1. Создайте новую программу на Python, в которой объявляется базовый класс и метод инициализации переменной экземпляра и второй метод для вывода значения этой переменной.

```
class Person :

'''Базовый класс.'''

def __init__( self , name ) :

self.name = name

def speak( self , msg = '(Calling The Base Class)' ) :

print( self.name , msg )
```


Man.py

2. Затем создайте программу, в которой объявляется производный класс с методом, переопределяющим второй метод базового класса.

```
from Person import *

'''Производный класс.'''

class Man( Person ) :

def speak( self , msg ) :

print( self.name , ':\n\tHello!' , msg )
```

3. Теперь создайте еще одну программу, которая также объявляет производный класс и содержит метод, который снова переопределяет тот же самый метод из базового класса.


Hombre.py

```
from Person import *

'''Производный класс.'''

class Hombre( Person ) :

def speak( self , msg ) :

print( self.name , ':\n\tHola!' , msg )
```

4. Сохраните три файла классов, затем начните новую программу, сделав доступными свойства обоих производных классов.

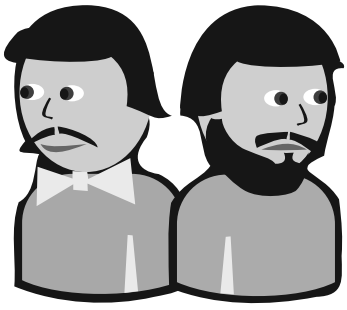

override.py

```
from Man import *

from Hombre import *
```

5. Затем создайте экземпляры каждого из производных классов, проинициализировав переменную экземпляра **name**.

```
guy_1 = Man( 'Richard' )

guy_2 = Hombre( 'Ricardo' )
```

6. Теперь вызовите переопределенные методы каждого из производных классов, присвоив различные значения аргументу **msg**.



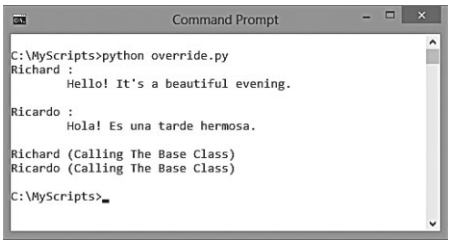
Man – Richard  Hombre - Ricardo

```
guy_1.speak( 'It\'s a beautiful evening.\n' )

guy_2.speak( 'Es una tarde hermosa.\n' )
```

7. Наконец вызовите явно метод базового класса, передав ссылку на каждый из производных классов, но не передав значение для аргумента **msg**, оставив использование значения по умолчанию.

```
Person.speak( guy_1 )

Person.speak( guy_2 )
```

8. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу — вы увидите результаты переопределения методов базового класса.

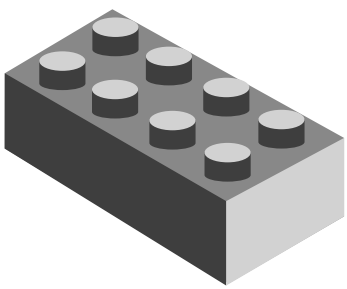



**На заметку**

Для того чтобы методы были переопределены, их объявления должны в точности совпадать.

# Реализация полиморфизма

Тремя краеугольными камнями объектно ориентированного программирования (ООП) являются инкапсуляция, наследование и полиморфизм. Примеры, представленные ранее в этой главе, показали, как данные могут быть инкапсулированы внутри класса Python и как производные классы наследуют свойства от базового. Следующий пример знакомит вас с третьим ключевым понятием ООП — принципом полиморфизма.



Термин **полиморфизм** (от греческого, означает «много форм») описывает способность назначать элементу различные смысловые значения в зависимости от контекста, в котором он используется. В языке Python, например, такой элемент, как символ `+`, может быть описан как полиморфный, потому что он представляет либо оператор арифметического сложения в контексте числовых операндов, либо оператор конкатенации строк, если операндами являются не числа, а символы. Возможно, еще более важным является факт, что методы класса в Python также могут быть полиморфными, потому что язык Python использует так называемую утиную типизацию — смысл которой в том, что … «если она ходит, как утка, плавает, как утка, и крякает, как утка, то эту птицу можно считать уткой».

В языке с утиной типизацией вы можете создать функцию, в которой берется объект любого типа, и вызываются методы этого объекта. Если объект в действительности имеет вызываемые методы (может считаться уткой), то они выполняются, в противном случае функция сигнализирует об ошибке выполнения. Одноименные методы можно создавать для нескольких классов, и в этом случае каждый созданный экземпляр этих классов будет выполнять соответствующую версию метода.


Duck.py

1. Создайте новую программу на Python, объявив класс с двумя методами, выводящими уникальные для него строковые значения.

```
class Duck :
def talk( self ) :
print( '\nDuck Says: Quack!' )
def coat( self ) :
print( 'Duck Wears: Feathers' )
```


Mouse.py

2. Затем создайте еще одну программу, объявив другой класс с двумя методами с теми же именами, что в первом, выводящими свои, уникальные для этого класса строковые значения.

```
class Mouse :
def talk( self ) :
print( '\nMouse Says: Squeak!' )
```

```
def coat( self ) :

print( 'Mouse Wears: Fur' )
```

3. Сохраните оба файла с классами, затем начните новую программу, сделав доступными в ней функции обоих классов.

```
from Duck import *

from Mouse import *
```

4. Теперь определите функцию, которая в качестве аргумента принимает любой одиночный объект и пытается вызвать методы этого объекта.

```
def describe( object ) :

object.talk()

object.coat()
```

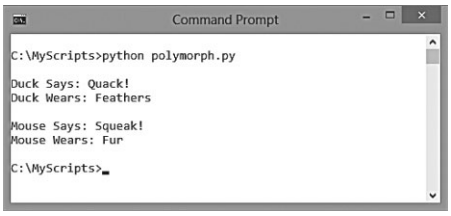
5. Теперь создайте экземпляры каждого из двух классов.

```
donald = Duck()

mickey = Mouse()
```

6. Наконец добавьте инструкции, вызывающие функцию с передачей ей в качестве аргумента сначала экземпляра первого класса, затем второго.

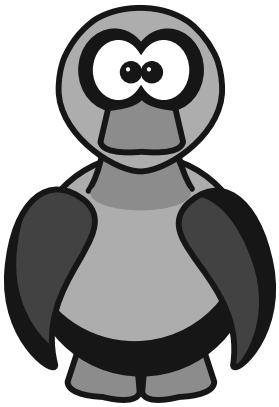
```
describe( donald )

describe( mickey )
```

7. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу — вы увидите, как вызываются методы, связанные с соответствующими классами.

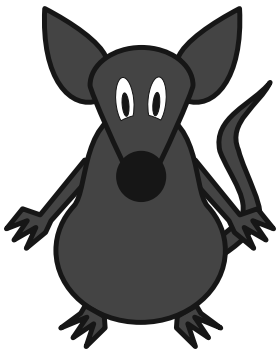



Объектно ориентированное программирование на языке Python позволяет использовать инкапсуляцию данных, наследование и полиморфизм. Методы базового класса могут быть переопределены одноименными методами из производных классов. Но приемы «перегрузки», используемые в других языках (когда в одном классе могут быть созданы методы с тем же именем, но с различным набором аргументов), Python не поддерживает.


polymorph.py



Класс `Duck` — экземпляр `donald`



Класс `Mouse` — экземпляр `mickey`

**На заметку**

Класс может содержать только один метод с определенным именем — перегрузка методов в Python не поддерживается.

# Заключение

- Класс представляет собой прототип структуры данных, описывающий свойства объектов с помощью методов и атрибутов.
- Каждое объявление класса начинается с ключевого слова `class`. После него идет выделенный отступом блок кода, который может содержать строку документации класса, переменные класса и методы класса.
- Переменные класса имеют глобальную область видимости, а переменные экземпляра (объявленные внутри определений методов) — только локальную.
- Переменные экземпляра инициализируются при создании экземпляра класса и надежно инкапсулируют данные в структуре класса.
- К методам и атрибутам внутри класса можно обратиться, используя точечную запись с префиксом `self`.
- Экземпляр класса — это копия прототипа класса. При первом создании экземпляра автоматически вызывается метод `__init__()`.
- С помощью точечной записи можно добавлять, изменять и удалять атрибуты класса. Альтернативным способом обработки атрибутов является использование встроенных функций `getAttr()`, `hasattr()`, `setAttr()` и `delattr()`.
- Имена атрибутов, поддерживаемых автоматически, начинаются с символа подчеркивания. Тем самым интерпретатор Python условно обозначает «частный» характер этих имен.
- Встроенный атрибут `__dict__` содержит словарь класса, в котором хранятся пары `имя_атрибута:значение_атрибута`.
- Интерпретатор Python автоматически выполняет сборку мусора, но для удаления объектов можно использовать ключевое слово `del`, вызывающее деструктор класса.
- Производный класс наследует методы и атрибуты родительского (базового) класса, от которого он произошел.
- Метод производного класса может переопределять одноименный метод родительского класса.
- Python является языком с «утиной типизацией», который поддерживает принцип полиморфизма для одноименных методов, принадлежащих нескольким различным классам.

# 8 Обработка запросов

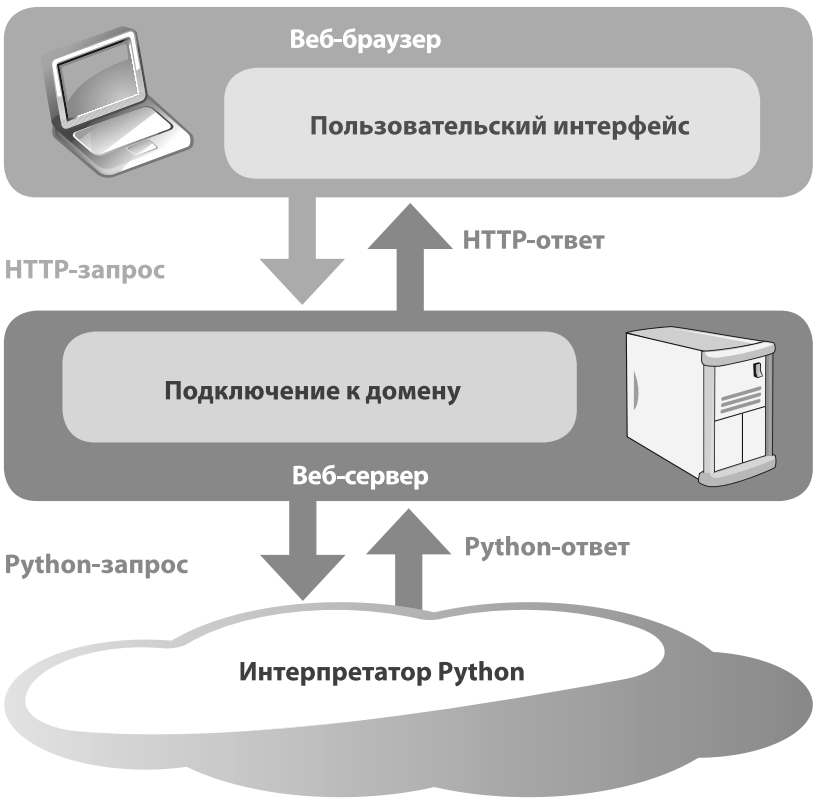
*В данной главе демонстрируется, каким образом можно создавать серверные программы на языке Python для обработки HTML-запросов.*

# Отправка ответов

Когда какой-либо пользователь Всемирной паутины просматривает в своем браузере веб-страницу, то на самом деле его браузер запрашивает страницу у веб-сервера и получает ее в ответ через протокол HTTP.

Адрес запрашиваемой веб-страницы представляет собой документ HTML (обычно с расширением файла *.html*), и веб-сервер возвращает его браузеру, который может отразить содержимое этого файла на экране пользователя.

Если на компьютере, где находится программное обеспечение веб-сервера, также установлен Python, то веб-сервер может быть сконфигурирован таким образом, чтобы распознавать программы на Python (обычно с расширением файла *.py*) и вызывать интерпретатор Python для обработки программного кода перед отправкой HTML-ответа веб-серверу, возвращающему данный ответ уже браузеру клиента.




Примеры в этой главе написаны с использованием свободного программного обеспечения веб-сервера Abyss Personal Edition, которое доступно на сайте **www.aprelium.com**. К веб-серверу можно будет обращаться по доменному имени **localhost**, либо по IP-адресу **127.0.0.1**.

Программа на Python, к которой обратился веб-браузер, может генерировать ответ в виде готового документа HTML, в первой строке кода которого указывается тип содержимого в виде `Content-type:text/html\r\n\r\n`. Веб-браузер анализирует содержимое файла разметки и отображает на экране пользователя.

1. Убедитесь, что веб-сервер запущен и сконфигурирован для использования программ на языке Python.



response.py

2. Теперь начните новую программу на Python с вывода типа сгенерированного содержимого, которое будет являться HTML-документом.

```
print( 'Content-type:text/html\r\n\r\n' )
```

3. Теперь добавьте инструкции для вывода всей веб-страницы, включая теги HTML-разметки.

```
print( '<!DOCTYPE HTML>' )
print( '<html lang="en">' )
print( '<head>' )
print( '<meta charset="UTF-8">' )
print( '<title>Python Response</title>' )
print( '</head>' )
print( '<body>' )
print( '<h1>Hello From Python Online!</h1>' )
print( '</body>' )
print( '</html>' )
```

4. Наконец сохраните файл в каталоге HTML-документов веб-сервера — обычно это */htdocs*.

5. Откройте веб-браузер и запросите файл с веб-сервера через протокол HTTP — вы увидите предоставленный программой на Python HTML-документ.


**На заметку**

Выходная строка описания `Content-type` отправляется в браузер в виде HTTP-заголовка и должна обязательно идти первой.

**Совет**

Заключайте значение атрибутов HTML в двойные кавычки, чтобы не путать их с одинарными кавычками, в которые заключаются строки.

# Обработка данных

Когда браузер производит HTTP-запрос, то из него разрешается передать какие-либо значения в программу на Python, находящуюся на веб-сервере. Эти значения могут быть использованы в программе, а также переданы в обратном ответе браузеру.

Для обработки данных, переданных из веб-браузера через HTTP-запрос в программу на Python, допустимо использовать модуль **cgi**. В нем есть конструктор **FieldStorage()**, который создает объект, хранящий переданные данные в виде словаря, содержащего пары ключ:значение. Любое значение можно будет получить с помощью метода **getvalue()** объекта FieldStorage, указав соответствующее имя ключа в виде параметра для данного метода.

Браузер может передавать данные в программу, используя метод **GET**, который просто добавляет к URL-адресу программы пару в виде **ключ=значение**. Эта пара следует после символа вопросительного знака, стоящего после имени файла. Таких пар может быть передано несколько, тогда они отделяются символом **&**, например **script.py?key1=value1&key2=value2**.


get.html

1. Создайте новый документ HTML, содержащий гиперссылки с добавленными значениями для передачи в программу на Python.

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Python Appended Values</title>
</head>
<body>
<h1>
<a href="get.py?make=Ferrari&model=Dino">Ferrari</a>
<a href="get.py?make=Fiat&model=Topolino">Fiat</a>
<a href="get.py?make=Ford&model=Mustang">Ford</a>
</h1>
</body>
</html>
```


get.py

2. Теперь начните новую программу на Python, сделав доступными функции модуля CGI, и создайте объект данных **FieldStorage**.

```
import cgi
data = cgi.FieldStorage()
```

3. Затем присвойте двум переменным переданные значения, указав их имена в качестве ключей.

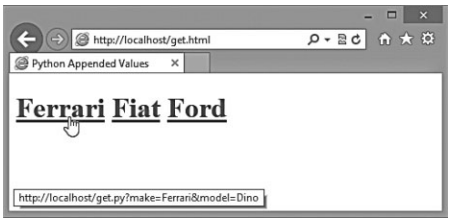
```
make = data.getvalue( 'make' )
model = data.getvalue( 'model' )
```

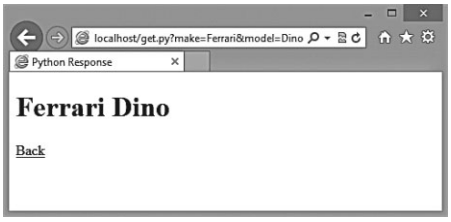
4. Теперь добавьте инструкции для вывода всей веб-страницы HTML, включив переданные значения в вывод.

```
print( 'Content-type:text/html\r\n\r\n' )
print( '<!DOCTYPE HTML>' )
print( '<html lang="en">' )
print( '<head>' )
print( '<meta charset="UTF-8">' )
print( '<title>Python Response</title>' )
print( '</head>' )
print( '<body>' )
print( '<h1>' , make , model , '</h1>' )
print( '<a href="get.html">Back</a>' )
print( '</body>' )
print( '</html>' )
```

5. Наконец сохраните оба файла в каталоге веб-сервера **/htdocs**.

6. Откройте веб-браузер и загрузите HTML-документ, затем щелкните по любой из гиперссылок. Вы увидите переданные значения в ответе.




**Внимание**

Строка запроса в методе GET ограничена 1024 символами, поэтому не подходит для передачи большого количества пар ключ:значение.

**На заметку**

Значения, добавляемые к URL-адресу, видны в адресной строке браузера, поэтому метод GET не следует использовать для отправки веб-серверу паролей либо других конфиденциальных данных.

# Передача данных через формы

Передача данных с веб-страницы на веб-сервер с помощью метода GET с добавлением к URL пар ключ:значение достаточна проста, но при этом имеет некоторые ограничения — длина строки запроса не может превышать 1024 символа, а передаваемые значения отображаются в адресной строке браузера. Существует альтернативный и более надежный способ, при помощи которого браузер может отправлять данные в программу без добавления какой-либо информации к URL-адресу — это использование метода POST, который пересылает информацию на веб-сервер в виде отдельного сообщения.

Для обработки данных формы, отправленных из браузера с помощью метода POST, также может быть использован модуль Python **cgi**, как и в случае с передачей методом GET. Конструктор **FieldStorage()** этого модуля создает объект, в котором данные, отправленные из формы, сохраняются в виде словаря, содержащего пары ключ:значение для каждого поля формы. После этого любое отдельное значение может быть получено с помощью метода **GetValue()** объекта **FieldStorage()** путем указания в качестве аргумента этому методу соответствующего имени ключа.


post.html

1. Создайте новый документ HTML, в котором находится форма с двумя текстовыми полями, содержащими значения по умолчанию, и с кнопкой отправки **Submit**, с помощью которой все значения формы отправляются в программу на Python.

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Python Form Values</title>
</head>
<body>
<form method="POST" action="post.py">
Make: <input type="text" name="make" value="Ford">
Model:
<input type="text" name="model" value="Mustang">
<input type="submit" value="Submit">
</form>
</body>
</html>
```


post.py

2. Затем создайте новую программу на Python, в которой сначала сделайте доступными функции модуля **cgi**, а потом создайте объект данных **FieldStorage**.

```
import cgi
data = cgi.FieldStorage()
```

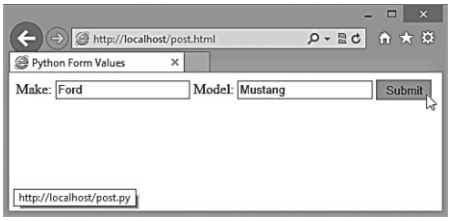
3. Теперь присвойте двум переменным переданные из формы значения, указав в качестве аргумента соответствующие имена, являющиеся ключами для метода `getvalue`.

```
make = data.getvalue( 'make' )
model = data.getvalue( 'model' )
```

4. Затем добавьте инструкции для вывода всех строк страницы HTML, включив в этот вывод также и переданные значения.

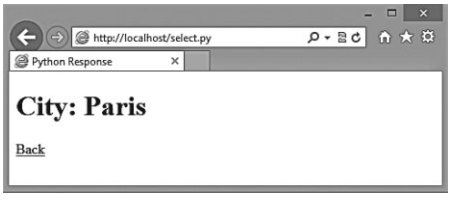
```
print( 'Content-type:text/html\r\n\r\n' )
print( '<!DOCTYPE HTML>' )
print( '<html lang="en">' )
print( '<head>' )
print( '<meta charset="UTF-8">' )
print( '<title>Python Response</title>' )
print( '</head>' )
print( '<body>' )
print( '<h1>' , make , model , '</h1>' )
print( '<a href="post.html">Back</a>' )
print( '</body>' )
print( '</html>' )
```

5. Наконец сохраните оба файла в каталоге */htdocs* вашего веб-сервера.





6. Откройте веб-браузер, загрузите документ HTML и нажмите кнопку **Submit** — вы увидите ответ сервера, выводящий переданные из формы значения.




**На заметку**

Все документы HTML в этой главе должны быть загружены в браузер с веб-сервера (в нашем случае с **localhost)** — открытие их напрямую не даст вам возможности увидеть, как работают примеры.

**Совет**

Нажмите кнопку **Назад** (Back) в браузере и измените значение текстовых полей, а затем снова нажмите кнопку **Submit**, и вы увидите в ответе ваши заново введенные значения.

# Использование текстовых областей

Для того чтобы передать с веб-страницы на веб-сервер большое количество текстовых данных, введенных пользователем, в дополнение к формам и методу POST применяется HTML-элемент **textarea**. Данный элемент не имеет атрибута **value**, следовательно, значение по умолчанию для него не может быть определено. По этой причине в программу на Python полезно добавлять проверку, является ли текстовая область пустой, и задавать значение для нее по умолчанию в случае, когда пользователь не ввел туда ни одного символа.


text.html

1. Создайте новый документ HTML, в котором находится форма с полем для текстовой области и с кнопкой отправки **Submit**.

```
<!DOCTYPE HTML>

<html lang="en">

<head> <meta charset="UTF-8">

<title>Text Area Example</title> </head>

<body>

<form method="POST" action="text.py">

<textarea name="Future Web" rows="5" cols="40">

</textarea>

<input type="submit" value="Submit">

</form>

</body>

</html>
```


text.py

2. Затем создайте новую программу на Python, в которой сделайте сначала доступными функции модуля **cgi**, а потом создайте объект данных **FieldStorage**.

```
import cgi

data = cgi.FieldStorage()
```

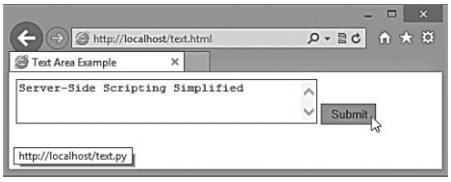
3. Теперь проверьте, является ли текстовая область пустой — если да, то в переменную **text** занесите значение по умолчанию, то есть строку с предупреждением, в противном случае — содержимое этой области.

```
if data.getvalue( 'Future Web' ) :

text = data.getvalue( 'Future Web' )

else :

text = 'Please Enter Text!'
```

4. Затем добавьте инструкции для вывода всех строк страницы HTML, включив в этот вывод также значение переменной **text**, переданное из формы или установленное по умолчанию в программе.

```
print( 'Content-type:text/html\r\n\r\n' )

print( '<!DOCTYPE HTML>' )

print( '<html lang="en">' )

print( '<head> <meta charset="UTF-8">' )

print( '<title>Python Response</title> </head>' )

print( '<body>' )

print( '<h1>' , text , '</h1>' )

print( '<a href="text.html">Back</a>' )

print( '</body>' )

print( '</html>' )
```

5. Наконец сохраните оба файла в каталоге */htdocs* вашего веб-сервера, загрузите документ HTML и нажмите кнопку **Submit** — вы увидите ответ сервера, выводящий строки, переданные из текстовой области.




6. Используя встроенный в браузер инструмент разработчика, проверьте составные части HTTP-запроса и HTTP-ответа, и убедитесь, что текстовая строка была передана в отдельном сообщении в теле запроса (HTTP Request body).


**На заметку**

Ширина символа в различных браузерах может варьироваться, поэтому фактический размер поля текстовой области в зависимости от используемого браузера может быть различным.

**Совет**

Для вызова инструмента разработчика в браузере Internet Explorer можете воспользоваться функциональной клавишей **F12**.

# Установка флажков

В HTML-формах существует такой элемент графического интерфейса, как флажок (или флаговая кнопка). Пользователь может переключать его состояния между «включено» и «выключено», тем самым определяя, добавлять или нет связанные с флажком данные в отправку на веб-сервер. В программе на Python, которой предстоит обрабатывать данные из формы, можно определить установку каждого из флажков, просто проверив, получено ли значение от флажка с определенным именем.


1. Создайте новый документ HTML, содержащий форму с тремя флажками, с которыми связаны определенные значения, и кнопку отправки **Submit**, с помощью которой в программу на Python передаются значения только тех флажков, которые установлены.

check.html

```
<!DOCTYPE HTML>
<html lang="en">
<head> <meta charset="UTF-8">
<title>Checkbox Example</title> </head>
<body>
<form method="POST" action="check.py">
Sailing: <input type="checkbox" name="sail" value="Sailing">
Walking: <input type="checkbox" name="walk" value="Walking">
Ski-ing: <input type="checkbox" name="skee" value="Ski-ing">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

2. Затем создайте новую программу на Python, в которой сначала сделайте доступными функции модуля `cgi`, а потом создайте объект данных `FieldStorage`.

```
import cgi
data = cgi.FieldStorage()
```


3. Теперь занесите в переменную `list` все значения помеченных флажков, получив таким образом в этой переменной неупорядоченный HTML-список.

check.py

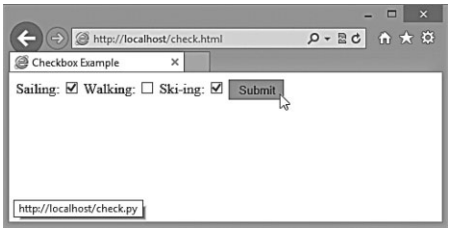
```
list = '<ul>'
if data.getvalue( 'sail' ) :
list += '<li>' + data.getvalue( 'sail' )
if data.getvalue( 'walk' ) :
```

```
list += '<li>' + data.getvalue( 'walk' )
if data.getvalue( 'skee' ) :
list += '<li>' + data.getvalue( 'skee' )
list += '</ul>'
```

4. Затем добавьте инструкции для вывода всех строк страницы HTML, включив в этот вывод также значение переменной `list`, содержащей список переданных значений.

```
print( 'Content-type:text/html\r\n\r\n' )
print( '<!DOCTYPE HTML>' )
print( '<html lang="en">' )
print( '<head>' )
print( '<meta charset="UTF-8">' )
print( '<title>Python Response</title>' )
print( '</head>' )
print( '<body>' )
print( '<h1>' , list , '</h1>' )
print( '<a href="check.html">Back</a>' )
print( '</body>' )
print( '</html>' )
```

5. Наконец сохраните оба файла в каталоге */htdocs* вашего веб-сервера, загрузите документ HTML и нажмите кнопку **Submit**— вы увидите ответ сервера, выводящий значения только установленных флажков.





**Совет**

Для того чтобы заранее пометить какой либо флажок, можно добавить в соответствующий элемент `input` еще один атрибут `checked` с установленным значением `"checked"`.

**На заметку**

Поскольку флажок **Walking** в данном примере не установлен, соответствующая ему пара ключ:значение не отправлена на веб-сервер.

# Установка переключателя в положение

HTML-формы предоставляют такой инструмент, как группа положений переключателя, с помощью которого пользователь может выбрать только один и передать связанные с этим переключателем данные на веб-сервер. Положения переключателя в группе имеют общее имя и, в отличие от флажков, являются взаимоисключающими, то есть, если переключатель установлен в одно из положений в группе, все остальные положения в этой группе выключаются. Программа на Python, которая должна обрабатывать данные формы, может проверять переданные значения от группы положений переключателя и предоставлять соответствующий ответ.


radio.html

1. Создайте документ HTML, содержащий форму с одной группой из трех положений переключателя, а также кнопку **Submit** для отправки значения выбранного положения переключателя в программу на языке Python.

```
<!DOCTYPE HTML>
<html lang="en">
<head> <meta charset="UTF-8">
<title>Radio Button Example</title> </head>
<body>
<form method="POST" action="radio.py">
<fieldset>
<legend>HTML Language Category?</legend> Script
<input type="radio" name="cat" value="Script" checked> Markup
<input type="radio" name="cat" value="Markup">
Program
<input type="radio" name="cat" value="Program">
<input type="submit" value="Submit">
</fieldset>
</form>
</body>
</html>
```

2. Теперь создайте новую программу на Python, сделав доступными в ней функции модуля `cgi`, а затем создайте объект данных `FieldStorage`.

```
import cgi
data = cgi.FieldStorage()
```

3. Затем выполните проверку значения, переданного от группы положений переключателя, и присвойте переменной соответствующий ответ.


radio.py

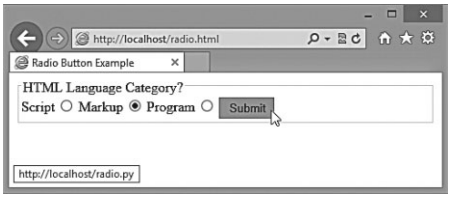
```
answer = data.getvalue( 'cat' )
if answer == 'Markup' :
result = answer + ' Is Correct'
else :
result = answer + ' Is Incorrect'
```

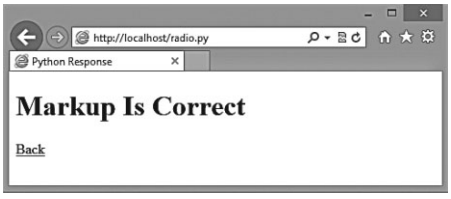
4. Затем добавьте инструкции для вывода всей страницы HTML, включающей переданные значения.

```
print( 'Content-type:text/html\r\n\r\n' )
print( '<!DOCTYPE HTML>' )
print( '<html lang="en">' )
print( '<head>' )
print( '<meta charset="UTF-8">' )
print( '<title>Python Response</title>' )
print( '</head>' )
print( '<body>' )
print( '<h1>' , result , '</h1>' )
print( '<a href="radio.html">Back</a>' )
print( '</body>' )
print( '</html>' )
```

5. Наконец сохраните оба файла в каталоге веб-сервера */htdocs*.

6. Загрузите HTML-документ в браузер, затем установите переключатель в положение, соответствующее верному ответу, и нажмите кнопку **Submit**. Вы увидите ответ от сервера, выводящий соответствующую выбранной кнопке строку.






**Совет**

Всегда используйте атрибут `checked` для того, чтобы автоматически установить выбор по умолчанию одного из положений переключателя.

**На заметку**

Элементы положений переключателя выполнены по принципу старых радиоприемников, в которых каждая выбранная кнопка включала свою радиостанцию, и нельзя было выбрать две или больше.

# Элементы списка

HTML-формы предоставляют еще один инструмент — так называемый выпадающий список, содержащий возможные варианты, из которых пользователь может выбрать один для того чтобы передать связанные с этим элементом данные на сервер. После этого переданное значение можно получить с помощью метода `getvalue()` объекта `FieldStorage`, указав в качестве аргумента этому методу имя соответствующего списка.


select.html

1. Создайте новый HTML-документ, содержащий форму с выпадающим списком выбора и кнопку для передачи данных **Submit**.

```
<!DOCTYPE HTML>
<html lang="en">
<head> <meta charset="UTF-8">
<title>Selection List Example</title> </head>
<body>
<form method="POST" action="select.py">
<select name="CityList">
<option value="New York">New York</option>
<option value="London">London</option>
<option value="Paris">Paris</option>
<option value="Beijing">Beijing</option>
</select>
<input type="submit" value="Submit">
</form>
</body>
</html>
```

2. Теперь создайте новую программу на Python, сделав доступными в ней функции модуля `cgi`, а затем создайте объект данных `FieldStorage`.

```
import cgi
data = cgi.FieldStorage()
```

3. Теперь присвойте значение выбранной опции переменной.

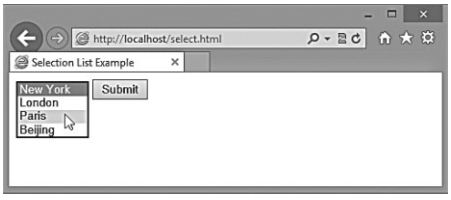
```
city = data.getvalue( 'CityList' )
```

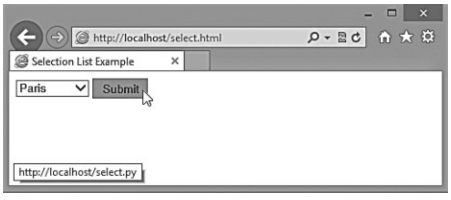
4. Добавьте инструкции для вывода всей страницы HTML, включая переданное значение.


select.py

```
print( 'Content-type:text/html\r\n\r\n' )

print( '<!DOCTYPE HTML>' )

print( '<html lang="en">' )

print( '<head> <meta charset="UTF-8">' )

print( '<title>Python Response</title> </head>' )

print( '<body>' )

print( '<h1>City:' , city , '</h1>' )

print( '<a href="select.html">Back</a>' )

print( '</body>' )

print( '</html>' )
```

5. Наконец сохраните оба файла в каталоге веб-сервера **/htdocs**, загрузите документ HTML в браузер и нажмите кнопку **Submit**, выбрав необходимый элемент списка. Вы увидите в ответе сервера выбранное вами значение.





**На заметку**

Обычно по умолчанию выбранным является первый элемент списка до тех пор, пока вы не прокрутите список и не выберете другой.

**Совет**

Вы можете сделать автоматический выбор одного из элементов списка, включив атрибут `selected` в теге `<option>`. Этим вы добавите выбор элемента по умолчанию.

# Выгрузка файлов

HTML-формы предоставляют средство для выбора файла, которое вызывает стандартное Windows-диалоговое окно, позволяющее пользователю выбрать файл, просматривая каталоги локальной файловой системы. Для того чтобы задействовать данное средство, в HTML-элемент `form` нужно включить атрибут `enctype` и указать тип кодировки `multipart/form-data`.

Полный путь к файлу (или полное имя файла), который выбран для загрузки, — это значение, сохраняющееся в объекте `FieldStorage`, и получить к нему доступ можно, используя соответствующее имя как ключ. С помощью метода `path.basename()` модуля `os` из полного пути разрешается выделить краткое имя файла.

Посредством чтения свойства `file` объекта `FieldStorage` можно записать копию загруженного на веб-сервер файла.


upload.html

1. Создайте HTML-документ, в котором содержится форма с инструментом выбора файла и кнопка для отправки данных **Submit**.

```
<!DOCTYPE HTML>

<html lang="en">

<head> <meta charset="UTF-8">

<title>File Upload Example</title> </head>

<body>

<form method="POST" action="upload.py"

enctype="multipart/form-data" >

<input type="file" name="filename" style="width:400px">

<input type="submit" value="Submit">

</form>

</body>

</html>
```


upload.py

2. Затем начните новую программу на Python, сделав доступными функции модулей `cgi` и `os`, а затем создайте объект данных `FieldStorage`.

```
import cgi , os

data = cgi.FieldStorage()
```

3. Теперь одной переменной присвойте значение полного пути к загруженному файлу, а другой — краткое имя файла.

```
upload = data[ 'filename' ]

filename = os.path.basename( upload.filename )
```

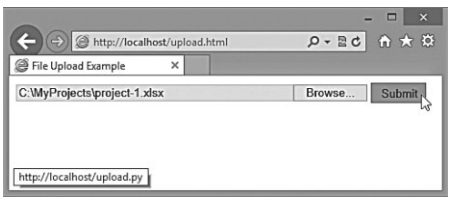
4. Затем запишите копию загруженного на веб-сервер файла.

```
with open( filename , 'wb' ) as copy :

copy.write( upload.file.read() )
```

5. Добавьте инструкции для вывода всей страницы HTML, включив в вывод имя загруженного файла.

```
print( 'Content-type:text/html\r\n\r\n' )

print( '<!DOCTYPE HTML>' )

print( '<html lang="en">' )

print( '<head>' )

print( '<meta charset="UTF-8">' )

print( '<title>Python Response</title>' )

print( '</head>' )

print( '<body>' )

print( '<h1>File Uploaded:' , filename , '</h1>' )

print( '<a href="upload.html">Back</a>' )

print( '</body>' )

print( '</html>' )
```

6. Наконец сохраните оба файла в каталоге сервера **/htdocs**, загрузите документ HTML в браузер, затем выберите файл для загрузки, и вы увидите ответ сервера, отображающий имя выбранного вами файла.





**Совет**

Обратите внимание, что для копирования загруженного файла здесь использовался двоичный режим.

# Заключение

- Интерпретатор Python может быть установлен на хост с работающим веб-сервером, для того чтобы обрабатывать серверные программы перед отправкой ответа веб-браузеру клиента.
- Серверная программа на Python может генерировать документ HTML, обозначая тип содержимого как `Content-type:text/html\r\n\r\n`.
- Модуль `cgi` предоставляет конструктор `FieldStorage()` для создания объекта, в котором сохраняются переданные данные в виде пар ключ:значение.
- К любому значению, хранящемуся в объекте `FieldStorage`, можно обратиться, указав в качестве аргумента методу `getvalue()` соответствующее имя ключа.
- Браузер может отправлять данные в программу, используя метод GET, который добавляет пары `ключ=значение` к URL-адресу после символа `?`.
- Используя метод GET, можно передать несколько пар данных `ключ=значение`, при этом они отделяются символом `&`.
- Строка запроса метода GET не может иметь длину больше, чем 1024 символа, и при этом она отображается в поле адреса браузера.
- Браузер способен отправлять данные в программу, используя метод POST, который передает пары `ключ=значение` в виде отдельного сообщения.
- Переданные из HTML-формы данные могут сохраняться в объекте `FieldStorage` в виде пар ключ:значение для каждого поля формы.
- Серверная программа на Python способна предоставлять значение по умолчанию для переданных полей формы HTML, которые пользователь оставил пустыми.
- Поля флажков HTML-формы, которые не установлены, не передаются на веб-сервер.
- Выбранное в группе положение переключателя при отправке формы передает значение, привязанное к имени этой группы.
- При использовании выпадающего списка выбранный из него элемент передает значение, связанное с именем списка.
- HTML-форма предоставляет возможность загрузки файлов на сервер, если в ее атрибуте `enctype` тип кодирования указан как `multipart/form-data`.

# 9 Разработка интерфейсов

*Данная глава демонстрирует пример разработки на Python приложения с графическим оконным интерфейсом.*

- **Запуск оконного интерфейса**
- **Работа с кнопками**
- **Вывод сообщений**
- **Прием данных от пользователя**
- **Выбор из списка**
- **Использование переключателей**
- **Флажки**
- **Добавление изображений**
- **Заключение**

# Запуск оконного интерфейса

В языке Python модуль, который вы можете использовать для создания графических приложений, называется `tkinter` (toolkit to interface, набор инструментов для интерфейса).

**Внимание**

В программе может присутствовать только один вызов конструктора `Tk()`, и находиться он должен в начале программы.

Чтобы обеспечить атрибуты и методы для создания оконного интерфейса, `tkinter`, как и другие модули, должен быть импортирован в программу на Python. Каждая программа, использующая ресурсы данного модуля, должна начинаться с вызова конструктора `Tk()`, создающего объект окна. При помощи метода `geometry()` этого объекта можно дополнительно указать размеры окна, передав в качестве аргумента строку '*ширинахвысота*'. Можно также создать заголовок для окна, указав методу `title()` строковый аргумент '*заголовок*'. Если размеры и заголовок не указаны, то будут использоваться значения, заданные по умолчанию.

Завершать каждую программу должен так называемый цикл обработки событий окна, вызываемый методом `mainloop()`, который реагирует на действия пользователя с окном в течение исполнения программы (закрытие окна для выхода из программы или просто изменение его размеров).

Все элементы управления графического интерфейса, создаваемые с использованием модуля `tkinter`, такие как кнопки или флажки, называют **виджетами**. Самым простым примером виджета является `Label`, который отображает обычный текст или изображение в интерфейсе приложения. Создается он при помощи конструктора `Label()`, которому в качестве аргументов указывается имя окна и текст для самой метки в виде `text='строка'`.

**Совет**

Работа менеджера `grid()` демонстрируется в примере в следующей главе.

После создания любого виджета его нужно поместить на окно. Для этого в Python существуют специальные методы, называемые **менеджерами размещений**.

- `pack()` — располагает виджеты по указанной стороне окна при помощи параметра `side=` , который может принимать значения четырех констант: `TOP`, `BOTTOM`, `LEFT` и `RIGHT`.
- `place()` — помещает виджет в точку окна с координатами X,Y, указанными ему в виде параметров `x=`, `y=`, заданными числовыми значениями в пикселах.
- `grid()` — позволяет разместить виджет в ячейку таблицы в соответствии с указанными числовыми параметрами `row=`, `column=`, определяющими номер строки и номер столбца этой ячейки.

Метод `pack()` может принимать также дополнительные аргументы, включая `fill`, определяющий возможность виджета заполнять свободное пространство вдоль какой-либо из осей, например вдоль оси x: `fill='x'`, а также аргументы `padx`, `pady`, задающие расширение виджета вдоль соответствующих осей на определенное количество пикселов.

1. Создайте новую программу на Python, в которой сначала сделайте доступными функции модуля `cgi`.

   ```
   from tkinter import *
   ```

2. Затем вызовите конструктор для создания объекта окна.

   ```
   window = Tk()
   ```

3. Теперь задайте заголовок для этого окна.

   ```
   window.title( 'Label Example' )
   ```

4. Добавьте инструкцию с вызовом конструктора, создающего объект `Label`.

   ```
   label = Label( window , text = 'Hello World!' )
   ```

5. Используйте менеджер размещений для добавления метки на окно, указав в параметрах заполнение по горизонтали и по вертикали.

   ```
   label.pack( padx = 200 , pady = 50 )
   ```

6. Наконец добавьте обязательную инструкцию с циклом обработки событий окна.

   ```
   window.mainloop()
   ```

7. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу с помощью команды `python tk_window.py` — появится окно, содержащее виджет `Label`.



`tk_window.py`

**На заметку**

Виджеты не появятся в окне, пока не будут туда добавлены при помощи одного из менеджеров размещения.

# Работа с кнопками

**Совет**

Для того чтобы вызвать функцию, которая должна обрабатывать опцию `command`, можно также использовать метод кнопки `invoke()`.

Виджет `Button` — элемент графического оконного интерфейса «кнопка», который может содержать либо текст, либо изображение, передающие смысловое назначение кнопки. Объект `Button` создается с помощью указания конструктору `Button()` имени окна и опций в качестве аргументов. Каждая опция определяется как пара *`опция=значение`*. Определяющей является опция `command`, в которой указывается имя функции или метода, которые должны быть вызваны, когда пользователь нажимает на кнопку. Наиболее популярные опции и их краткое описание приведены в таблице ниже.

| Опция | Описание |
|---|---|
| `activebackground` | Цвет фона активного элемента |
| `activeforeground` | Цвет переднего плана активного элемента |
| `bd` | Ширина рамки в пикселах (значение по умолчанию: 2) |
| `bg` | Цвет фона |
| `command` | Функция, вызываемая при нажатии |
| `fg` | Цвет переднего плана |
| `font` | Шрифт для метки кнопки |
| `height` | Высота кнопки (для текста в количестве строк, для изображений — в пикселах) |
| `highlightcolor` | Цвет рамки при наведении |
| `image` | Изображение для вывода вместо текста |
| `justify` | Вид выравнивания (по левому краю, по центру, по правому краю) |
| `padx` | Количество пикселов до края по горизонтали |
| `pady` | Количество пикселов до края по вертикали |
| `relief` | Вид рельефности рамки (`SUNKEN` — утопленная, `RIDGE` — выпуклая кайма, `RAISED` — выпуклая, `GROOVE` — канавка) |
| `state` | Состояние (`NORMAL` — рабочее или `DISABLED` — отключена) |
| `underline` | Порядковый номер символа в тексте, который необходимо подчеркнуть (значение по умолчанию: 1) |
| `width` | Ширина кнопки (в символах для текста, в пикселах для изображения) |
| `wraplength` | Параметр, определяющий ширину, в которую вписывается текст |

Значения, присвоенные различным опциям, определяют внешний вид виджета. Опции разрешается изменять с помощью метода виджета `configure()`, указав ему в качестве аргумента новое значение пары *`опция=значение`*. Кроме того, можно получить значение каждой опции с помощью метода виджета `cget()`, указав в качестве строкового аргумента имя определенной опции.

1. Начните новую программу на Python, сделав доступными в ней функции модуля `gui`, и создайте окно, определив заголовок.

`tk_button.py`

```
from tkinter import *

window = Tk()

window.title( 'Button Example' )
```

2. Теперь создайте кнопку, по нажатию которой будет происходить выход из программы.

```
btn_end = Button( window , text = 'Close' , command=exit )
```

3. Затем добавьте функцию, которая станет переключать цвет фона окна после нажатия еще одной кнопки.

```
def tog() :

if window.cget( 'bg' ) == 'yellow' :

window.configure( bg = 'gray' )

else :

window.configure( bg = 'yellow' )
```

4. Теперь создайте кнопку, по нажатию которой будет вызываться функция.

```
btn_tog = Button( window , text = 'Switch' , command=tog )
```

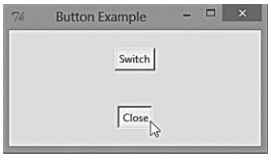
5. Добавьте обе кнопки на окно, указав параметры заполнения по горизонтали и вертикали.

```
btn_end.pack( padx = 150 , pady = 20 )

btn_tog.pack( padx = 150 , pady = 20 )
```

6. Наконец добавьте цикл обработки событий окна.

```
window.mainloop()
```

7. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу с помощью команды `python tk_button.py` — нажмите кнопку, чтобы посмотреть результат изменения цвета фона окна.


**Внимание**

В опции `command` указывается только имя функции без добавления скобок.

**Совет**

Цвет окна по умолчанию является серым (`gray`).

# Вывод сообщений

В программе на Python можно выводить сообщения пользователю в виде диалоговых окон с помощью методов, представляемых модулем `tkinter.messagebox`. Данный модуль должен быть импортирован отдельно, а вместо его длинного имени допускается назначить псевдоним с помощью инструкции `import as`.

Диалоговое окно может быть создано с помощью указания заголовка окна, а также самого сообщения в виде двух аргументов для одного из следующих методов.

| Метод | Значок | Кнопки |
|---|---|---|
| `showinfo()` |  | **OK** |
| `showwarning()` |  | **OK** |
| `showerror()` |  | **OK** |
| `askquestion()` |  | **Yes** (возвращает строку `'yes'`) и **No** (возвращает строку `'no'`) |
| `askokcancel()` |  | **OK** (возвращает 1) и **Cancel** |
| `askyesno()` |  | **Yes** (возвращает 1) и **No** |
| `askretrycancel()` |  | **Retry** (возвращает 1) и **Cancel** |

**Совет**

Только метод `askquestion()` возвращает два значения. Кнопка **No** метода `askyesno()`, а также **Cancel** не возвращают ничего.

Методы, которые выводят диалоговое окно, содержащее единственную кнопку **OK**, не возвращают никакого значения при ее нажатии пользователем. В тех методах, где возвращается значение, можно использовать данное возвращаемое значение для дальнейшего условного ветвления.


`tk_message.py`

1. Начните новую программу на Python, сделав доступными функции модуля `gui`, а также модуля диалоговых окон, который обозначьте кратким псевдонимом `box`.

```
from tkinter import *

import tkinter.messagebox as box
```

2. Затем создайте объект окна и укажите заголовок для него.

```
window = Tk()

window.title( 'Button Example' )
```

3. Добавьте функцию для вывода различных диалоговых окон.

```
def dialog() :

var = box.askyesno( 'Message Box' , 'Proceed?' )

if var == 1 :

box.showinfo( 'Yes Box', 'Proceeding...' )

else :

box.showwarning( 'No Box', 'Cancelling...' )
```

4. Затем создайте кнопку, при нажатии которой будет вызываться функция.

```
btn = Button( window , text = 'Click' , command=dialog )
```

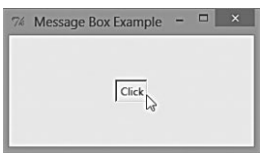
5. Добавьте кнопку на окно, указав параметры заполнения.

```
btn.pack( padx = 150 , pady = 50 )
```

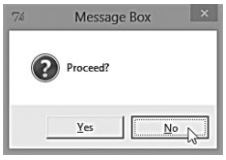
6. Наконец добавьте цикл обработки событий окна.

```
window.mainloop()
```

7. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу с помощью команды `python tk_message.py` — нажмите кнопку, чтобы увидеть появляющиеся диалоговые окна.







**Совет**

Для всех этих методов можно добавить дополнительную опцию в качестве третьего аргумента. Например, чтобы получить три кнопки, добавьте `type='abortretryignore'`.

# Прием данных от пользователя

Для получения данных, введенным пользователем в приложении с графическим интерфейсом, в модуле `tkinter` существует виджет `Entry`, который предоставляет однострочное поле ввода. Объект ввода создается при помощи конструктора `Entry()` указанием ему в качестве аргументов имени родительского контейнера (например, окна или фрейма), а также используемых опций, каждая из которых передается в виде пары *`опция=значение`*. Наиболее применяемые опции и их краткое описание представлены в таблице ниже:

| Опция | Описание |
|---|---|
| `bd` | Ширина рамки в пикселах (значение по умолчанию: 2) |
| `bg` | Цвет фона |
| `fg` | Цвет переднего плана |
| `font` | Шрифт для текста |
| `highlightcolor` | Цвет рамки при наведении |
| `selectbackground` | Цвет фона выделенного текста |
| `selectforeground` | Цвет переднего плана выделенного текста |
| `show` | Использовать вместо видимых символов маскирующие |
| `state` | Состояние (`NORMAL` — рабочее или `DISABLED` — отключена) |
| `width` | Ширина поля ввода в символах |

**Совет**

Если вам необходимо, чтобы пользователь имел возможность вводить не одну строку, а многострочный текст, вместо `Entry` используйте виджет `Text`.

В целях оптимизации размещения несколько виджетов можно сгруппировать во фреймы. Объект фрейм создается при помощи конструктора `Frame()` указанием ему в качестве аргумента имени окна. После этого имя фрейма может быть передано первым аргументом конструктору виджета, чтобы указать, что данный фрейм является для виджета контейнером. При добавлении виджета на фрейм вы можете указывать его привязку к определенной стороне фрейма, используя константы `TOP`, `BOTTOM`, `LEFT` и `RIGHT`. Например, `entry.pack( side=LEFT )`.

Как правило, виджет Entry для ввода текста размещают рядом с меткой, в которой описывается, что должен вводить пользователь, или рядом с кнопкой, которую пользователь может нажать, чтобы выполнить какие-то действия над введенными им данными. Поэтому размещение виджетов в одном фрейме является оптимальным вариантом.

Данные, введенные пользователем в виджете **Entry**, можно получить в программе, используя метод виджета **get()**.

1. Создайте новую программу на Python, в которой сделайте сначала доступными функции модуля **cgi**, а затем подключите модуль для работы с диалоговыми окнами, указав для него псевдоним.


tk_entry.py

```
from tkinter import *

import tkinter.messagebox as box
```

2. Затем создайте объект окна, указав строку для его заголовка.

```
window = Tk()

window.title( 'Entry Example' )
```

3. Теперь создайте фрейм, в который будет помещено поле для ввода.

```
frame = Frame( window )

entry = Entry( frame )
```

4. Добавьте функцию для отображения данных из поля ввода.

```
def dialog() :

box.showinfo( 'Greetings' , 'Welcome ' + entry.get() )
```

5. Теперь создайте кнопку, при нажатии на которую будет вызываться функция.

```
btn = Button( frame, text = 'Enter Name' , command=dialog )
```

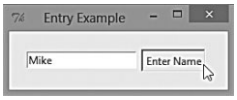
6. Добавьте кнопку и поле ввода на фрейм, указав параметры их расположения на фрейме.

```
btn.pack( side = RIGHT , padx = 5 )

entry.pack( side = LEFT )

frame.pack( padx = 20 , pady = 20 )
```

7. Наконец, добавьте цикл обработки событий окна.

```
window.mainloop()
```

8. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу с помощью команды **python tk_entry.py** — теперь введите ваше имя в поле для ввода, нажмите кнопку, и вы увидите, как появится приветственное сообщение.





**Совет**

Если вам необходимо отобразить текст, который пользователю нельзя редактировать, вместо **Entry** используйте виджет **Label**.

**На заметку**

В модуле `tkinter` полоса прокрутки организуется при помощи отдельного виджета `scrollbar`, который необходимо привязывать к виджетам `Listbox`, `Text`, `Canvas` и `Entry`.

# Выбор из списка

При помощи виджета `Listbox` вы можете добавлять в приложение список элементов, предлагаемых пользователю для выбора. Для создания объекта `listbox` используется конструктор `Listbox()`, которому в качестве аргументов указываются имя родительского контейнера (например, окна или фрейма) и возможные опции, самые популярные из которых представлены вместе с кратким описанием в таблице ниже.

| Опция | Описание |
|---|---|
| `bd` | Ширина рамки в пикселах (значение по умолчанию: 2) |
| `bg` | Цвет фона |
| `fg` | Цвет переднего плана |
| `font` | Шрифт для текста |
| `height` | Количество строк в списке (значение по умолчанию: 10) |
| `selectbackground` | Цвет фона выделенного текста |
| `selectmode` | Режим выбора (`SINGLE` — одиночный по умолчанию или `MULTIPLE` — множественный) |
| `width` | Ширина поля ввода списка в символах (значение по умолчанию: 20) |
| `yscrollcommand` | Привязка к полосе прокрутки |

Элементы добавляются в список при помощи метода `Insert()`, которому в качестве аргументов указывается порядковый номер элемента в списке и строка, определяющая сам элемент. Любой элемент списка можно получить с помощью метода `get()`, указав его порядковый номер в качестве аргумента. Кроме того, у объекта `listbox` существует полезный метод `curselection()`, возвращающий порядковый номер текущего выбранного элемента, так что его можно использовать как аргумент для метода `get()`, чтобы получить текущий выбор.


`tk_listbox.py`

1. Создайте новую программу на Python, в которой сначала сделайте доступными функции модуля `cgi`, а затем подключите модуль для работы с диалоговыми окнами, указав для него псевдоним.

```
from tkinter import *
import tkinter.messagebox as box
```

2. Затем создайте объект окна, указав для него заголовок.

```
window = Tk()
window.title( 'Listbox Example' )
```

3. Теперь создайте фрейм, в который будут помещены виджеты.

```
frame = Frame( window )
```

4. Создайте виджет **Listbox** с тремя элементами списка.

```
listbox = Listbox( frame )
listbox.insert( 1 , 'HTML5 in easy steps' )
listbox.insert( 2 , 'CSS3 in easy steps' )
listbox.insert( 3 , 'JavaScript in easy steps' )
```

5. Затем добавьте функцию для вывода выбранных элементов списка.

```
def dialog() :
box.showinfo( 'Selection' , 'Your Choice: ' + \
listbox.get( listbox.curselection() ) )
```

6. Теперь создайте кнопку, при нажатии на которую будет вызываться функция.

```
btn = Button( frame, text = 'Choose' , command = dialog )
```

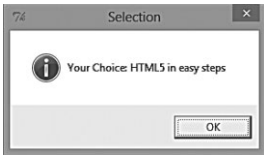
7. Поместите кнопку со списком на фрейм, указав параметры размещения.

```
btn.pack( side = RIGHT , padx = 5 )
listbox.pack( side = LEFT )
frame.pack( padx = 30 , pady = 30 )
```

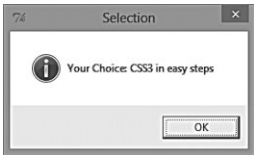
8. Наконец добавьте цикл обработки событий окна

```
window.mainloop()
```

9. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу с помощью команды **python tk_listbox.py** — теперь выберите строку из списка, нажмите кнопку и вы увидите, как появится информационное сообщение, подтверждающее ваш выбор.






**Внимание**

Если дополнительная опция **selectmode** (режим выбора) установлена в значение **MULTIPLE** (множественный выбор), то метод **curselection()** возвращает кортеж из порядковых номеров выбранных элементов.

# Использование переключателей

Виджет **Radiobutton** позволяет добавить в графическое приложение один элемент положения переключателя, в которое тот может быть установлен пользователем. Если используется группа таких положений, то пользователь имеет возможность выбрать из этой группы только одно. Объекты положений переключателя группируются вместе, когда необходимо задать одну управляющую переменную и присвоить ей значение в зависимости от выбора пользователя. Для создания такой переменной-объекта применяется конструктор **StringVar()**, создающий пустую строку, или конструктор **IntVar()**, инициализирующий пустую целочисленную переменную-объект.

**Внимание**

Для хранения значений, переданных от выбранного положения переключателя, используются не обычные переменные, а специальные объекты, выполняющие роль переменных.

Объект положения переключателя создается при помощи конструктора **Radiobutton()**, которому передаются четыре аргумента:

- имя родительского контейнера, например фрейма;
- текстовая строка, которая будет являться меткой, передаваемая в виде пары **text='*текст*'**;
- управляющая переменная-объект, указываемая в виде пары **variable=*имя-переменной***;
- значение для присваивания, передаваемое в виде пары **value=*значение***.

Каждый объект положения переключателя содержит метод **select()**, который может быть использован, чтобы определить, какая кнопка в группе выбрана по умолчанию при запуске программы. Строковое значение, которое присвоилось в результате выбора переключателя, может быть получено из переменной-объекта методом **get()**.


**tk_radio.py**

1. Создайте новую программу на Python, в которой сделайте доступными функции модуля **cgi**, а затем подключите модуль для работы с диалоговыми окнами, указав для него псевдоним.

```
from tkinter import *
import tkinter.messagebox as box
```

2. Затем создайте объект окна, указав для него заголовок.

```
window = Tk()
window.title( 'Radio Button Example' )
```

3. Теперь создайте фрейм, в который будут помещены виджеты.

```
frame = Frame( window )
```

4. Затем создайте строковую переменную-объект, где станет храниться результат выбора.

```
book = StringVar()
```

5. Создайте три виджета положения переключателя, значения которых при выборе будут заноситься в переменную-объект.

```
radio_1 = Radiobutton( frame , text = 'HTML5' , \
variable = book , value = 'HTML5 in easy steps' )
radio_2 = Radiobutton( frame , text = 'CSS3' , \
variable = book , value = 'CSS3 in easy steps' )
radio_3 = Radiobutton( frame , text = 'JS' , \
variable = book , value = 'JavaScript in easy steps' )
```

6. Теперь добавьте инструкцию, определяющую, в какое из положений по умолчанию будет установлен переключатель при запуске программы.

```
Radio_1.select()
```

7. Затем добавьте функцию, отображающую текущий выбор пользователя, а также кнопку, по нажатию которой будет вызываться данная функция.

```
def dialog() :
box.showinfo( 'Selection' , \
'Your Choice: \n' + book.get() )
btn = Button( frame, text = 'Choose' , command = dialog )
```

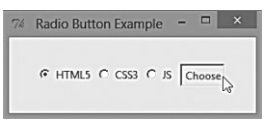
8. Теперь поместите кнопку с тремя положениями переключателя на фрейм, указав параметры размещения.

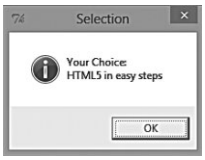
```
btn.pack( side = RIGHT , padx = 5 )
radio_1.pack( side = LEFT )
radio_2.pack( side = LEFT )
radio_3.pack( side = LEFT )
frame.pack( padx = 30 , pady = 30 )
```

9. Наконец добавьте цикл обработки событий окна.

```
window.mainloop()
```

10. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу с помощью команды `python tk_radio.py` — теперь установите переключатель в одно из положений, нажмите кнопку, и вы увидите, как появится информационное сообщение, подтверждающее ваш выбор.




**Совет**

Существует возможность сбросить выделение элементов группы положений переключателя в программе — для этого используется метод `deselect()` объекта `Radiobutton`.

# Флажки

Виджет **Checkbutton** обеспечивает добавление в графическое приложение одного элемента «флажок», который может быть установлен (выбран) пользователем. Обычно используется группа таких флажков, и пользователь имеет возможность выбрать из этой группы один или несколько. Объекты флажков задают каждый свою управляющую переменную-объект, и ей присваивается значение в соответствии с тем, отмечен флажок пользователем или нет. Для создания такой переменной-объекта применяется конструктор **StringVar()**, создающий пустую строку, или конструктор **IntVar()**, инициализирующий пустую целочисленную переменную-объект.

Объект флажка создается при помощи конструктора **Checkbutton()**, которому передаются пять аргументов.

- Имя родительского контейнера, например фрейма.
- Текстовая строка, которая будет являться меткой, передаваемая в виде пары **text='*текст*'**.
- Управляющая переменная-объект, указываемая в виде пары **variable=*имя-переменной***.
- Значение для присваивания, в случае если флажок установлен пользователем, передаваемое в виде пары **onvalue=*значение***.
- Значение для присваивания, в случае если флажок пользователем не установлен (сброшен), передаваемое в виде пары **offvalue=*значение***.

Целочисленное значение, которое присвоено в результате установки флажка, может быть получено из переменной-объекта методом **get()**.


tk_check.py

1. Создайте новую программу на Python, в которой сделайте сначала доступными функции модуля **cgi**, а затем подключите модуль для работы с диалоговыми окнами, указав для него псевдоним.

```
from tkinter import *
import tkinter.messagebox as box
```

2. Затем создайте объект окна, указав для него строку заголовка.

```
window = Tk()
window.title( 'Check Button Example' )
```

3. Теперь создайте фрейм, в который будут помещены виджеты.

```
frame = Frame( window )
```

4. Затем создайте три целочисленных переменных-объекта, где будут храниться значения в зависимости от результата выбора пользователя.

```
var_1 = IntVar()
var_2 = IntVar()
var_3 = IntVar()
```

5. Создайте три виджета флажков, значения которых будут заноситься в переменные-объекты.

```
book_1 = Checkbutton( frame , text = 'HTML5' , \
variable = var_1 , onvalue = 1, offvalue = 0 )
book_2 = Checkbutton( frame , text = 'CSS3' , \
variable = var_2 , onvalue = 1, offvalue = 0 )
book_3 = Checkbutton( frame , text = 'JS' , \
variable = var_3 , onvalue = 1, offvalue = 0 )
```

6. Теперь добавьте функцию, отображающую текущий выбор пользователя.

```
def dialog() :
str = 'Your Choice:'
if var_1.get() == 1 : str += '\nHTML5 in easy steps'
if var_2.get() == 1 : str += '\nCSS3 in easy steps'
if var_3.get() == 1 : str += '\nJavaScript in easy steps'
box.showinfo( 'Selection' , str )
```

7. Затем создайте кнопку, при нажатии на которую будет вызываться функция.

```
btn = Button( frame, text = 'Choose' , command = dialog )
```

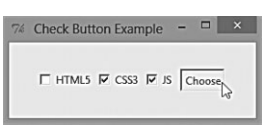
8. Теперь поместите кнопку с тремя флажками на фрейм, указав параметры размещения.

```
btn.pack( side = RIGHT , padx = 5 )
book_1.pack( side = LEFT )
book_2.pack( side = LEFT )
book_3.pack( side = LEFT )
frame.pack( padx = 30 , pady = 30 )
```

9. Наконец добавьте цикл обработки событий окна.

```
window.mainloop()
```

10. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу с помощью команды `python tk_check.py` — теперь пометьте один или несколько флажков, нажмите кнопку, и вы увидите, как появится информационное сообщение, подтверждающее ваш выбор.





**Совет**

Объект `Checkbutton` содержит методы `select()` и `deselect()`, которые могут использоваться для переключения состояния флажка, например `check_1.select()`.

**Совет**

Состояние объекта `Checkbutton` можно поменять на противоположное, используя метод объекта `toggle()`.

# Добавление изображений

Модуль `tkinter` позволяет работать с файлами изображений в форматах GIF или PGM/PPM, которые могут быть выведены на виджетах `Label`, `Text`, `Button` или `Canvas`. Для этих целей используется конструктор `Photoimage()`, который создает объект изображения. Достаточно указать ему в качестве аргумента имя нужного файла с изображением в виде `file = '`*`имя-файла`*`'`. Можно воспользоваться полезным методом `subsample()` для уменьшения изображения, указав в качестве аргументов параметры дискретизации по горизонтали и вертикали в виде `x=`*`значение`* и `y=`*`значение`*. Например, указанные значения `x=2`, `y=2` приведут к отбрасыванию каждого второго пиксела, то есть изображение уменьшится наполовину по отношению к оригиналу.

**Совет**

В классе `Photoimage` существует и обратный метод `zoom()`, увеличивающий размер изображения в соответствии с заданными аргументами `x` и `y`.

После того как объект изображения создан, его можно добавлять на виджеты `Label` или `Button` при помощи указания опции `image=` в соответствующих конструкторах.

У объектов виджетов `Text` существует метод `image_create()`, при помощи которого изображение встраивается в текстовое поле. Данный метод принимает два аргумента: первый — для определения позиции размещения (например, `'1.0'` указывает первую строку и первый символ), второй — непосредственная ссылка на само изображение в виде опции `image=`.

Объекты Canvas имеют аналогичный метод `create_image()`, принимающий тоже два аргумента, только первый из них, отвечающий за расположение, представлен в виде пары координат (`x`,`y`), которые определяют точку на холсте (элементе Canvas), куда помещается изображение.




`tk_image.py`

1. Начните новую программу на Python, сделав доступными методы и атрибуты модуля `cgi`, а затем создайте объект окна, указав строку для его заголовка.

```
from tkinter import *

window = Tk()

window.title( 'Image Example' )
```


`python.gif`
`(200 x 200)`

2. Затем создайте объект изображения из файла локальной системы.

```
img = PhotoImage(file = 'python.gif')
```

3. Теперь создайте объект Label для вывода картинки поверх желтого фона метки.

```
label = Label( window , image = img , bg = 'yellow')
```

4. Затем создайте еще один объект изображения, содержащий уменьшенную в два раза первоначальную картинку.

```
small_img = PhotoImage.subsample(img , x = 2 , y = 2 )
```

5. Создайте кнопку для вывода уменьшенного изображения.

```
btn = Button( window , image = small_img )
```

6. Теперь создайте текстовое поле, встройте в него уменьшенное изображение, а затем добавьте после него строку текста.

```
txt = Text( window , width = 25 , height = 7 )

txt.image_create( '1.0' , image = small_img )

txt.insert( '1.1', 'Python Fun!' )
```

7. Создайте элемент холста и поместите туда второе изображение, а затем диагональную линию поверх него.

```
can = \

Canvas( window , width = 100 , height = 100 , bg = 'cyan' )

can.create_image( ( 50 , 50 ), image = small_img )

can.create_line( 0 , 0 , 100 , 100, width = 25 , fill = 'yellow' )
```

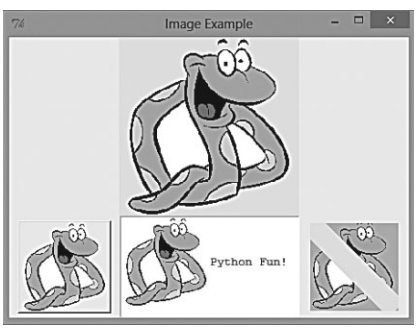
8. Теперь добавьте виджеты на окно.

```
label.pack( side = TOP )

btn.pack( side = LEFT , padx = 10 )

txt.pack( side = LEFT )

can.pack( side = LEFT, padx = 10 )
```

9. Наконец добавьте цикл обработки событий окна.

```
window.mainloop()
```

10. Сохраните файл в вашем рабочем каталоге, откройте командную строку и запустите программу с помощью команды `python tk_photo.py`, чтобы наблюдать результат работы с изображениями.




**Внимание**

Обратите внимание, что методы `image_create()` класса `Text` и `create_image()` класса `Canvas`, хотя и очень похожи, все-таки отличаются. Не путайте их.

**Совет**

На странице **docs.python.org/3.3/library/tkinter.html** вы можете познакомиться ближе с такими мощными и гибкими инструментами разработки графического интерфейса, как виджеты `Canvas` и `Text`.

# Заключение

- Для разработки приложений с оконными интерфейсами в Python используется модуль `tkinter`, методы и атрибуты которого можно импортировать в любую программу.
- Каждая программа, работающая с модулем `tkinter`, должна начинаться с вызова конструктора `Tk()`, создающего окно, а также содержать метод обработки событий окна `mainloop()`.
- При помощи метода объекта окна `title()` вы можете указать заголовок для окна.
- Виджет `Label` (метка) создается при помощи конструктора `Label()`, которому в качестве аргументов указываются имя родительского контейнера и текст, образующий метку.
- Виджеты добавляются в приложение при помощи менеджеров размещения — методов `pack()`, `grid()` и `place()`.
- Виджет `Button` (кнопка) создается при помощи конструктора `Button()`, которому в качестве аргументов передаются имя родительского контейнера, текст, помещаемый на кнопку, и имя функции, вызываемой при нажатии.
- Для создания диалоговых окон используются методы и атрибуты импортируемого в программу на Python модуля `tkinter.messagebox`.
- Диалоговые окна, запрашивающие ответ пользователя, возвращают в программу значения, которые используются далее для условного ветвления.
- Для удобства позиционирования виджетов их можно группировать в специальные контейнеры — фреймы, создаваемые при помощи конструктора `Frame()`.
- Конструктор `Entry()` создает однострочное поле для ввода, текущее содержимое которого можно получить при помощи метода `get()`.
- Добавление элементов в список, представленный объектом `Listbox`, осуществляется при помощи метода `insert()`, а вызвать нужный элемент списка можно, указав его порядковый номер в качестве аргумента методу `get()`.
- Объекты `Radiobutton` (переключатель) и `Checkbutton` (флажок) передают значения в специальной переменной-объекте `variable`, принадлежащей одному из классов `StringVar` или `IntVar`.
- Объекты изображений создаются при помощи конструктора `Photoimage()` и имеют полезный инструмент для масштабирования в виде метода `subsample()`.
- Изображения можно добавлять на такие объекты, как кнопка, метка, текстовая область или холст.

# 10 Разработка приложений

*В этой главе рассказывается о том, как построить готовое к работе приложение, собрав воедино все элементы, описанные в предыдущих главах.*

- **Генерирование случайных чисел**
- **Подготовка плана программы**
- **Построение интерфейса**
- **Определение постоянных величин**
- **Инициализация изменяемых значений**
- **Добавление рабочей функциональности**
- **Тестирование программы**
- **Компиляция программы**
- **Распространение приложения**
- **Заключение**

# Генерирование случайных чисел

Графическое приложение, рассматриваемое далее на страницах книги, будет генерировать шесть случайных чисел из определенного диапазона. Первоначально данный функционал мы реализуем при помощи консольного приложения, а потом продемонстрируем, как его можно применить для работы компонентов приложения с графическим интерфейсом.

В стандартной библиотеке языка Python содержится модуль **random**, который обеспечивает методы для генерирования псевдослучайных чисел. Для настройки генератора случайных чисел используется системное время, поэтому при его инициализации все время будет порождаться новая последовательность.

Для того чтобы сгенерировать псевдослучайное число с плавающей точкой в диапазоне от 0.0 до 1.0, используется метод **random()** модуля **random**. Диапазон генерируемых значений можно увеличить, применив оператор умножения, *, а если требуется в результате получить целые числа, то придется воспользоваться преобразованием типов при помощи встроенной функции **int()**. Например, если вам нужно случайное целое от 0 до 9, используйте инструкцию:

```
int( random.random() * 10 )
```

А если от 1 до 10, то:

```
int( random.random() * 10 ) + 1
```

Для генерирования нескольких значений можно использовать данную инструкцию в цикле, но при этом генерируемые числа могут повторяться. Чтобы гарантировать их уникальность, необходимо использовать метод **sample()** из модуля **random**. Этому методу в качестве аргументов указывается диапазон и количество возвращаемых чисел из этого диапазона. Для указания диапазона полезно пользоваться функцией **range()**, которая создает последовательность чисел в виде списка. Например, для того, чтобы сгенерировать шесть уникальных целых чисел в диапазоне от нуля до девяти:

```
random.sample( range( 10 ) , 6 )
```

А для диапазона от одного до десяти:

```
random.sample( range( 1, 11 ) , 6 )
```

При помощи такого метода легко получить шесть уникальных чисел от 1 до 49 для нашей с вами задачи.

**Внимание**

Преобразование числа с плавающей точкой в целое выполняется при помощи встроенной функции **int()**, которая просто отбрасывает значение после десятичной точки.

**На заметку**

Для функции **range()** можно указывать начальное и конечное значение диапазона. Если начальное опущено, предполагается, что оно равно нулю.

1. Запустите простейший текстовый редактор и создайте в нем новую программу, импортировав в нее пару функций из модуля `random`.


`sample.py`

```
from random import random , sample
```

2. Затем присвойте переменной значение случайного числа с плавающей точкой и выведите это значение.

```
num = random()

print( 'Random Float 0.0-1.0 : ' , num )
```

3. Теперь умножьте значение переменной на 10, приведите его к целочисленному типу, после чего еще раз выведите.

```
num = int( num * 10 )

print( 'Random Integer 0 - 9 : ' , num )
```

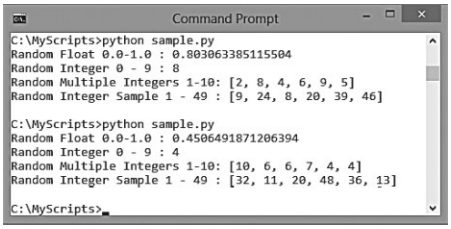
4. Добавьте цикл для множественного присваивания случайных целых элементам списка, а затем выведите все эти элементы.

```
nums = [] ; i = 0

while i < 6 :

nums.append( int( random() * 10 ) + 1 )

i += 1

print( 'Random Multiple Integers 1-10 :' , nums )
```

5. Наконец создайте список из шести уникальных целых чисел нужного диапазона и выведите его элементы заново.

```
nums = sample( range( 1, 49 ), 6 )

print( 'Random Integer Sample 1 - 49 : ' , nums )
```

6. Сохраните файл и запустите программу, чтобы посмотреть на результат генерирования чисел.






**Совет**

Функция `random.sample()` просто возвращает список, но никаких действий над его элементами не производит.

# Планирование программы

Перед созданием нового графического приложения полезно потратить некоторое количество времени на подготовку плана его построения. Необходимо четко определить задачи будущей программы, решить, какая функциональность будет необходима и какие элементы интерфейса потребуются для воплощения идеи.


Приблизительный план простого приложения, моделирующего выпадение номеров числовой лотереи «6 из 49», может выглядеть следующим образом.

## Назначение программы

- Программа должна генерировать набор из шести уникальных случайных чисел в диапазоне от 1 до 49 и иметь возможность перезагрузки (например, по кнопке **Reset**).

## Требуемая функциональность

- Функция для генерирования и последующего вывода шести уникальных случайных чисел.
- Функция для удаления всех сгенерированных значений с экрана.

## Необходимые элементы интерфейса

- Одно базовое окно постоянного размера, в котором будут содержаться все остальные виджеты и отображаться заголовок нашего приложения.
- Один виджет `Label`, на который мы поместим статическое изображение логотипа приложения, — просто чтобы улучшить внешний вид интерфейса.
- Шесть виджетов `Label` для динамического отображения набора сгенерированных чисел — по одному числу на каждый виджет.
- Один виджет `Button` для генерации и вывода чисел на виджеты `Label` при нажатии его кнопки. Данный виджет должен быть неактивным, если сгенерированные числа отображаются на экране.
- Еще один виджет `Button` для очистки данных с виджетов `Label` при нажатии его кнопки. Должен быть неактивным в случае, когда сгенерированные числа на экране не отображаются.

После того как мы с вами определились с планом программы, можно приступать к разработке базовых частей приложения путем создания всех необходимых виджетов.



**Совет**

Если переключить значение свойства `state` виджета `Button` с `NORMAL` на `DISABLED`, то можно некоторым образом влиять на действия пользователя — например, чтобы получить новую серию номеров, он вынужден будет нажать единственную активную в это время кнопку **Reset**.

1. Запустите простейший текстовый редактор и создайте в нем новую программу на Python, сделав сначала доступными методы и атрибуты модуля `tkinter`.

```
# Виджеты:
from tkinter import *
```

2. Затем создайте объекты окна и изображения.

```
window = Tk()
img = PhotoImage(file = 'logo.gif')
```

3. Теперь добавьте инструкции, создающие все необходимые виджеты.

```
imgLbl = Label( window, image = img )
label1 = Label( window, relief = 'groove', width = 2 )
label2 = Label( window, relief = 'groove', width = 2 )
label3 = Label( window, relief = 'groove', width = 2 )
label4 = Label( window, relief = 'groove', width = 2 )
label5 = Label( window, relief = 'groove', width = 2 )
label6 = Label( window, relief = 'groove', width = 2 )
getBtn = Button( window )
resBtn = Button( window )
```

4. Затем добавьте виджеты на окно, используя менеджер размещения `grid` — теперь виджеты готовы принимать аргументы, которые будут определять их позиционирование на экране на следующей стадии разработки.

```
# Размещение:
imgLbl.grid()
label1.grid()
label2.grid()
label3.grid()
label4.grid()
label5.grid()
label6.grid()
getBtn.grid()
resBtn.grid()
```

5. Наконец, добавьте цикл обработки событий окна.

```
# Обработка событий окна:
window.mainloop()
```

6. Сохраните файл и запустите программу — появится окно, содержащее все необходимые виджеты.


`lotto(widgets).py`

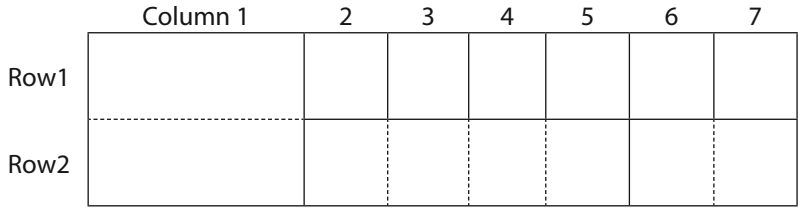

`lotto.gif`

**На заметку**

Свойство `relief` определяет стиль рамки, а свойство `width` — ширину метки в символах.

# Построение интерфейса

После того, как все виджеты созданы, можно приступать к разработке шаблона интерфейса. Для этого надо задать аргументы, определяющие параметры размещения виджетов на экране. Мы будем использовать горизонтальное расположение, при котором слева разместим виджет **Label** с логотипом, а справа от него — шесть виджетов для номеров лотереи, под которыми будут находиться две управляющие кнопки. Для этого воспользуемся менеджером размещений **grid**, который позволяет располагать виджеты в ячейках таблицы. В результате виджет с логотипом будет находиться у нас в первом столбце и занимать две строки, виджеты для номеров займут в верхней строке ячейки со второй по седьмую, первая кнопка расположится во второй строке, заняв 4 ячейки, начиная со второй, а две последние ячейки этой строки — место для второй кнопки, примерно вот так:





lotto(layout).py

1. Отредактируйте программу, начатую на предыдущей странице, разместив сначала виджет **Label** для логотипа в первой строке и первом столбце, указав для него параметр заполнения на две строки.

```
# Размещение:

imgLbl.grid( row = 1, column = 1, rowspan = 2 )
```

2. Затем расположите во втором столбце и первой строке виджет **Label** для первого номера лотереи, добавив слева и справа от него пространство в 10 пикселов.

```
label1.grid( row = 1, column = 2, padx = 10 )
```

3. В третьем столбце и первой строке разместите виджет **Label** для вывода второго номера лотереи, добавив ему те же 10 пикселов пространства.

```
label2.grid( row = 1, column = 3, padx = 10 )
```

4. Далее по аналогии выполните шаги с 4 по 7: для третьего номера

```
label3.grid( row = 1, column = 4, padx = 10 )
```

5. Для четвертого номера

```
label4.grid( row = 1, column = 5, padx = 10 )
```

6. Для пятого номера

```
label5.grid( row = 1, column = 6, padx = 10 )
```

7. Для виджета **Label**, на котором будет размещаться шестой номер, дополнительно укажите пространство справа в 20 пикселов.

```
label6.grid( row = 1, column = 7, padx = ( 10, 20) )
```

8. Затем разместите виджет первой кнопки во втором столбце второй строки, позволив ему занять четыре ячейки.

```
getBtn.grid( row = 2, column = 2, columnspan = 4 )
```

9. Теперь виджету для второй кнопки определите место в шестом столбце той же строки, указав, что он займет в ней две ячейки.

```
getBtn.grid( row = 2, column = 6, columnspan = 2 )
```

10. Сохраните файл и запустите программу — должно появиться окно, содержащее все необходимые виджеты, которые упорядочены по ячейкам таблицы.


Размеры окна автоматически подстроились под его содержимое, а положение двух кнопок центрировалось относительно занятых ими ячеек.

**Совет**

Свойства `rowspan` и `columnspan` менеджера `grid` работают по тому же принципу, что и атрибуты HTML `rowspan` и `colspan`, определяющие поведение ячеек таблицы.

**Совет**

Дополнительное увеличение отступа для последнего виджета `Label` сделано для задавания небольшой области поля справа.

# Определение постоянных величин

Разместив все необходимые виджеты в «сеточном» шаблоне на предыдущей странице, вы можете определять значения, которые будут постоянными для виджетов, то есть не будут меняться во время исполнения программы.


lotto(static).py

1. Измените программу с предыдущей страницы добавлением в нее нового раздела перед последней инструкцией с циклом обработки событий. Данный раздел будет начинаться с определения заголовка окна

```
# Постоянные величины:

window.title( 'Lotto Number Picker' )
```

2. Затем добавьте инструкцию, которая будет запрещать пользователю изменять размеры окна, как по горизонтали, так и по вертикали. В результате стандартная кнопка изменения размеров окна будет деактивирована.

```
window.resizable( 0, 0 )
```

3. Теперь добавьте инструкцию, в которой определите текст, размещаемый на первой кнопке.

```
getBtn.configure( text = 'Get My Lucky Numbers' )
```

4. Затем добавьте инструкцию для определения текста, размещаемого на второй кнопке.

```
resBtn.configure( text = 'Reset' )
```

5. Сохраните файл и запустите программу — теперь вы видите, что у окна появился заголовок, кнопка изменения размеров деактивирована, а две нижние кнопки изменили свой размер в соответствии с размещенным на них текстом.


**Совет**

После того как виджеты созданы, значения их свойств могут быть добавлены или изменены при помощи метода `configure()`.

# Инициализация изменяемых значений

Определив значения, которые остаются постоянными, необходимо задать те переменные, которые будут изменены во время исполнения программы.

1. Измените программу с начальной страницы раздела, добавив перед последней инструкцией цикла обработки событий новый блок, в котором определите первоначальные значения для всех шести меток, где будут выводиться номера лотереи.


lotto(initial).py

```
# Начальные значения:

label1.configure( text = '...' )

label2.configure( text = '...' )

label3.configure( text = '...' )

label4.configure( text = '...' )

label5.configure( text = '...' )

label6.configure( text = '...' )
```

2. Затем добавьте инструкцию, определяющую деактивированное состояние правой нижней кнопки в первоначальный момент.

```
resBtn.configure( state = DISABLED )
```

3. Сохраните файл и запустите программу — на всех верхних метках теперь отображены знаки многоточия, а кнопка **Reset** деактивирована.


**На заметку**

Состояния кнопок определяются модулем `tkinter` как `DISABLED` (выключена), `NORMAL` (включена) и `ACTIVE` (нажата).

# Добавление рабочей функциональности

Определив начальные значения динамических переменных на предыдущей странице, вы можете задавать функциональность, которая определяет реакцию на нажатие пользователем кнопок во время исполнения программы.


lotto.py

1. Измените программу с предыдущей страницы добавлением еще одного раздела перед последней инструкцией цикла обработки событий. Здесь вы сделаете доступной функцию `sample()` из модуля `random`.

```
# Изменяемые величины:

from random import sample
```

2. Затем определите функцию, которая будет генерировать шесть уникальных случайных чисел и присваивать их шести меткам, а также менять состояние двух кнопок на противоположное.

```
def pick() :

nums = sample( range( 1, 49 ), 6 )

label1.configure( text = nums[0] )

label2.configure( text = nums[1] )

label3.configure( text = nums[2] )

label4.configure( text = nums[3] )

label5.configure( text = nums[4] )

label6.configure( text = nums[5] )

getBtn.configure( state = DISABLED )

resBtn.configure( state = NORMAL )
```

**Совет**

Данные шаги организуют функциональность, похожую на ту, которая была представлена консольным приложением в начале этой главы.

3. Теперь определите функцию, которая будет выводить во всех верхних метках знак многоточия, а также возвращать значение двух кнопок в их начальное состояние.

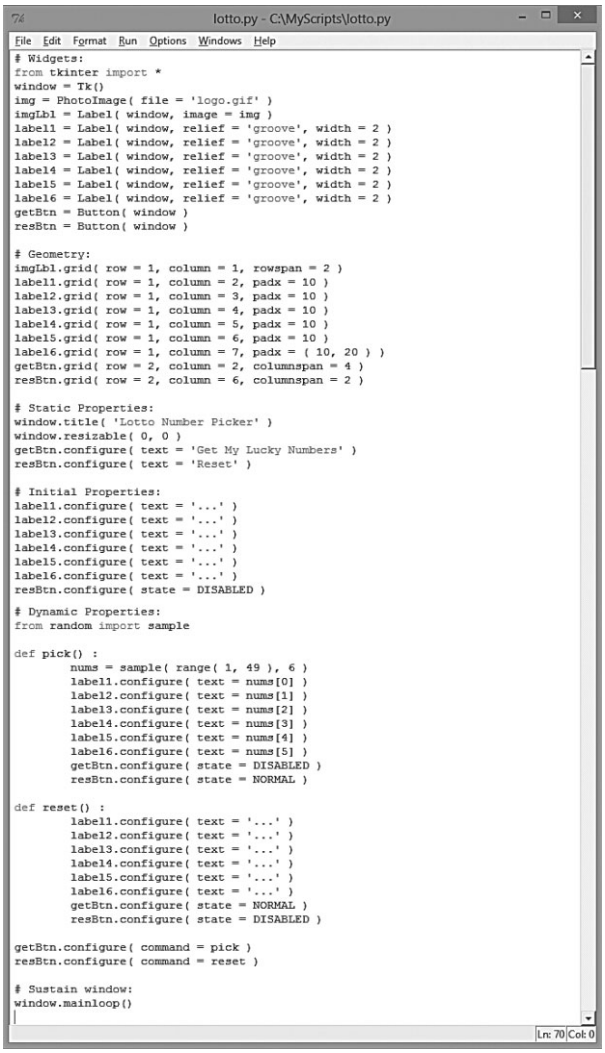
```
def reset() :

label1.configure( text = '...' )

label2.configure( text = '...' )

label3.configure( text = '...' )

label4.configure( text = '...' )

label5.configure( text = '...' )
```

```
label6.configure( text = '...' )

getBtn.configure( state = NORMAL )

resBtn.configure( state = DISABLED )
```

4. Теперь определите инструкции, в которых будете задавать целевую функцию, вызываемую при нажатии пользователем каждой кнопки.

```
getBtn.configure( command = pick )

resBtn.configure( command = reset )
```

5. В завершение сохраните файл — готовая программа будет выглядеть примерно таким образом.




**На заметку**

По договоренности, все инструкции `import` ставятся в самом начале программы, но это не обязательно.

**Внимание**

Цвета в редакторе IDLE могут отличаться от представленных здесь, но текст программ в точности соответствует шагам этой главы.

# Тестирование программы

Определив на предыдущих страницах план программы, необходимые элементы управления и добавив требуемую функциональность, мы с вами готовы протестировать программу.

1. Запустите приложение и оцените его внешний вид.


Статический текст появился на заголовке окна и на двух кнопках, стандартная кнопка изменения размеров окна деактивирована, а верхние метки содержат текстовые значения, определяющие первоначальный текст (многоточия), кнопка **Reset** находится в первоначальном отключенном состоянии.

2. Затем нажмите кнопку **Get My Lucky Numbers** — при этом исполнятся все инструкции функции `pick()`.


На метках должны появиться номера из желаемого диапазона, а нижние кнопки — изменить свое состояние. Следующий набор номеров не сгенерируется до тех пор, пока приложение не будет перезагружено с помощью кнопки **Reset**.

3. Запишите сгенерированные на этом шаге числа для последующего сравнения.

**На заметку**

В любой сгенерированной серии нет повторяющихся номеров, так как используемая функция `sample()` модуля `random` возвращает набор уникальных целых значений.

4. Нажмите кнопку **Reset**, при этом исполнятся все инструкции функции `reset()` и вы увидите, что приложение возвратится в свое начальное состояние, как и требовалось.


5. Снова нажмите кнопку **Get My Lucky Numbers**, чтобы заново выполнить функцию `pick()`, и удостоверьтесь, что новая серия номеров отличается от предыдущей.


6. Наконец перезагрузите приложение и нажмите кнопку **Get My Lucky Numbers** еще раз и убедитесь, что первая серия сгенерированных номеров отличается от тех, что вы записали, то есть от первой серии предыдущего запуска программы.


**На заметку**

Серия сгенерированных номеров не повторяется при каждом новом запуске, так как работа генератора случайных чисел основывается на текущем системном времени, которое каждый раз при вызове генератора, естественно, различается.

# Компиляция программы

Успешно протестировав приложение на предыдущей странице, вы, возможно, захотите использовать его на других компьютерах, на которых может быть не установлен интерпретатор Python. Для того чтобы приложение могло успешно запускаться без него, ваши программные файлы должны быть скомпилированы в исполняемый файл (с расширением *.exe*).

**Совет**

Графические приложения требуют другой базы на Windows, чем консольные (различного набора файлов и библиотек).

Существует полезный набор скриптов и модулей **cx_Freeze** для компиляции программы Python в набор исполняемых файлов для операционных систем Windows, Mac или Linux. Данный инструмент является кроссплатформенным и работает на всех платформах, на которых работает сам Python. Он находится в свободном доступе, и получить его вы можете, загрузив со страницы **cx-freeze.sourceforge.net**. Для загрузки доступны инсталляторы MSI для Windows, RPM для Linux, как для 32-, так и для 64-битных версий систем. Просто выберите подходящую для вашей системы версию, загрузите ее и запустите инсталлятор.

Инструмент **cx_Freeze** использует модуль Python **distutils** и требует так называемой программы настройки (setup script), описывающей, как и что устанавливать, и являющейся своеобразным файлом поддержки для распространения вашего приложения. Программа настройки обычно называется *setup.py* и состоит в основном из вызова функции **setup()**, которая предоставляет информацию в виде пар аргументов. В ней указываются любые необходимые опции для компиляции, какие нужно включить файлы изображений или модули, а также идентифицируется исполняемый файл программы и тип платформы, на которой она будет работать. Например, программа настройки для приложения, разработанного в этой главе, должна включать файл изображения логотипа *logo.gif* и определять в качестве исполняемой программу *lotto.py*.

Программа настройки, которая может быть запущена с помощью команды **built** , создает подкаталог с именем *built*, содержащий в свою очередь подкаталоги с именами, начинающимися с *exe.* и заканчивающимися идентификатором вашей платформы, например *win32-3.3*.


**setup.py**

1. Запустите простейший текстовый редактор и начните программу на Python, сделав доступными функции модуля **sys** и модуля **cx_Freeze**.

```
import sys from cx_Freeze import setup, Executable
```

2. Затем добавьте инструкции для идентификации используемой базовой платформы.

```
base = None

if sys.platform == 'win32' : base = 'Win32GUI'
```

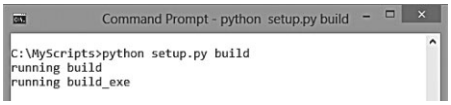
3. Теперь добавьте инструкцию со списком подключаемых файлов (опций).

```
opts = { 'include_files' : [ 'logo.gif' ] , 'includes' : [ 're' ]}
```

4. Наконец добавьте инструкции с вызовом функции `setup()`, которой передается в качестве аргументов вся необходимая информация.

```
setup( name = 'Lotto' ,

version = '1.0' ,

description = 'Lottery Number Picker' ,

author = 'Mike McGrath' ,

options = { 'build_exe' : opts } ,

executables = [ Executable( 'lotto.py', base = base ) ] )
```

5. Сохраните файл *setup.py* рядом с файлами вашего приложения, затем запустите программу настройки, для того чтобы сгенерировать распространяемый пакет.




6. Дождитесь, пока процесс построения создаст набор файлов в подкаталоге *built*, а затем скопируйте их все на переносное устройство, например, накопитель USB.

7. Теперь скопируйте распространяемый набор на другой компьютер, где Python не установлен, и запустите исполняемый файл. Вы увидите, как запустится ваше приложение.

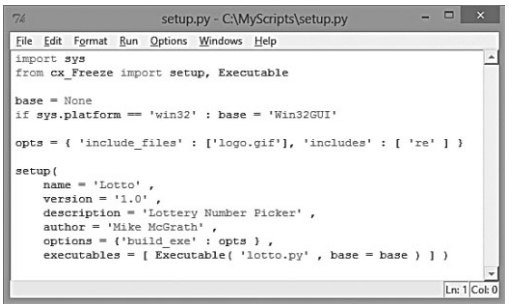

**Внимание**

Модуль `re` (регулярных выражений) добавлен здесь в качестве опции компиляции вручную, поскольку на момент написания данной книги инструмент `cx_Freeze` не добавлял его автоматически.

# Распространение приложения

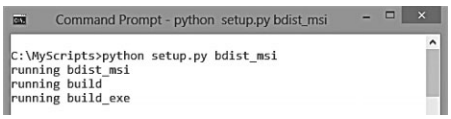
Приложение, разработанное на языке Python, можно распространять на компьютеры с операционной системой Windows, используя инструмент **cx_Freeze**, представленный в предыдущем разделе и создающий простой инсталлятор MSI.

Он использует в точности ту же программу настройки (setup script), что была представлена в предыдущем примере; вы ее можете видеть ниже.




Программа настройки *setup.py* может быть запущена с помощью команды **bdist_msi** аналогично команде **build**, при этом создается подкаталог с именем *dist*, содержащий инсталлятор MSI для вашего приложения. Имя инсталлятора будет включать имя приложения, версию и реквизиты вашей системы.

1. Сохраните программу настройки вместе с вашими файлами приложения и запустите для создания инсталлятора Windows.




2. Подождите, пока не создастся инсталлятор в подкаталоге *dist*, затем скопируйте его на переносное устройство, например накопитель USB.

**Совет**

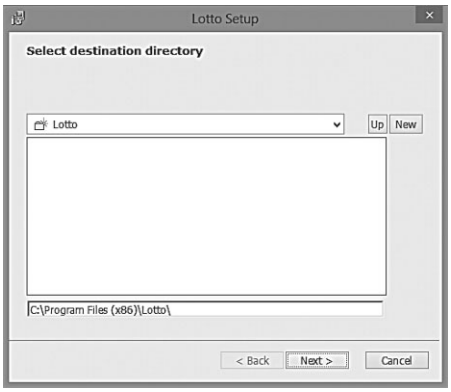
На компьютерах под управлением операционной системы OS X вы можете использовать команду **bdist_dmg** для построения образа диска.

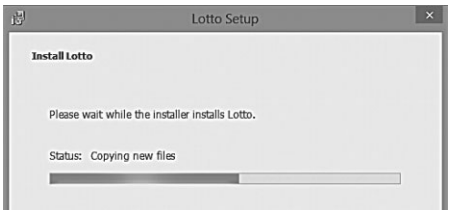
**Совет**

Подробную информацию об инструменте cx_Freeze вы можете найти на странице **cx-freeze.readthedocs.org/en/latest/index.html**.

3. Теперь скопируйте инсталлятор на другой компьютер под управлением операционной системы Windows, на которой не установлен интерпретатор Python, и запустите его на выполнение.

4. Выберите место для установки, либо подтвердите предлагаемое по умолчанию.




5. После того как инсталлятор закончит копирование файлов, выберите из списка установленное приложение и запустите исполняемый файл.

# Заключение

- Функции для генерирования псевдослучайных чисел обеспечиваются модулем `random`, содержащимся в стандартной библиотеке языка Python.
- Для того чтобы сгенерировать псевдослучайное число с плавающей точкой в диапазоне от 0.0 до 0.1, нужно воспользоваться функцией `random()` модуля `random`.
- Для генерирования уникального набора целых чисел из заданного диапазона необходимо использовать функцию `sample()` из того же модуля `random` .
- Назначение программы, требуемая функциональность и необходимые элементы интерфейса — все это определяется на этапе планирования будущего приложения.
- Менеджер размещения `grid` — один из инструментов для построения интерфейса программы — организует расположение виджетов по строкам и столбцам таблицы.
- Статические свойства остаются постоянными в течение выполнения программы.
- Свойства, которые изменяют свое значение во время исполнения программы в результате ответов на действия пользователя, называются динамическими.
- После того как текст программы готов, протестируйте ее, чтобы убедиться, что все работает так, как было задумано.
- Файлы программы разрешается «заморозить» (скомпилировать) в готовый для исполнения пакет, который может быть распространен на другие компьютеры, где нет интерпретатора Python.
- Чтобы подготовить исполняемые файлы для Windows, OS X или Linux, можно воспользоваться инструментом `cx_Freeze`, который задействует модуль `distutils` интерпретатора Python.
- Для описания готового распространяемого набора используйте программу настройки (setup script).
- Чтобы получить распространяемый набор, готовый к выполнению на других компьютерах, нужно программу настройки запустить с дополнительной командой `build`.
- При помощи инструмента `cx_Freeze` можно создать простую программу установки для распространения на компьютерах под управлением операционной системы Windows.
- Для создания MSI-установщика, включающего все необходимые файлы для исполнения, программу настройки следует запустить с дополнительной командой `bdist_msi`.

# Предметный указатель

# Начни программировать прямо сейчас!

## САМОЕ ВАЖНОЕ:

- ФУНКЦИИ
- ПЕРЕМЕННЫЕ
- КЛЮЧЕВЫЕ СЛОВА
- МЕТОДЫ
- ОБЪЕКТЫ
- ОПЕРАТОРЫ
- АТРИБУТЫ
- ЗАПРОСЫ
- И МНОГОЕ ДРУГОЕ

Книга **«ПРОГРАММИРОВАНИЕ НА PYTHON ДЛЯ НАЧИНАЮЩИХ»** является исчерпывающим руководством для того, чтобы научиться программировать на языке Python.

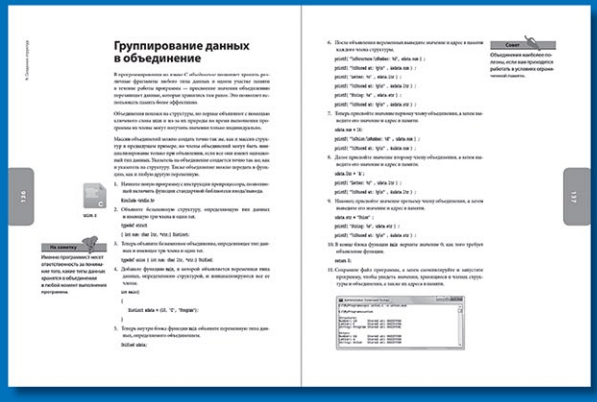
В этой книге с помощью примеров программ и иллюстраций, показывающих результаты работы кода, разбираются все ключевые аспекты языка. Установив свободно распространяемый интерпретатор Python, вы с первого же дня сможете создавать свои собственные исполняемые программы!

Познакомившись с основами языка, вы перейдете к объектно-ориентированному программированию и созданию CGI-сценариев для обработки данных веб-форм, научитесь создавать графические приложения с оконным интерфейсом и распространять их на другие устройства. В обучении вам помогут готовые примеры которые можно скачать по адресу: **http://eksmo.ru/Python_examples.zip**

Книга «Программирование на Python для начинающих» идеально подойдет программистам, переключающимся на работу с другим языком, студентам и школьникам, изучающим язык Python.

## ЧТО ВНУТРИ?

Эти значки сделают обучение еще проще. Каждый раз, когда при чтении книги вы встречаете один из этих значков, знайте — мы приготовили для вас какой-то полезный совет, придающий остроты процессу обучения, выделили нечто необходимое для запоминания или выносим предостережения держаться подальше от возможных проблем.




**«Отличный старт для решивших начать изучать программирование с Python. Даже если вы никогда до этого не создавали программы, с помощью этой книги вы сможете пройти все шаги от установки интерпретатора до запуска и отладки своих первых приложений».**

**Александр Корчагин,**
*руководитель отдела финансовых информационных систем,*
*Центральная Дистрибьюторская Компания*

ISBN 978-5-699-81406-0