Продвинутый Python в примерах

Автор: Замятин Д. С.

e-mail: dsz@ukr.net

Версия от 13.03.17

Последняя версия: http://goo.gl/KVqeiQ

Киев — 2017

Введение в программирование на Python

Язык программирования Python в своей основной реализации (cPython) является интерпретируемым. Это значит, что инструкции языка не переводятся в машинный код для выполнения процессором компьютера, а исполняются непосредственно программой-интерпретатором. Такой подход имеет как преимущества, в частности программы легко переносятся с одной платформы на другую, так и недостатки, проявляющиеся прежде всего в невысокой, по сравнению с компилируемыми языками, производительности. Благодаря тому, что Python – интерпретируемый, инструкции языка могут выполняться в двух режимах: интерактивном и пакетном. При использовании интерактивного режима пользователь взаимодействует с консолью интерпретатора, которая позволяет ему ввести инструкцию, сразу же выполнить ее и получить результат, что удобно в процессе отладки и изучения языка. В интерактивном режиме результат вычисления введенного выражения сразу выводится на экран, а в пакетном режиме для этого приходится использовать оператор print:

```
>>> print "Hello, world!"
Hello, world!
```

В Python 3 вместо оператора используется встроенная функция print():

```
>>> print("Hello, world!")
Hello, world!
```

Также, для того, чтобы позволить пользователю вводить значения с клавиатуры во время выполнения программы, может быть использована функция input():

```
>>> s = input("? ")
? Hello
>>> s
'Hello'
```

В Python 2 функция input() имела название raw_input(). Следует обратить внимание на то, что в Python 2 также существует функция input(), но она выполняет попытку интерпретации пользовательского ввода в соответствии с синтаксисом языка, т. е. ее вызов эквивалентен вызову eval(raw_input()). Подобное поведение позволяет пользователю ввести и выполнить произвольный программный код, что может

привести к потере данных или другим нежелательным последствиям, поэтому данная функция небезопасна.

Кроме того, в интерактивном режиме определена специальная переменная с именем _, которая хранит значение последнего вычисленного выражения.

Пакетный режим предполагает выполнение интерпретатором последовательности инструкций языка, которые сохранены в файле в порядке, соответствующем их выполнению. Такие файлы называются программами, и они позволяют многократно исполнять одни и те же последовательности инструкций с разными входными данными. Как правило, файлы программ на языке Python принято называть модулями и сохранять с расширением .py. Для обеспечения возможности повторного использования кода Python позволяет импортировать одни модули из других. Благодаря этому один раз написанный программный код может быть использован многократно. Модули, которые уже хотя бы раз были проимпортированы, интерпретатор сохраняет в специальном формате для ускорения запуска. Обычно такие файлы имеют расширение .pyc. В Python 2 .pyc-файлы располагаются в том же каталоге, что и сам модуль, а в Python 3 для этих целей в текущем каталоге создается дополнительный каталог __pycache__.

Комментариями в программах на языке Python считается набор символов, который начинается с символа '#' и до конца строки. Такой набор символов игнорируется интерпретатором и служит для коммуникации разработчиков.

Файлы программ являются обычными текстовыми файлами в кодировке ASCII для Python 2 или UTF-8 для Python 3. Если возникает необходимость изменить кодировку файла, например, на UTF-8 для использования кириллицы в Python 2, то первой строкой файла программы должен быть комментарий вида:

```
# coding: utf-8
```

Допустимы и другие форматы представления данного комментария. Кроме этого, в POSIX-совместимых системах для того, чтобы программа могла запускаться непосредственно из командной строки, в самом начале файла программы необходимо указать путь к интерпретатору:

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

или

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

При написании исходных кодов (файлов-программ) на Python, как правило, пользуются соглашениями о стилях оформления, которые определяют внешний вид программы, порядок расстановки отступов, способ именования объектов языка и т. п. Благодаря этим соглашениям внешний вид всех программ является одинаковым, что упрощает их чтение и повышает производительность труда разработчиков. Для Python стили оформления определены в специальном документе PEP8 (https://www.python.org/dev/peps/pep-0008/). В частности, PEP8 указывает, что длина строки в программе не может быть более 79 символов, чтобы код было удобно читать на мониторах любых размеров. Поскольку в Python символ перевода строки указывает на конец оператора, для переноса кода на следующую строку используется символ \, например:

```
if 1 <= day <=31 \
     and 1 <= month <=12
```

Если оператор содержит круглые, квадратные или фигурные скобки, то после открывающей скобки переводы строки допускаются:

```
weakdays = (
     'Sunday',
     'Monday',
     …
)
```

Объекты

Программа на языке Python содержит последовательность инструкций, которые определяют действия, необходимые для получения результата. Эти инструкции также называют операторами. Каждый оператор предполагает наличие операндов – данных, над которыми действие производится. В языке Python любые данные, которые используются при обработке представляются объектами. Объект определяется рядом свойств, при чем обязательными свойствами являются: identity, тип и значение. Identity или уникальный идентификатор позволяет отличать объекты друг от друга. Его можно получить при помощи функции id():

```
>>> id(1)
152373424
>>> id('hello')
3070667648
```

В реализации cPython на данный момент для определения идентификатора используется физический адрес объекта в памяти, поэтому значения id() могут отличаться от приведенных в примере. Для определения, являются ли два объекта одним и тем же, используется функция is:

```
>>> True is False
False
>>> s = 'The same'
>>> s1 = s
>>> s is s1
True
```

Вторым обязательным свойством любого объекта в Python является тип. Он в первую очередь позволяет определить набор операций, который можно выполнять с объектом, его способность изменять свое значение (mutability) и объем памяти, который занимает объект. Узнать тип объекта можно с помощью функции type():

```
>>> type(1)
<type 'int'>
>>> type('hello')
<type 'str'>
>>> type(True)
<type 'bool'>
>>> type(None)
<type 'NoneType'>
```

Считается, что Python – сильнотипизированный язык с динамической типизацией. Сильный контроль типов в языке предполагает возможность неявного приведения типов операндов в выражении, если это не приводит к потере данных. В противном случае, если неявное приведение невозможно, выводится сообщение об ошибке. Такой контроль типов упрощает выявление многих ошибок в программе на ранних стадиях тестирования. Динамическая типизация подразумевает, что язык не требует от разработчика явно определять типы объектов, интерпретатор способен это делать самостоятельно из контекста.

Кроме неявного приведения типов существует возможность явного приведения. Имя каждого типа фактически является конструктором значений данного типа:

```
>>> int(3.2)
3
```

```
>>> float(2)
2.0
>>> int('123')
123
>>> str(4.5)
'4.5'
>>> str(True)
'True'
>>> int('abc')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10:
'abc'
```

Последний пример показывает, что если значение невозможно привести к заданному типу, возникает исключение ValueError.

Наконец, третьим свойством объекта является его значение, над которым собственно и проводятся операции.

Следует отметить, что переменные в Python представляют собой лишь ссылки на объекты. Поэтому сама по себе переменная не привязана к определенному типу данных, как в других языках программирования. Переменная с одним и тем же именем в течении одного сеанса работы может ссылаться на значения разных типов:

```
>>> a = 1
>>> type(a)
<class 'int'>
>>> a = 'hello'
>>> type(a)
<class 'str'>
```

В данном случае, отображаемый тип относится не к переменной, а к значению, на которое переменная ссылается.

Как Python работает с переменными можно проследить путем анализа идентификаторов объектов:

```
>>> a = 1
>>> id(a)
152373424
>>> b = 1
>>> id(b)
152373424
```

Видно, что константа 1 в памяти определена один единственный

раз, а переменные a и b всего лишь ссылаются на нее. Это сделано для уменьшения объема памяти, занимаемого программой при выполнении. При присваивании значения переменной изменяемые и неизменяемые объекты ведут себя по разному. Для неизменяемых объектов ссылка просто переносится на другую константу:

```
>>> b = 2
>>> id(b)
152373412
```

Изменяемые объекты являются контейнерами для констант, поэтому при изменении их содержимого идентификаторы не изменяются:

```
>>> l = [1, 2, 3]
>>> id(l)
3074694316L
>>> l[0] = 0
>>> l
[0, 2, 3]
>>> id(l)
3074694316L
```

Каждое значение хранится в памяти до тех пор, пока существуют переменные, ссылающееся на него. Чтобы определить, нужно ли еще хранить значение в памяти или его можно удалить, используется счетчик ссылок, связанный с этим значением. Каждый раз, когда появляется переменная, ссылающаяся на данное значение, этот счетчик увеличивается на 1 и, соответственно, каждый раз, когда переменная исчезает, счетчик уменьшается. Когда значение счетчика становится равным 0, значение удаляется из памяти. Узнать количество ссылок на значение можно с помощью функции getrefcount() модуля sys.

```
>>> from sys import getrefcount
>>> a = 1
>>> getrefcount(1)
797
>>> b = 1
>>> getrefcount(1)
798
>>> b = 2
>>> getrefcount(1)
797
>>> del a
```

```
>>> getrefcount(1)
796
```

Первый вызов getrefcount() показывает, что константа 1 уже используется 797 раз для внутренних целей интерпретатора. Создание новой переменной с этим значением увеличивает счетчик на 1, а присвоение нового значения или удаление переменной уменьшает его. К сожалению, данный способ работает не всегда. В случаях, когда значение переменной ссылается на само себя, т. е. имеют место так называемые циклические ссылки, даже при удалении переменной счетчик ссылок не будет равным 0. Создать циклическую ссылку достаточно просто при использовании любого изменяемого контейнера, например, словаря:

```
d = {}
>>> d[0] = d
>>> d
{0: {...}}
```

В данном случае элемент словаря с ключем 0 содержит ссылку на сам словарь. Если бы интерпретатор попытался вывести на экран содержимое данного элемента, это привело бы к бесконечному циклу, поэтому он распознает циклические ссылки и отображает их при помощи многоточия. Для решения проблемы циклических ссылок в Python предусмотрен специальный модуль, который называется сборщиком мусора. Его работой можно управлять с помощью модуля gc стандартной библиотеки. Сборщик мусора запускается автоматически на основании порогов, заданных в модуле gc.

```
 >>> import gc
>>> gc.get_threshold()
(700, 10, 10)
```

Первый порог определен как разность между количеством созданных и удаленных переменных. Когда эта разность превышает установленное пороговое значение, запускается сборка мусора. Все объекты, созданные до процесса сборки относятся к поколению 0; объекты, пережившие сборку, переносятся в поколение 1; а после следующей сборки — в поколение 2. Обычно при сборке мусора анализируется поколение 0, но если количество сборок превысило второй порог, выполняется анализ поколения 1. Соответственно, когда количество сборок с учетом поколения 1 превышает третий порог, анализируется поколение 2. Существует возможность

принудительного запуска сборки мусора с помощью функции collect() модуля gc. Чтобы продемонстрировать работу этой функции, создадим несколько объектов с циклическими ссылками.

```
>>> import gc
>>> for i in range(10):
...     d = {}
...     d[0] = d
>>> gc.collect()
9
```

В данном примере на каждую итерацию цикла создается по одному экземпляру словаря. Далее элементу с ключом 0 присваивается экземпляр самого словаря и таким образом создается циклическая ссылка. На следующей итерации переменной d присваивается новый словарь, что приводит к тому, что счетчик ссылок старого словаря уменьшается на 1 и должен стать равным 0. Но в данном случае сам объект содержит ссылку на самого себя, поэтому счетчик не становится нулевым и удаление не происходит. Принудительный вызов collect() обнаруживает подобные объекты и возвращает число удаленных значений — 9. На самом деле было создано 10 объектов, но для последнего из них ссылка продолжает хранится в переменной, поэтому удалено 9 объектов.

Чтобы помочь встроенному механизму освобождения памяти, основанному на подсчете ссылок, циклические ссылки можно пометить как слабые. Слабые ссылки не учитываются при подсчете и соответственно не влияют на время жизни объекта. Для этого используется модуль стандартной библиотеки weakref.

Python имеет ряд встроенных (builtin) типов данных, которые имеет смысл рассмотреть. Наиболее простым типом является тип NoneType. Этот тип определяет всего лишь одну константу, которая имеет имя None. Данный тип используется в случае, если значение отсутствует. В частности, функции не имеющие оператора return, и, как следствие, ничего не возвращающие, на самом деле возвращают None.

```
>>> def f():
...     pass
...
>>> a = f()
>>> a
>>> type(a)
<type 'NoneType'>
```

Обратите внимание, что при попытке вывода None на экран не выводится ничего. Иногда None также используется для инициализации переменных, когда у них еще нет значения.

Аналогично тип NotImplementedType определяет одну константу NotImplemented, которая используется для того, чтобы показать, что в данной реализации интерфейса некоторая функция не определена.

Также, для удобства написания кода для работы со списками существует специальный тип ellipsis, единственное значение которого отображается на экране многоточием.

```
>>> type(...)
<class 'ellipsis'>
```

Данное значение может быть применяется в библиотечном коде для индексов списков, а также может быть использовано как константа, например, для определения функции без тела:

```
>>> def f(): ...
>>> f()
>>>
```

Числовые типы

Тип целых чисел включает в себя 2 основных типа: bool, int (в Python 2 также присутствует тип long). Тип bool используется для представления логических значений и определяет две константы: True – истина и False – ложь. Над значениями данного типа определены логические операции конъюнкции and, дизъюнкции or и инверсии not. Таблицы истинности данных операций приведены ниже.

x	y	x and y	x or y	not x
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

Операции and и or вычисляются по сокращенному алгоритму: если хотя бы один аргумент конъюнкции равен False, то вся конъюнкция считается ложной и остальные аргументы не вычисляются. Аналогично, если хотя бы один аргумент дизъюнкции True, то вся дизъюнкция считается истинной.

Логический тип используется, как правило, для представления

различных логических флагов, хранения и анализа результатов сравнения и т. д.

Типы int и long служат для представления целых чисел. В Python 2 максимальное значение для типа int определялось при помощи модуля sys:

```
>>> import sys
>>> sys.maxint
2147483647
```

При превышении максимального значения в Python2 число автоматически преобразуется в тип long. Поскольку преобразование выполняется прозрачно для пользователя, в Python 3 от типа long отказались, а тип int ведет себя как long. Значения целого типа не имеют ограничений по длине, поэтому в Python отсутствуют ошибки, связанные с целочисленными переполнениями.

```
>>>999999999999999999999999999999999999999999999999
999999999999999999999999999999999999999999999999999
999999999999999999999999999999999999999999 + 1
100000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000
```

Целочисленные константы представляются как последовательность цифр, которой может предшествовать знаки «минус» или «плюс» (обычно не используется):

```
0    1    -25  +4   32   99999999
```

Можно представлять целочисленные константы в системах счисления, отличных от десятичной:

```
>>> 0123 # восьмеричная
83
>>> 0x1A # шестнадцатеричная
26
>>> 0Xabc # шестнадцатеричная
2748
>>> 0b10101 # двоичная
21
```

Т. к. 0 в префиксе числа может восприниматься пользователем

как незначащий, а не как признак восьмеричной системы счисления, в Python 3 целое число вообще не может начинаться с 0, а для указания на восьмеричную систему используется комбинация символов 0o.

```
>>> 012
  File "<stdin>", line 1
    012
      ^
SyntaxError: invalid token
>>> 0o12
10
```

Вообще, преобразование из произвольной позиционной системы счисления можно выполнить с помощью функции int(), где вторым параметром является основание системы счисления:

```
>>> int('123', 4)
27
>>> int('abc', 15)
2427
```

Для обратных преобразований существуют соответствующие встроенные функции (результат выполнения будет иметь строковый тип):

```
>>> bin(123)
'0b1111011'
>>> oct(12)
'014'
>>> hex(255)
'0xff'
```

Константы типа long в Python 2 выглядят так:

```
0L   999999999991  -987654321L
```

Для перечисленных целочисленных типов определен стандартный набор арифметических операций, битовые операции и операции сравнения:

```
>>> 2 + 3 # сложение
5
>>> 2 - 3 # вычитание
-1
>>> 2 * 3 # умножение
```

```
6
>>> 5 / 2 # деление
2.5
>>> 3 // 2 # целочисленное деление
1
>>> 3 % 2 # остаток от деления
1
>>> 2 ** 3 # возведение в степень
8
```

Следует обратить внимание на то, что операция деления в Python 2 для целых операндов возвращает целый результат. В Python 3 деление реализовано более логично: результат для любых операндов будет числом с плавающей точкой, а для целочисленного деления определен специальный оператор //.

```
>>> 123 | 456 # побитовое «ИЛИ»
507
>>> 123 & 456 # побитовое «И»
72
>>> 123 ^ 456 # побитовое исключающая «ИЛИ»
435
>>> ~1223 # побитовая инверсия
-1224
>>> 128 >> 3 # битовый сдвиг на 3 разряда вправо
16
>>> 3 << 4 # битовый сдвиг на 4 разряда влево
48
>>> 2 > 3 # больше
False
>>> 2 < 3 # меньше
True
>>> 2 >= 3 # больше или равно
False
>>> 2 <= 3 # меньше или равно
True
>>> 2 == 3 # равно
False
>>> 2 != 3 # не равно (в Python 2 можно <>)
True
>>> 1 < 2 < 3 # проверка на принадлежность диапазону
True
```

Выражение в последней строке примера не имеет ограничений на количество операндов, т. е. Python позволяет строить цепочки

сравнений.

```
>>> 1 < 2 == 2 < 4
True
>>> 1 < 2 == 3 < 4
False
>>> 5 > 3 > 2 > 1
True
```

Приоритет операций в выражениях определяется при помощи круглых скобок:

```
>>> (2 + 2) * 2
8
```

Для представления рациональных чисел в Python используется тип float, который соответствует представлению чисел с плавающей точкой. Строго говоря, числа типа float являются приближенным представлением и их точность сравнительно невысока. Это, в частности, ограничивает их использование для хранения денежных и т. п. значений. Константы данного типа выглядят следующим образом:

```
0.5  -2.3     7.   .33  2.34e2   -125E-3
```

В случае отсутствия целой или дробной части числа 0 перед и после десятичной точки позволяется опускать. Последние два значения записаны в экспоненциальной форме, т. е. MeP эквивалентно записи M * (10 ** P). Таким образом, приведенные выше значения равны 234 и -0.125 соответственно.

Для чисел с плавающей точкой также определены арифметические операции и операции сравнения:

```
>>> 2.0 + 2.3 # сложение
4.3
>>> 4.5 - 3.2 # вычитание
1.2999999999999998
>>> 2.3 * 4.5 # умножение
10.35
>>> 5.6 / 3.2 # деление
1.7499999999999998
>>> 5.6 // 3.2 # целочисленное деление
1.0
>>> 4.3 % 3.2 # остаток от деления
1.0999999999999996
>>> 2.3 ** 1.5 # возведение в степень
```

```
3.488122704263713
>>> 3.2 > 2.3 # больше
True
>>> 3.2 < 2.3 # меньше
False
>>> 3.2 >= 4.5 # больше или равно
False
>>> 4.5 <= 5.6 # меньше или равно
True
>>> 3.4 == 3.4 # равно
True
```

Вообще, последний вид сравнения, а именно сравнение чисел типа float на равенство, редко дает удовлетворительный результат в виду того, что операции над такими числами имеют невысокую точность. В частности, например

```
>>> 14 * 0.1
1.4000000000000001
>>> .1 + .2
0.30000000000000004
```

Поэтому

```
>>> 14 * 0.1 == 1.4
False
```

хотя, очевидно, равенство должно соблюдаться. Чтобы избежать проблем при сравнении, обычно для чисел с плавающей точкой равенство определяют с заданной точностью, т. е. например, для равенства с абсолютной точностью не более 1e-8:

```
>>> abs(14 * 0.1 - 1.4) < 1e-8
True
```

Вообще, более правильным способом сравнения на равенство в данном случае будет способ, который основан на определении относительной точности:

```
>>> def isclose(a, b, rel_tol=1e-09, abs_tol=0.0):
    return abs(a-b) <= max(
         rel_tol * max(abs(a), abs(b)), abs_tol)
>>> isclose(14 * 0.1, 1.4)
True
```

Такое определение функции isclose() включено в модуль math стандартной библиотеки начиная с версии 3.5.

Поскольку, как правило, во внутреннем представлении числа с плавающей точкой соответствуют стандарту IEEE 754, результатом выполнения операций над ними могут оказаться специальные константы +/- бесконечность (inf), -0.0, не число (nan, Not-A-Number):

```
>>> 1e+2000
inf
>>> 1e-2000
0.0
>>> -1e-2000
-0.0
>>> -1e2000
-inf
>>> float('nan')
nan
>>> float('nan') + 2.0
nan
>>> float('nan') == float('nan')
False
```

Значительное количество математических функций и констант с плавающей точкой определено в модуле math стандартной библиотеки, например:

```
acos, acosh, asin, asinh, atan, atan2, atanh, ceil,
copysign, cos, cosh, degrees, e, erf, erfc, exp,
expm1, fabs, factorial, floor, fmod, frexp, fsum,
gamma, hypot, isinf, isnan, ldexp, lgamma, log, log10,
log1p, modf, pi, pow, radians, sin, sinh, sqrt, tan,
tanh, trunc.
```

Среди приведенных функций имеет смысл обратить внимание на функцию fsum, которая предназначена для вычисления суммы элементов последовательности с большей точностью, чем обычный оператор сложения. На разных платформах повышение точности достигается по-разному, но в большинстве случаев fsum() дает лучший результат.

```
>>> sum([.1] * 10)
0.9999999999999999
```

```
>>> math.fsum([.1] * 10)
1.0
```

При выполнении операций со значениями различных числовых типов выполняется неявное приведение, например:

```
>>> 2 + 2.0
4.0
```

Значения типа bool приводятся следующим образом:

```
>>> int(True)
1
>>> int(False)
0
>>> 1 + True
2
```

К числам можно применить ряд других встроенных функций:

```
>>> min(1, 2, 3) # наименьшее значение
1
>>> max(1, 2, -3) # наибольшее значение
2
>>> abs(-5) # абсолютное значение (модуль)
5
>>> pow(2, 3) # то же, что и **
8
>>> divmod(5, 2) # целая часть от деления и остаток
(2, 1)
```

Еще одним встроенным типом является тип complex, предназначенный для представления комплексных чисел. Комплексная единица задается константой 1j, соответственно, комплексные константы выглядят следующим образом:

```
2 + 1j   3 - 25J  -2.3 + 4.2j
```

Примеры работы с комплексными числами:

```
>>> type(1j)
<type 'complex'>
>>> 1j ** 2
(-1+0j)
>>> 1 + 2j * 5 - 3j
```

```
(1+7j)
>>> (1 + 2j) * (5 - 3j)
(11+7j)
>>> (2 + 3j).real
2.0
>>> (2 + 3j).imag
3.0
>>> abs(1 + 1j)
1.4142135623730951
```

Для работы с комплексными числами в стандартной библиотеке существует отдельный модуль cmath.

Кроме встроенных числовых типов стандартная библиотека Python включает два дополнительных типа для представления чисел. Тип decimal.Decimal предназначен для выполнения вычислений с действительными числами представленными в десятичной системе счисления (IBM's General Decimal Arithmetic Specification). Такое представление позволяет значительно повысить точность результата, однако при этом увеличивается время вычислений. Для данного типа переопределен стандартный набор арифметических операций:

```
>>> from decimal import Decimal
>>> pi = Decimal(3.1415926)
>>> 2 * pi *Decimal(5)
Decimal('31.415926000000000684053702 57')
```

Тип fractions.Fraction предназанчен для работы с обыкновенными дробями:

```
>>> from fractions import Fraction
>>> Fraction(2, 3)
Fraction(2, 3)
>>> Fraction(2, 4)
Fraction(1, 2)
>>> Fraction(7, 8) + Fraction(3, 4)
Fraction(13, 8)
>>> Fraction(7, 8).denominator
8
>>> Fraction(7, 8).numerator
7
```

Строки

Для представления строк в Python 2 используются два типа: str и unicode. Первый предназначен для хранения последовательностей символов из стандартного ASCII-набора, т. е. один символ занимает один байт в памяти. Специального символьного типа в Python нет, одиночный символ представляется строкой длины 1. Строковые константы оборачиваются в одинарные или двойные кавычки:

```
"Hello"  'Hello'
```

В случае, если строка заключена в кавычки одного типа, внутри нее могут встречаться кавычки другого типа:

```
"Say: 'Hello'!"
```

Синтаксис языка позволяет записывать строку как последовательность строковых констант, разделенный пробелами. В таком случае все константы конкатенируются в одну строку.

```
>>> "This" " " 'is' " " "a" " " 'string'
'This is a string'
```

Также предусмотрен специальный синтаксис для значений строкового типа, которые состоят из нескольких строк (также допустимы одинарные и двойные кавычки):

```
>>> """This
... is
... multiline
... string"""
'This\nis\nmultiline\nstring'
```

Последняя строка отображает однострочное представление данной строки: \n означает символ перевода строки. Вообще, комбинации символов, которые начинаются с символа '\' рассматриваются интерпретатором как один символ, в частности: \t означает символ табуляции, \" – двойную кавычку, \\ – просто \. Чтобы интерпретатор не воспринимал такие символы как специальные, можно использовать raw-строки, т. е. строки, начинающиеся с символа r:

```
>>> print 'AC\\DC'
AC\DC
>>> print r'AC\\DC'
```

```
AC\\DC
```

Такие строки часто используются для записи регулярных выражений.

Для получения символа по его ASCII-коду и наоборот используются следующие встроенные функции:

```
>>> chr(35)
'#'
>>> ord('$')
36
```

Для хранения строк с символами различных национальных алфавитов предназначен строковый тип unicode. Как следует из названия, в таких строках символы представлены в кодировке Unicode, и на каждый символ отводится 2 байта. Константы unicode представляются следующим образом:

```
u'Привет'u"Привет"
```

Получить Unicode-символ по его коду можно следующим образом:

```
>>> unichr(2323)
u'\u0913'
>>> print unichr(324)
ń
```

Поскольку представление строковых данных в Unicode несколько избыточно, в частности символы из набора ASCII требуют использования в 2 раза большего объема памяти, для передачи и хранения данных в настоящее время применяется специальная кодировка UTF-8. В этом случае символ может занимать от 1 до 4 байтов:

Диапазон кодов символа	Представление в UTF-8
0 — 7Fh	0xxxxxxx
80h — 7FFh	110xxxxx 10xxxxxx
800h — FFFFh	1110xxxx 10xxxxxx 10xxxxxx
1000h — 1FFFFh	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Такой способ значительно экономнее, но неудобен при обработке, т. к. каждый символ может иметь различную длину и чтобы найти n-й символ в строке, необходимо проанализировать предыдущие n-1. Поэтому при необходимости обработки данных, которые получены, например, из интернета или из базы данных в кодировке UTF-8 необходимо перекодировать. Для прямой и обратной перекодировки строк используются функции decode()/encode():

```
>>> u'Привет'.encode('utf-8')
'\xd0\x9f\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'
>>> s = u'Привет'.encode('utf-8')
>>> print s.decode('utf-8')
Привет
```

В Python 3 работа строковые типы имеют несколько иное представление. В частности, основной тип для хранения строк сохранил имя str, однако, в новой версии он предназначен для хранения строк в кодировке Unicode предоставляя 2 или 4 байта для представления одного символа (см. PEP 393). В то же время, для хранения байтовых последовательностей используется тип bytes. Соответственно, константы типа bytes имеют специальный префикс.

```
>>> type(b'hello')
<class 'bytes'>
```

Префикс u для строковых констант присутствует в Python 3 для совместимости со старыми версиями, но не имеет никакого смысла (см. PEP 414).

Таким образом, типу str в Python 3 соответствует тип bytes, а типу unicode — str. Также, в Python 3 отсутствует функция unichr(), а функции chr() и ord() работает с символами Unicode.

```
>>> ord('Я')
1071
>>> chr(1071)
'Я'
```

Список поддерживаемых в стандартной библиотеке кодировок можно найти по адресу:

https://docs.python.org/3/library/codecs.html#standard-encodings

Для строк определен ряд операций:

```
>>> 'foot' + 'ball'
'football'
```

```
>>> 'Say "' + 'A' * 10 + '"'
'Say "AAAAAAAAAA"'
>>> 'ball' in 'football'
True
>>> 'football'[3] # нумерация с 0
't'
```

Длина строки определяется с помощью функции len():

```
>>> len('football')
8
```

Для выделения подстроки из строки используется оператор среза:

```
>>> 'football'[0:4]
'foot'
>>> 'football'[:4]
'foot'
>>> 'football'[4:]
'ball'
>>> 'football'[-4:]
'ball'
>>> 'football'[::2] # третий параметр — это шаг
'fobl'
>>> 'football'[::-1]
'llabtoof'
```

Следует отметить, что строки являются неизменяемыми, т. е. при выполнении любой операции над ними создается новая строка. Поэтому построение строк с помощью конкатенации в цикле является исключительно неэффективным.

При необходимости формирования строки на основании значений нескольких переменных рекомендуется использовать оператор (функцию) форматирования строк:

```
>>> 'Name: %s; age: %d' % ('Bob', 22)
'Name: Bob; age: 22'
>>> 'Name: {}; age: {}'.format('Bob', 22)
'Name: Bob; age: 22'
```

Подробнее шаблоны форматирования описаны в официальной документации:

https://docs.python.org/2/library/string.html#formatspec

Для строк определены функции сравнения, причем строки

сравниваются посимвольно в соответствии с порядком символов в кодовой таблице:

```
>>> 'A' > 'B'
False
>>> 'a' > 'A'
True
>>> "a" > "aa"
False
>>> 'boat' > 'board'
True
>>> '1' < '2'
True
>>> '11' < '2' # сравниваются коды символов, а не числа
True
```

Кроме описанных выше, строковые объекты в Python имеют ряд других удобных встроенных функций, в частности:

```
>>> "abrakadabra".count("ab")
2
>>> "football".endswith("ball")
True
>>> "football".startswith("foot")
True
>>> "abrakadabra".find("ab") # поиск слева
0
>>> "abrakadabra".rfind("ab") # поиск справа
7
>>> "abrakadabra".find("ks") # -1, если нет
-1
>>> "abrakadabra".find("ka")
4
>>> "abrakadabra".index("ks") # ValueError, если нет
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> "abrakadabra".index("ka")
4
>>> "abc123".isalnum()
True
>>> "abc".isalpha()
True
>>> "123".isalpha()
```

```
False
>>> "123".isdigit()
True
>>> "ACDC".islower()
False
>>> "ACDC".isupper()
True
>>> "python".capitalize()
'Python'
>>> "acdc".upper()
'ACDC'
>>> "ACDC".lower()
'acdc'
>>> "aBcD".swapcase()
'AbCd'
>>> "abrakadabra".partition("ka")
('abra', 'ka', 'dabra')
>>> "football".replace("foot", "hand")
'handball'
>>> "1,2,3".split(",")
['1', '2', '3']
>>> ",".join(['1', '2', '3'])
'1,2,3'
```

Последнюю функцию часто применяют при склеивании списков строк, чтобы не использовать медленную операцию конкатенации:

```
>>> months = ['January', 'February', 'March', 'April',
'May',   'June',   'July',   'August',   'September',
'October', 'November', 'December']
>>> ' '.join(months)
'January February March April May June July August
September October November December'
```

Дополнительный набор функций описан в модуле string, там же определен ряд полезных констант:

```
>>> import string
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

```
>>> string.digits
'0123456789'
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

Константы, определенные в модуле string могут быть использованы, например, для генерации случайных последовательностей символов, таких как пароли.

```
>>> import random
>>> import string
>>> ''.join([random.choice(string.ascii_uppercase) for
x in range(10)])
'JEEOTOSOJY'
>>> ''.join(random.sample(string.ascii_uppercase, 10))
'KALFMIXJWN'
```

Функция random.choice() возвращает случайный элемент заданного контейнера, а random.sample() генерирует указанное во втором параметре количество случайных элементов контейнера без повторений. Следует, однако, помнить, что пароли, полученные на основе генератора случайных чисел не являются надежными.

При обработке текстов на естественных языках часто возникает необходимость удаления символов пунктуации. Для этого удобно использовать функцию translate(), предназначенную для конвертации строк из одной кодировки в другую. В Python 2 данная функция в качестве первого параметра получает строку длиной в 256 символов, каждый из которых используется для замены символа с кодом, соответствующему его индексу. Второй параметр содержит символы, подлежащие удалению. Таким образом, удаление символов пунктуации можно выполнить так:

```
>>> "abc.sdf,gf:".translate(None, string.punctuation)
'abcsdfgf'
```

В Python 3 строки представляются в кодировке Unicode, поэтому функция translate() работает несколько иначе. В качестве аргумента она получает словарь, в котором ключами являются коды заменяемых символов, а значениями — символы, на которые выполняется замена.

```
>>> "abc.sdf,gf:".translate(
    {ord(c): '' for c in string.punctuation}
)
'abcsdfgf'
```

Мощным инструментом поиска строковых данных являются регулярные выражения. Основные функции для работы с ними определены в модуле стандартной библиотеки re. Регулярные выражения в Python имеют синтаксис, подобный регулярным выражениям языка Perl. Подробное описание последовательностей символов, которые поддерживаются модулем re приведено в официальной документации: https://docs.python.org/2/library/re.html#re-syntax. Функция match() позволяет определить, соответствует ли начало заданной строки определенному регулярному выражению.

```
>>> import re
>>> re.match(r'[0-9]+', '12345')
<_sre.SRE_Match object at 0xb74a7b10>
>>> re.match(r'[0-9]+', 'a12345')
>>> p = re.compile(r'[0-9]+')
>>> p.match('12345')
<_sre.SRE_Match object at 0xb70c9d40>
>>> p.match('a12345')
```

В данном случае регулярное выражение соответствует последовательности десятичных цифр, содержащей хотя бы одну цифру. Если соответствие обнаружено, возвращается match-объект, иначе — None. Если регулярное выражение планируется использовать многократно, для увеличения производительности его можно предварительно откомпилировать при помощи функции complile().

Для поиска фрагментов строки, соответствующих регулярному выражению могут быть использованы функции search() и findall().

```
>>> m = p.search('(044) 222-11-11')
>>> m.group()
'044'
>>> m = p.findall('(044) 222-11-11')
>>> m
['044', '222', '11', '11']
```

Операторы

Операторы предназначены для описания действий, которые необходимо выполнить интерпретатору. Все операторы за исключением оператора присваивания представляют собой зарезервированные идентификаторы, которые называются ключевыми словами. Список ключевых слов можно получить следующим образом:

```
>>> import keyword
>>> keyword.kwlist
['and', 'as', 'assert', 'break', 'class', 'continue',
'def', 'del', 'elif', 'else', 'except', 'exec',
'finally', 'for', 'from', 'global', 'if', 'import',
'in', 'is', 'lambda', 'not', 'or', 'pass', 'print',
'raise', 'return', 'try', 'while', 'with', 'yield']
```

Оператор присваивания имеет следующую структуру:

<идентификатор> = <выражение>

<идентификатор> – это имя переменной, которой в результате выполнения оператора будет присвоено значение. Идентификатор представляет собой последовательность латинских букв (в Python 3 допустимо использование Unicode), цифр и символа '_', которая не начинается с цифры. Регистр букв различается, т. е. X и x – разные идентификаторы. Если идентификатор в левой части оператора присваивания в программе встречается первый раз, создается новая переменная с соответствующим именем. Предварительно объявлять переменные как в языках программирования со статической типизацией нет необходимости. Переменные в Python являются ссылками на объекты, поэтому тип переменной фактически определяется как тип объекта, на который она ссылается. Сама переменная не имеет определенного типа.

```
>>> a = 1
>>> type(a)
<class 'int'>
>>> a = 'hello'
>>> type(a)
<class 'str'>
```

Для инициализации нескольких переменных существует специальная форма оператора:

```
>>> a = b = c = 0
>>> a
0
>>> b
0
>>> c
0
```

Следует отметить, что это — отдельная синтаксическая конструкция, а не присваивание результата оператора присваивания, как, например, в языке C. В Python оператор присваивания результата не возвращает.

Также, существует сокращенная форма оператора присваивания для случаев, когда переменная-результат входит в выражение, которое вычисляется:

```
counter += 1           # counter = counter + 1
a -= 1                 # a = a - 1
factorial *= (k + 1)   # factorial =
                       # factorial * (k + 1)
d /= 10                # d = d / 10, число без
                       # последней цифры
d %= 10                # d = d % 10, последняя
                       # цифра числа
```

Операторы декремента и инкремента ++ и --, присутствующие в синтаксисе C-подобных языков программирования в Python не используются. Удалить созданную ранее переменную можно при помощи оператора del:

```
>>> a = 5
>>> del a
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

Для изменения последовательности выполнения операторов в зависимости от значения определенного условного выражения используется оператор ветвления:

if <условие1>:
<отступ><оператор>
<отступ><оператор>
<отступ>…
elif <условие2>:
<отступ><оператор>
<отступ><оператор>
<отступ>…
elif ...
else:
<отступ><оператор>
<отступ><оператор>

<отступ>…

<условие1>, <условие2> и т. д. представляют собой выражения, результат вычисления которых либо имеет тип bool, либо неявно приводится к типу bool. Если результат вычисления <условие1> истинен, еще говорят «условие выполняется», будет выполнена только первая секция операторов до оператора elif. Тоже касается и всех остальных условий. Если ни одно из условий не выполняется, будет выполнена секция else. Секция elif может повторяться сколько угодно раз, при этом ни elif, ни else не являются обязательными. При истинности какого-либо из условий оператора ветвления остальные условия не вычисляются, и, после выполнения соответствующего набора операторов, управление передается следующему оператору после оператора ветвления. Секция операторов или операторный блок, который соответствует определенной ветке оператора if, задается одинаковым отступом, который предшествует каждому оператору блока. В соответствии с PEP8 один отступ должен быть равен 4 пробелам.

Пример использования оператора ветвления:

```
>>> a = 1
>>> b = 2
>>> c = 1
>>> D = b * b - 4 * a * c
>>> if D < 0:
...     print "Корней нет"
... elif D == 0:
...     print "1 корень"
... else:
...     print "2 корня"
...
1 корень
```

Неполный вариант условия:

```
>>> age = 18
>>> if age > 16:
...     print "Учиться никогда не поздно"
...
Учиться никогда не поздно
```

Вложенные условия:

```
>>> age = 5
>>> gender = 'M'
>>> if age < 7:
```

```
...     if gender == 'F':
...             print "Девочка"
...     else:
...             print "Мальчик"
... else:
...     print "Уже поздно"
...
Мальчик
```

Составные логические выражения:

```
>>> age = 15
>>> if 20 < age < 60:
...     print "Отлично"
... elif age <= 20 or age >= 60:
...     print "Тоже ничего"
... else:
...     print "Так не бывает"
...
Тоже ничего
```

Для случая, когда оператор ветвления служит для определения значения одной переменной, например:

```
if x < 0:
    y = -x
else:
    y = x
```

может быть использован укороченный вариант записи:

```
y = -x if x < 0 else x
```

Особым вариантом оператора ветвления можно считать оператор обработки исключительных ситуаций try/except. Синтаксис оператора выглядит следующем образом:

try:
<отступ><оператор> # Здесь может
<отступ><оператор> # быть ошибка
…
except E1 as e:
<отступ><оператор> # выполняется,
<отступ><оператор> # когда ошибка E1
…
except (E2, E3):

<отступ><оператор> # выполняется,
<отступ><оператор> # когда ошибка E2 или E3
…
else:
<отступ><оператор> # выполняется,
<отступ><оператор> # когда всё Ok
…
finally:
<отступ><оператор> # выполняется
<отступ><оператор> # всегда
…

Под исключительной ситуацией или исключением принято понимать какую-либо ошибку, возникающую при выполнении программы. В принципе, любую возможную ошибку, кроме синтаксических, т. е. SyntaxError, можно обработать. Для этого достаточно операторный блок, который содержит потенциально ошибочные операторы, обернуть в секцию между try и except. Как правило, данный способ обработки ошибок используется, когда заранее предсказать наличие ошибки невозможно, например, при открытии файла, конвертации данных, введенных пользователем и т. п. Каждая ошибка в зависимости от ситуации, в которой она возникла, порождает свой тип исключения. Все исключения являются объектами классов, унаследованных от встроенного класса BaseException. В операторе except, за которым следует код обработки ошибки, необходимо указать тип исключения, который перехватывается данным оператором. Наиболее частыми типами ошибок являются:

ValueError – ошибка в значении входного параметра, возникает, например, при конвертации типов;

TypeError – возникает при несовместимости типов;

KeyError, IndexError — ошибка в ключе или индексе при доступе к элементам контейнера;

NameError – обращение к несуществующей переменной, идентификатор не определен;

IOError – ошибка ввода/вывода, может возникать при работе с файлами.

Пример кода обработки ошибок:

```
>>> l = [1, 2, 3]
>>> try:
   a = l[int(input("Input index "))]
except ValueError:
   print('Incorrect value')
except IndexError:
```

```
   print('Incorrect index')
Input index 5
Incorrect index
```

В данном примере выполняется попытка интерпретировать ввод пользователя как индекс элемента в списке. Для этого в коде выполняется приведение введенного значения к целому типу, а затем получение значения по индексу. В первом случае, при ошибочном вводе будет сгенерировано исключение ValueError, а во втором — IndexError. Соответственно, для перехвата каждой из этих ошибок в коде предусмотрена ветка except, в которой выводится сообщение об ошибке. Синтаксис языка предполагает, что в операторе except конкретный тип ошибки может быть не указан и, в таком случае, будет перехвачена любая ошибка. Такой вариант использования данного оператора является крайне нежелательным, т. к. в этом случае маскируются любые ошибки, а не только те, которые ожидаются в данном блоке. Кроме того, except без типа исключения перехватывает ошибки начиная с самого верхнего уровня, т. е. BaseException. Это означает, что кроме ошибок, связанных с бизнес-логикой программы, будут перехватываться ошибки KeyboardInterrupt и SystemExit, которые вызваны принудительным прерыванием программы. В результате, выполнение программы невозможно будет прервать извне.

Вообще, при перехвате исключений выбирается ветка except, тип исключения которой либо идентичен сгенерированной ошибке, либо является суперклассом для нее. В случае, если возникает необходимость перехватить все возможные ошибки, в операторе except необходимо указать тип Exception, который унаследован от BaseException, но не включает ошибки, связанные с прерыванием программы. Когда в операторе except используется ключевое слово as, например, except ValueError as e, переменная e при возникновении ошибки будет содержать в себе экземпляр класса порожденного исключения. Т.е. корректная обработка всех возможных ошибок должна выполнятся так:

```
>>> try:
    int('a')
except Exception as e:
    print(e)
invalid literal for int() with base 10: 'a'
```

Т. к. объекты исключений сопоставляются с суперклассами, а не только с идентичными исключениями, последовательность типов исключений в ветках except должна начинаться с самых нижних по

иерархии наследования. Например,

```
>>> try:
    2 / 0
except ArithmeticError:
    print("Arithmetic Error")
except ZeroDivisionError:
    print("Zero Error")
Arithmetic Error
```

В данном случае, ветка ZeroDivisionError является недосягаемой, поскольку это исключение наследуется от ArithmeticError.

Секции else и finally не являются обязательными. Первая из них предназначена для описания действий в случае, если ошибка не произошла, а вторая – выполняется в любом случае обязательно, вне зависимости от наличия ошибки. finally обычно используется для закрытия файлов и освобождения других ресурсов. Допустимо использование finally без except, но в таком случае ошибка перехвачена не будет.

```
>>> try:
    2/0
finally:
    print("Ok")
Ok
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
```

Из примера видно, что блок finally был выполнен, но сообщение об ошибке все равно было выведено на экран. Исключения можно создавать принудительно с помощью оператора raise, например:

```
raise NameError
```

На самом деле, данный оператор эквивалентен следующему.

```
raise NameError()
```

Т.е. при выбросе исключения создается экземпляр указанного класса исключения, при этом в скобках можно указать сообщение, которое описывает возникшую ошибку.

```
>>> raise ValueError("My error message")
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
ValueError: My error message
```

Иногда возникает необходимость обработать выброшенное исключение в месте его возникновения, а затем пробросить его выше для последующей обработки, например:

```
>>> try:
...     5 / 0
... except ZeroDivisionError as e:
...     print e
...     raise ZeroDivisionError
integer division or modulo by zero
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
ZeroDivisionError
```

Данный способ работает, но, как видно из примера, сообщение об ошибке содержит номер строки, в которой содержится raise, а не номер ошибочной строки. Для того, чтобы сохранить полную информацию о возникшей ошибке, можно воспользоваться оператором raise без параметров. Следует отметить, что такая форма оператора raise возможна только внутри блока except.

```
>>> try:
...     5 / 0
... except ZeroDivisionError as e:
...     print e
...     raise
integer division or modulo by zero
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: integer division or modulo by zero
```

Рассмотрим пример использования перехвата исключений:

```
try:
     a = 5 / x
#     raise ValueError
except ZeroDivisionError:
     print "Ошибка"
     exit()
except NameError:
     print "x отсутствует"
else:
```

```
     print a
finally:
     print "Конец"
print "Самый настоящий конец"
```

При x = 1, вывод программы будет таким:

```
5
Конец
Самый настоящий конец
```

Поскольку в секции try никаких ошибок не возникло, интерпретатор сразу после выполнения деления перешел к секции else и вывел результат. После этого в обязательном порядке была выполнена секция finally, а затем последний оператор print.

Если перед запуском программы переменная x не была определена, то в секции try возникнет исключение NameError и вывод программы будет выглядеть следующим образом:

```
x отсутствует
Конец
Самый настоящий конец
```

Снова обязательно выполнилась секция finally и отработал оператор print.

Если x = 0, интерпретатор сразу перейдет в первую секцию except, предназначенную для обработки ошибки деления на 0, в которой выводится соответствующее сообщение и выполнятся принудительный выход из программы. После этого снова выполнится секция finally. Вывод программы:

```
Ошибка
Конец
```

Наконец, если, при x = 1, раскомментировать оператор raise, будет выброшено исключение, обработка которого не предусмотрена. Интерпретатор остановит выполнение программы и выдаст собственное сообщение об ошибке. При этом секция finally все равно выполнится:

```
Конец
Traceback (most recent call last):
  File "except.py", line 6, in <module>
    raise ValueError
```

```
ValueError
```

Для организации многократного повторения операторного блока используется оператор цикла while:

while <условие>:
<отступ><оператор>
<отступ><оператор>
…

В данном случае операторный блок или тело цикла будет повторяться до тех пор, пока <условие> истинно. Например, данный код позволяет вывести все натуральные числа, квадрат которых меньше 50:

```
>>> i = 1
>>> while i * i < 50:
...     print "%d * %d = %d" % (i, i, i * i)
...     i += 1
...
1 * 1 = 1
2 * 2 = 4
3 * 3 = 9
4 * 4 = 16
5 * 5 = 25
6 * 6 = 36
7 * 7 = 49
```

При написании циклов следует особенно внимательно следить за тем, чтобы хотя бы одна переменная, участвующая в условном выражении изменялась в теле цикла, иначе выполнение цикла будет бесконечным. Пример бесконечного цикла:

```
while True:
    pass
```

Ключевое слово pass означает, что операторный блок пуст, т. е. ничего выполнять не надо.

Если существуют другие причины выхода из цикла, кроме ложного значения условия в операторе while, может быть использован оператор break. Данный оператор приводит к принудительному прекращению выполнения цикла. Также для принудительного прекращения выполнения текущей итерации и возврата к оператору while используется оператор continue. Ниже показано применение обоих операторов:

```
import random
secret = random.randint(1, 10)
while True:
     guess = raw_input(“?”)
     try:
          guess = int(guess)
     except ValueError:
          continue
if guess == secret:
          print “=”
          break
     elif guess > secret:
          print “>”
     else:
          print “<”
```

В данном примере реализована игра «Угадай число». Пользователь вводит свою догадку, а программа сообщает ему, является ли она большей, меньшей или равной загаданному числу. Для загадывания числа, в программе использован генератор псевдослучайных чисел, описанный в стандартной библиотеке языка Python. Для этого в первых двух строках программы был проимпортирован модуль random, а затем была вызвана функция random.randint(), которая возвращает целое число в заданных пределах. После этого, поскольку заранее не известно, с какой попытки пользователь угадает число, запускается бесконечный цикл. В нем выполняется ввод догадки с клавиатуры и производится попытка преобразования введенного числа к типу int. Если пользователь ошибся и ввел не число, а какую-либо другую последовательность символов, выброшенное исключение, будет обработано в соответствующем операторе except. Это приведет к выполнению оператора continue, в результате чего интерпретатор прекратит выполнение тела цикла и снова перейдет к оператору while, позволяя пользователю ввести новое число. Далее происходит проверка, равно ли введенное число загаданному. Если да, выводится сообщение об окончании игры и цикл прерывается, что в данном случае приводит к завершению программы. Иначе на экран выводится соответствующая подсказка.

Часто при разработке программ случаются ситуации, когда необходимо перебрать все элементы какого-либо контейнера. В таком случае количество повторений заранее известно, соответственно применение цикла while несколько избыточно. Поэтому, для прохода по содержимому контейнера существует специальный тип цикла:

for <переменная> in <контейнер>:

<отступ><оператор>
<отступ><оператор>
…

На данный момент мы рассмотрели только один вид контейнеров – строки, которые являются последовательностями символов, поэтому рассмотрим использование цикла for на примере строк:

```
>>> for i in "Hello":
...     print i
...
H
e
l
l
o
```

Оператор цикла for, подобно оператору присваивания, создает новую переменную, которая после выхода из цикла будет хранить последнее полученное значение. Аналогично циклам while в циклах for можно использовать операторы break и continue. В общем случае при завершении цикла невозможно узнать, по какой причине произошел выход: было достигнуто условие завершения цикла или был выполнен оператор break. Чтобы упростить задачу определения причины выхода из цикла, в Python возможно использование оператора else сразу после тела цикла. Блок else будет выполнен только в случае, если цикл принудительно не завершался. В качестве примера, можно рассмотреть программу для определения вхождения символа в строку (в реальных программах для решения этой задачи используется оператор in):

```
str = 'Hello'
for c in str:
    if c == x:
        print 'Найден'
        break
else:
    print 'Не найден'
```

Для разделения текста программы на отдельные смысловые блоки и исключения дублирования программного кода в Python используются функции. Функция определяется при помощи оператора def:

def <имя>(<список параметров>):
<отступ><оператор>

<отступ><оператор>

…

Именем функции может быть любой допустимый идентификатор, а список параметров содержит набор переменных, разделенных символом ','. Для выхода из функции и возвращения результата используется оператор return. Если return отсутствует или после него не задано возвращаемое значение, то функция возвращает None. Пример определения функции:

```
>>> def add(x, y):
...     return x + y
```

Здесь определена функция add, которая принимает два параметра x и y. Результатом выполнения функции будет сумма x и y, которая возвращается оператором return.

Примеры вызова функции:

```
>>> add(2, 3)
5
>>> add(4, -4)
0
>>> a = 1
>>> b = 2
>>> add(a, b)
3
```

В последнем случае при вызове функции значения переменных a и b подставляются вместо значений параметров x и y.

Подробнее использование функций описано в разделе «Функции» ниже.

Контейнеры

Одним из преимуществ языка Python является наличие целого ряда встроенных контейнеров, имеющих удобный и унифицированный интерфейс. Одним из примеров контейнеров являются строки рассмотренные ранее. Другим, не менее популярным типом контейнера является список. Для создания списков используется специальный синтаксис:

```
>>> l = [1, 2, 3]
>>> l
[1, 2, 3]
```

Элементы списка могут быть разных типов и даже содержать в себе другие списки:

```
>>> l = [1, -0.005, 2 + 4j, 'abc', [1, 2]]
```

Другим способом создания списка является использование функции-конструктора list:

```
>>> list('Hello')
['H', 'e', 'l', 'l', 'o']
```

Параметром данной функции должен быть другой контейнер. Список может быть пустым:

```
>>> l = []
>>> len(l)
0
```

Как видно из последнего примера, функция len() одинаково работает как со списками, так и со строками. Одним из положительных качеств Python является идентичный интерфейс для всех типов контейнеров:

```
>>> [1, 3, 5] + [2, 4]
[1, 3, 5, 2, 4]
>>> [1, 2, 3, 4][2]
3
>>> [1, 2, 3, 4][0:2]
[1, 2]
>>> [1, 2, 3, 4][3:]
[4]
>>> [1, 2, 3, 4][::-1]
[4, 3, 2, 1]
>>> [1] * 10
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Следует обратить внимание на то, что пустой список приводится к bool следующим образом:

```
>>> bool([])
False
```

В связи с этим, идиоматической конструкцией проверки списка, как и любого другого контейнера, на пустоту является следующая:

```
if l:
    …
```

Тем не менее

```
>>> bool([""])
True
```

т. к. несмотря на то, что внутри списка пустая строка, сам список пустым не является.

При работе со списками следует учитывать, что они являются изменяемыми:

```
>>> l = [1, 2, 3]
>>> l[1:2] = [5, 6, 7]
>>> l
[1, 5, 6, 7, 3]
>>> l[0] = 0
>>> l
[0, 5, 6, 7, 3]
```

Поскольку списки изменяемы, а переменные в Python являются указателями на значения, если несколько переменных ссылаются на один и тот же список, изменение содержимого одной переменной приведет к изменению всех:

```
>>> l = [1, 2, 3]
>>> m = l
>>> m[0] = -1
>>> l
[-1, 2, 3]
```

Подобный эффект может имеет место и при попытке создать список списков при помощи операции умножения списка на число:

```
l = [[1]]*10
>>> l[0][0] = 0
>>> l
[[0], [0], [0], [0], [0], [0], [0], [0], [0], [0]]
```

Чтобы избежать подобных неожиданностей, при изменении списков рекомендуется делать их копию. Копию можно сделать при помощи функции copy() модуля стандартной библиотеки copy:

```
>>> l = [1, 2, 3]
```

```
>>> from copy import copy
>>> m = copy(l)
>>> m[0] = -1
>>> l
[1, 2, 3]
```

Более коротко данную задачу можно решить при помощи среза, который также возвращает копию, или при помощи функции list:

```
>>> l[:]
[1, 2, 3]
>>> list(l)
[1, 2, 3]
```

В Python 3 также существует эквивалентный встроенный метод:

```
>>> l.copy()
[1, 2, 3]
```

Данные методы не работают в случае, если приходится иметь дело со вложенными списками:

```
>>> l = [1, [2, 3], 4]
>>> m = copy(l)
>>> m[1][1] = 5
>>> l
[1, [2, 5], 4]
```

Т. к. вложенный список во внешнем списке представляется ссылкой, то в копии будет ссылка на тот же элемент, что и в оригинальном списке. Для полного копирования вложенных списков необходимо пользоваться функцией deepcopy():

```
>>> from copy import deepcopy
>>> l = [1, [2, 3], 4]
>>> m = deepcopy(l)
>>> m[1][1] = 5
>>> l
[1, [2, 3], 4]
>>> m
[1, [2, 5], 4]
```

Поскольку список хранит в себе ссылки на элементы, возможно применение следующих конструкций:

```
>>> l = [1, 2, 3]
>>> l.append(l)
>>> l
[1, 2, 3, [...]]
>>> l[3]
[1, 2, 3, [...]]
>>> l[3].append(4)
>>> l
[1, 2, 3, [...], 4]
>>> l is l[3]
True
```

В данном случае список содержит ссылку на самого себя, поэтому добавление элемента во вложенный список приводит к изменению оригинального списка.

Для создания списков, состоящих из последовательного набора целых чисел, например, для использования в цикле for, можно применить функции range() и xrange():

```
>>> range(10) # 10 элементов, от 0 до 9
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(3, 13)
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
>>> range(3, 13, 2) # 2 — приращение
[3, 5, 7, 9, 11]
>>> range(3, 13, -3)
[]
>>> range(13, 3, -3)
[13, 10, 7, 4]
>>> for i in range(3):
...     print i, i ** 2
...
0 0
1 1
2 4
>>> for i in xrange(3):
...     print i, i ** 2
...
0 0
1 1
2 4
```

Функция xrange() отличается от range() тем, что вместо списка она возвращает объект xrange, который работает аналогично итератору, т. е. при каждом обращении возвращает следующее

значение из указанного диапазона. Эта функция может быть значительно эффективнее range() в случае обхода списка большого объема в цикле for. В Python 3 традиционная функция range() отсутствует, а xrange() переименована в range().

Удобным способом создания и обработки списков является использование списочных выражений:

```
>>> [i for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [i ** 2 for i in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [i ** 2 for i in range(10) if i % 2]
[1, 9, 25, 49, 81]
>>> [x + y for x in range(3) for y in range(3)]
[0, 1, 2, 1, 2, 3, 2, 3, 4]
```

Данный способ генерации списков достаточно читабельный и, обычно, более эффективен, чем использование функций map() и filter() в подобном контексте. Объект типа список имеет необходимый набор встроенных функций:

```
>>> l = [1, 2, 3]
>>> l.append(4) # добавление элемента
>>> l
[1, 2, 3, 4]
>>> l.append(4)
>>> l.count(4) # количество элементов
2
>>> l.extend([5, 6, 7]) # добавление списка
>>> l
[1, 2, 3, 4, 4, 5, 6, 7]
>>> l.index(3) # индекс элемента
2
>>> l.index(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 10 is not in list
>>> l.insert(3, 10) # вставка в позицию 3
>>> l
[1, 2, 3, 10, 4, 4, 5, 6, 7]
>>> l.pop() # удаление элемента с конца
7
>>> l
[1, 2, 3, 10, 4, 4, 5, 6]
>>> l.pop(3) # удаление по индексу
```

```
10
>>> l
[1, 2, 3, 4, 4, 5, 6]
>>> l.remove(5) # удаление по значению
>>> l
[1, 2, 3, 4, 4, 6]
>>> l.remove(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
>>> l.reverse()
>>> l
[6, 4, 4, 3, 2, 1]
>>> l.sort()
>>> l
[1, 2, 3, 4, 4, 6]
```

В Python 3 интерфейс списков дополнен методом clear() для удаления всех элементов и ранее рассмотренным методом copy(). Также при применении метода sort() в Python 3 следует иметь в виду, что значения разных типов в этой версии языка не всегда являются сравнимыми, поэтому использование данного метода может приводить к ошибке.

```
>>> l = [1, 2, 3, 'a']
>>> l.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() < int()
```

Неизменяемым аналогом списков являются кортежи. Конструктором кортежа в Python является символ ',', но для лучшей читабельности или для указания приоритета операций, кортежи, как правило, заключаются в круглые скобки.

```
>>> t = 1, 2, 3
>>> t
(1, 2, 3)
>>> type(t)
<type 'tuple'>
>>> t = tuple([1, 2, 3])
>>> t
(1, 2, 3)
```

Т. к. кортежи неизменяемы, они обладают рядом преимуществ

перед списками, в частности: их обработка выполняется быстрее, они могут быть использованы в качестве ключей словаря (являются хешируемыми) и они обеспечивают константность элементов. Относительно константности следует отметить, что неизменяемыми являются ссылки на элементы, содержащиеся в кортеже. Если же сами эти элементы — изменяемые, они могут быть изменены:

```
>>> t = 1, [2, 3], 4
>>> t[1][0] = 0
>>> t
(1, [0, 3], 4)
```

Как видно из примера, к кортежам применима операция получения элемента по индексу, но только для чтения, аналогично могут быть использованы срезы и функция len(). Кортежи и списки могут быть распакованы:

```
>>> width, length, height = 100, 200, 300
>>> width
100
>>> length
200
>>> height
300
```

На основе распаковки кортежей основан идиоматический способ обмена значениями двух переменных:

```
>>> a = 1
>>> b = 2
>>> a, b = b, a
>>> a
2
>>> b
1
```

Распаковка кортежей также может быть использована в операторе for, в частности, встроенная функция enumerate() в качестве параметра получает контейнер, а возвращает набор кортежей, первый элемент которого является индексом элемента, а второй — самим элементом:

```
>>> for index, value in enumerate("Hello"):
...     print index, value
...
0 H
```

```
1 e
2 l
3 l
4 o
```

Эта функция очень удобна в том случае, когда вместе с элементом необходимо получить и его индекс. Функция enumerate() может принимать второй параметр, который указывает начальное значение индекса:

```
>>> list(enumerate('abc', 10))
[(10, 'a'), (11, 'b'), (12, 'c')]
```

В Python 3 есть возможность неполной распаковки:

```
>>> l = [1, 2, 3, 4, 5]
>>> a, b, *c = l
>>> c
[3, 4, 5]
```

Переменная c в данном случае получит все нераспакованные элементы в виде списка. Из встроенных функций кортежи поддерживают только count() и index(), которые не изменяют значение кортежа. Иногда кортежи используют для возвращения нескольких значений из функции.

```
>>> dm = divmod(5, 2) # встроенная функция
>>> dm
(2, 1)
>>> def f(a, b): # пользовательская функция
...     return a + b, a * b
...
>>> p, s = f(2, 3)
>>> p, s
(5, 6)
```

Строки, списки и кортежи являются упорядоченными контейнерами, т. е. сохраняют порядок, в котором в них были добавлены элементы. В Python также существуют неупорядоченные контейнеры, к которым относятся словари и множества. Словари позволяют хранить пары ключ-значение, причем доступ к элементу по ключу выполняется очень быстро, сравнимо с доступом к элементу списка по индексу. Поскольку в основе словаря лежит хеш-таблица, ключами в словаре могут быть только неизменяемые элементы: числа,

строки, кортежи и т. д. Словари создаются следующим образом:

```
>>> d = {} # пустой словарь
>>> d
{}
>>> type(d)
<class 'dict'>
>>> d = {'a': 1, 2: 'b'}
>>> d = dict(zip(['a', 'b'], [1, 2]))
>>> d
{'a': 1, 'b': 2}
```

Поиск элемента по ключу выполняется с помощью квадратных скобок:

```
>>> d['a']
1
>>> d['b'] # несуществующий ключ
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'b'
>>> d['b'] = 3 # добавление элемента
>>> d
{'b': 3, 2: 'b', 'a': 1}
>>> 'b' in d # вхождение элемента в словарь
True
```

При использовании оператора for итерирование выполняется по ключам. Чтобы получить сразу ключ и значение можно воспользоваться функцией items():

```
>>> for key in d:
...     print key, d[key]
...
a 1
c 3
b 2
>>> for key, value in d.items():
...     print key, value
...
a 1
c 3
b 2
```

Для создания словаря на основе содержимого другого контейнера,

например, списка, могут быть использованы выражения, подобные списочным:

```
>>> d = {x: x ** 2 for x in range(10)}
>>> d
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49,
8: 64, 9: 81}
>>> d[7]
49
```

В данном случае был создан словарь, ключами которого являются натуральные числа, а значениями — их квадраты.

Как правило, обращение к словарю предполагает проверку на наличие ключа. Так, при получении либо значения по ключу, либо значения по умолчанию обычно используется следующий код:

```
if key in d:
    value = d[key]
else:
    value = DEFAULT_VALUE
```

Его можно заменить одной строкой кода благодаря методу словаря get():

```
value = d.get(key, DEFAULT_VALUE)
```

При отсутствии второго параметра, если ключа в словаре нет, данный метод вернет значение None. Аналогично удалить элемент без предварительной проверки можно методом pop(), который удаляет значение по ключу и возвращает его:

```
value = d.pop(key, DEFAULT_VALUE)
```

Еще один популярный способ использования словаря состоит в том, что значение, соответствующее ключу, модифицируется, если ключ присутствует в словаре, иначе инициализируется начальным значением. Типичный пример подобного использования — построение частотного словаря, т. е. подсчет частоты встречаемости слов в заданном тексте:

```
d = {}
for word in text.split():
    if word in d:
        d[word] += 1
```

```
    else:
        d[word] = 0
```

Есть несколько способов записать этот код более компактно. Во-первых, можно воспользоваться методом setdefault(), который позволяет инициализировать значение словаря в случае его осутствия.

```
d = {}
for word in text.split():
    d.setdefault(word, 0)
    d[word] += 1
```

Во-вторых, в модуле collections стандартной библиотеки реализован контейнер аналогичный словарю, для которого можно установить значение по умолчанию.

```
from collections import defaltdict
d = defaultdict(int)
for word in text.split():
    d[word] += 1
```

Первый метод обычно используется при работе с уже существующими словарями, когда необходимо либо установить значение по умолчанию, либо изменить существующее. Кроме описанных выше встроенных методов словарей имеет смысл обратить внимание на методы update() и clear().

```
>>> d = {'a': 1, 'b': 2}
>>> d.update({'b': 4, 'c': 5} )
>>> d
{'a': 1, 'c': 5, 'b': 4}
>>> d.clear()
>>> d
{}
```

Еще одним типом неупорядоченных контейнеров являются множества. Множества в каком-то смысле соответствуют аналогичному математическому понятию и представляют собой последовательность уникальных элементов. Попытка добавления элемента, который уже входит в множество, просто игнорируется.

```
>>> s = {1, 2, 3}
>>> s.add(4)
>>> s.add(1)
>>> s
```

```
set([1, 2, 3, 4])
>>> type(s)
<type 'set'>
```

Элементами множества, аналогично ключам словаря, могут быть любые неизменяемые (хешируемые) объекты. Для множеств реализован набор классических теоретико-множественных операций:

```
>>> s1 = {1, 2, 3}
>>> s2 = {2, 3, 4}
>>> s1 | s2
set([1, 2, 3, 4])
>>> s1 & s2
set([2, 3])
>>> s1 - s2
set([1])
>>> s2 - s1
set([4])
>>> s1 ^ s2
set([1, 4])
>>> s1 <= s2
False
>>> s1 <= {1, 2, 3, 4}
True
>>> 1 in s1
True
```

Каждый из приведенных выше операторов имеет аналоги в виде встроенных функций, которые могут принимать в качестве параметров любые итерируемые последовательности, например,

```
>>> set('abc').intersection('bcd')
set(['c', 'b'])
```

Существует также встроенный тип данных frozenset, который фактически является неизменяемым множеством и может быть использован в качестве ключа в словаре и элемента других множеств.

Следует также упомянуть о встроенном типе bytearray, который представляет собой изменяемый аналог строкового типа str. Он позволяет хранить последовательности чисел в диапазоне от 0 до 255 и предоставляет как набор функций для обработки строк, так и функции для модификации содержимого контейнера: insert(), remove(), pop() и т. д. Данный тип удобно использовать для работы с байтовыми последовательностями, например, данными в бинарных форматах.

В стандартной библиотеке языка реализован ряд функций, значительно упрощающих работу с контейнерами. В частности, функции sum(), max() и min() позволяют вычислить сумму элементов, найти максимальный и минимальный элементы контейнера соответственно.

```
>>> l = [1, 5, 2, 7, 3]
>>> sum(l)
18
>>> max(l)
7
>>> min(l)
1
```

Функции max() и min() также могут принимать произвольное количество аргументов.

```
>>> max(1, 2, 3)
3
```

Эту особенность удобно использовать в тех случаях, когда необходимо найти минимальное или максимальное значение, получаемое в результате вычислений, выполняемых в цикле. Вместо

```
max_val = max_val if max_val > new_val else new_val
```

можно написать

```
max_val = max(max_val, new_val)
```

Также во многих случаях бывают полезными функции all() и any(). Первая из них возвращает True, когда все элементы входного контейнера истинны, а вторая, когда хотя бы один из них является таковым.

```
>>> all([x % 2 for x in [1, 3, 5]])
True
```

В данном случае проверяется все ли элементы входного списка являются нечетными.

```
>>> any([x % 2 for x in [2, 4, 6, 7]])
True
```

А здесь определяется наличие хотя бы одного нечетного элемента.

Также для объединения нескольких контейнеров в кортежи может быть использована функция zip():

```
>>> zip([1, 2, 3], [4, 5, 6, 7])
[(1, 4), (2, 5), (3, 6)]
```

Функции

Каждая функция в Python определяет контекст имен. Это значит, что переменные, объявленные в теле функции, существуют лишь в ее пределах, а при выходе из нее уничтожаются. Поэтому важным моментом при написании функций является вопрос, какую из переменных использует интерпретатор в случае, если в контексте функции и в глобальном контексте объявлены одинаковые имена.

Сразу после запуска интерпретатора становится доступным набор идентификаторов, который называется встроенным. Эти имена определены в модуле __builtin__ или builtins в Python 3. При запуске интерпретатора данный модуль импортируется как __builtins__ и его содержимое может быть просмотрено соответствующей командой.

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError',
'AttributeError', 'BaseException', 'BufferError',
'BytesWarning', 'DeprecationWarning', 'EOFError',
'Ellipsis', 'EnvironmentError', 'Exception', 'False',
'FloatingPointError', 'FutureWarning',
'GeneratorExit', 'IOError', 'ImportError',
'ImportWarning', 'IndentationError', 'IndexError',
'KeyError', 'KeyboardInterrupt', 'LookupError',
'MemoryError', 'NameError', 'None', 'NotImplemented',
'NotImplementedError', 'OSError', 'OverflowError',
'PendingDeprecationWarning', 'ReferenceError',
'RuntimeError', 'RuntimeWarning', 'StandardError',
'StopIteration', 'SyntaxError', 'SyntaxWarning',
'SystemError', 'SystemExit', 'TabError', 'True',
'TypeError', 'UnboundLocalError',
```

```
'UnicodeDecodeError', 'UnicodeEncodeError',
'UnicodeError', 'UnicodeTranslateError',
'UnicodeWarning', 'UserWarning', 'ValueError',
'Warning', 'ZeroDivisionError', '_', '__debug__',
'__doc__', '__import__', '__name__', '__package__',
'abs', 'all', 'any', 'apply', 'basestring', 'bin',
'bool', 'buffer', 'bytearray', 'bytes', 'callable',
'chr', 'classmethod', 'cmp', 'coerce', 'compile',
'complex', 'copyright', 'credits', 'delattr', 'dict',
'dir', 'divmod', 'enumerate', 'eval', 'execfile',
'exit', 'file', 'filter', 'float', 'format',
'frozenset', 'getattr', 'globals', 'hasattr', 'hash',
'help', 'hex', 'id', 'input', 'int', 'intern',
'isinstance', 'issubclass', 'iter', 'len', 'license',
'list', 'locals', 'long', 'map', 'max', 'memoryview',
'min', 'next', 'object', 'oct', 'open', 'ord', 'pow',
'print', 'property', 'quit', 'range', 'raw_input',
'reduce', 'reload', 'repr', 'reversed', 'round',
'set', 'setattr', 'slice', 'sorted', 'staticmethod',
'str', 'sum', 'super', 'tuple', 'type', 'unichr',
'unicode', 'vars', 'xrange', 'zip']
```

Любая переменная, объявленная за пределами функции, считается объявленной в глобальном контексте или просто глобальной. Ее имя помещается в глобальный словарь имен, который можно проанализировать при помощи функции globals().

```
>>> globals()
{'__builtins__': <module '__builtin__' (built-in)>,
'__name__': '__main__', '__doc__': None,
'__package__': None}
>>> a = 1
>>> globals()
{'__builtins__': <module '__builtin__' (built-in)>,
```

```
'__name__': '__main__', '__doc__': None, 'a': 1,
'__package__': None}
```

Из примера видно, что после определения переменной a, она появилась в глобальном словаре имен.

При определении переменной внутри функции ее имя попадает во внутренний словарь функции.

```
>>> def f():
...     a = 3
...     print locals()
...     print globals()
...
>>> f()
{'a': 3}
{'a':   1,   'f':   <function   f   at   0xb7455e9c>,
'__builtins__':  <module  '__builtin__'  (built-in)>,
'__package__':    None,    '__name__':    '__main__',
'__doc__': None}
```

Видно, что переменные с именем a в локальном и глобальном контексте имеют разные значения. Ситуация усугубляется тем, что функции могут быть вложены одна в другую. Для того, чтобы однозначно определить, какая из переменных будет использована при обращении в случае, если переменные с одинаковыми именами определены в различных контекстах, используется правило LEGB. Оно расшифровывается как Local-Enclosed-Global-Builtins и определяет порядок поиска имен. Это означает, что при обращении к имени интерпретатор попытается найти его определение в локальном контексте функции; при неудаче — продолжит искать последовательно в каждой из функций, в которые данная функция вложена, затем в глобальном контексте и, наконец, во встроенных именах.

```
>>> abs(-5)
5
>>> def abs(x):
...     print 'abs'
>>> def f():
...     def abs(x):
...             print 'Local abs'
...     abs(-5)
>>> f()
Local abs
```

```
>>> abs(-5)
abs
```

Первый вызов в данном примере приводит к обращению в словарь глобальных имен, и, поскольку там функция abs() отсутствует, поиск производится во встроенных именах. Далее выполняется создание глобальной функции abs() и функции f(), внутри которой также определяется функция abs(). При вызове функции f() поиск производится среди локальных имен, а при вызове abs() в глобальном контексте вызывается новоопределенная глобальная функция.

Следует отметить, что если переменная определена в функции, но до ее определения выполняется попытка обращения к глобальной переменной с тем же именем, интерпретатор выдаст ошибку.

```
>>> a = 1
>>> def f():
...     print a
...     a = 2
...
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
UnboundLocalError: local variable 'a' referenced
before assignment
```

Это происходит потому, что во время создания функции интерпретатором локальная переменная a заносится в словарь локальных имен. Соответственно, при попытке вывести значение a, по правилу LEGB, первой будет найдена переменная в локальном контексте, но она еще не определена. Также неудачей обернется попытка присвоить значение глобальной переменной в локальном контексте.

```
>>> a = 1
>>> def f():
...     a = 2
>>> f()
>>> a
1
```

Из примера видно, что значение глобальной переменной не изменилось. Очевидно, что оператор присваивания не находит в локальном контексте переменную a и создает ее, а не использует

глобальное значение. Для того, чтобы получить возможность изменения значений глобальных переменных в функциях, необходимо использовать оператор global.

```
>>> a = 1
>>> def f():
...     global a
...     a = 2
>>> f()
>>> a
2
```

Данный оператор указывает, что переменную, к которой производится обращение, необходимо взять из глобального контекста. В Python 3 добавлен оператор nonlocal, который позволяет указать, что переменную во вложенной функции необходимо искать за ее пределами, но не в глобальном контексте.

Стоит иметь в виду, что изменение глобальных переменных внутри функций является небезопасным и может приводить к неожиданным ошибкам.

```
>>> counter = 0
>>> def is_positive(x):
...     global counter
...     counter += 1
...     return x > 0
...
>>> def test(flag, x):
...     if flag and is_positive(x):
...             print 'Ok'
...
>>> test(True, 3)
Ok
>>> test(False, 3)
>>> counter
1
```

В данном примере функция is_positive() имеет побочный эффект в виде изменения значения глобальной переменной counter. Однако, поскольку логические выражения в Python вычисляются по сокращенному алгоритму, а именно, если первый аргумент and ложный, то второй аргумент не вычисляется и все выражение считается ложным, то во втором случае is_positive() не вызывается вообще и, соответственно, counter не увеличивается на 1. Подобные

конструкции, особенно при большой степени вложенности, часто приводят к ошибкам и очень сложны в отладке, поэтому global желательно избегать.

Т. к. каждая функция имеет свой локальный контекст, в Python возможны рекурсивные вызовы, т. е. функция может вызывать сама себя. Ниже приведено несколько примеров использования рекурсии.

Функция рекурсивного вычисления значения факториала:

```
def fact(n):
    if n < 1:
        return 1
    else:
        return n * fact(n-1)
```

Или более короткий вариант:

```
def fact(n):
    return 1 if n < 1 else n * fact(n - 1)
```

Рекурсивное вычисление суммы элементов списка:

```
def rec_sum(l):
    if l:
        return rec_sum(l[1:]) + l[0]
    else:
        return 0
```

Вычисление суммы элементов кортежа с произвольным количеством вложенности:

```
>>> tree
(1, 2, (3, (4, 5), 3), (4, 2))
>>> def sum_tree(tree):
...     try:
...         s = 0
...         for el in tree:
...             s += sum_tree(el)
...         return s
...     except TypeError:
...         return tree
>>> sum_tree(tree)
24
```

Здесь для проверки, является ли текущий элемент кортежем, используется оператор try. Такой способ проверки характерен для

языков программирования с «утиной» типизацией и является отражением принципа EAFP (Easier to Ask Forgiveness than Permission) — «проще попросить прощения, чем разрешения».

Следует иметь в виду, что каждый рекурсивный вызов создает локальный контекст и, поэтому, расходует оперативную память. Это значит, что если глубина вложенности рекурсивных вызовов превышает определенный предел, интерпретатор прервет выполнение программы. При этом вывод интерпретатора в консоли будет выглядеть приблизительно так:

```
>>> def rec_loop():
...     rec_loop()
>>> rec_loop()
...
  File "<stdin>", line 2, in rec_loop
  File "<stdin>", line 2, in rec_loop
  File "<stdin>", line 2, in rec_loop
RuntimeError: maximum recursion depth exceeded
```

Максимальная воженность рекурсивных вызовов хранится во внутренней переменной языка и может быть получена и изменена с помощью специальных вызовов.

```
>>> import sys
>>> sys.getrecursionlimit()
1000
>>> counter = 0
>>> def rec_loop():
...     global counter
...     counter += 1
...     rec_loop()
>>> sys.setrecursionlimit(10)
>>> try:
...     rec_loop()
... except RuntimeError:
...     print "Stop on %d" % counter
Stop on 9
```

Для параметров функции, которые в преобладающем количестве ситуаций имеют одно и то же значение, могут быть заданы значения по умолчанию. Если входное значение параметра не отличается от значения по умолчанию, его можно не задавать.

```
>>> def mul(a, b=2):
...     return a * b
```

```
>>> mul(2)
4
>>> mul(3)
6
>>> mul(3, 3)
9
```

Если параметров по умолчанию несколько и есть необходимость задать только некоторые из них или, из соображений читабельности, возникает желание указать имя передаваемого параметра, можно воспользоваться именованными параметрами.

```
>>> def sum(a, b, c, d=0, e=0):
...     return a + b + c + d + e
...
>>> sum(1, 2, 3)
6
>>> sum(1, 2, 3, e=5)
11
>>> sum(c=1, b=2, a=3, e=5)
11
>>> sum(e=5, 1, 2, 3)
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
```

В последнем примере возникла ошибка потому что в соответствии с синтаксисом языка позиционные параметры всегда должны быть установлены перед именованными, иначе порядок передачи параметров становится неоднозначным.

Значения по умолчанию для параметров функций вычисляются один раз при создании функции и сохраняются в свойствах функции. Эта особенность позволяет реализовать один из способов определения замыканий в Python. Рассмотрим пример, в котором будет создана функция, зависящая от глобальной переменной.

```
>>> a = 2
>>> def multiply(x):
...     return a * x
>>> multiply(2)
4
>>> a = 3
>>> multiply(2)
6
```

В данном случае функция ведет себя вполне ожидаемо и умножает значение параметра на значение глобальной переменной a. При изменении значения глобальной переменной изменяется и результат выполнения функции.

Перепишем функцию с использованием значений по умолчанию.

```
>>> a = 2
>>> def multiply(x, a=a):
...     return a * x
...
>>> multiply(2)
4
>>> a = 3
>>> multiply(2)
4
```

Теперь при создании функции значение глобальной переменной a сохраняется во внутренней структуре данных, связанной с функцией. Эта операция выполняется единственный раз при создании функции и в дальнейшем данной значение не изменяется. В результате функция оказывается больше не связна с глобальным контекстом и любые его изменения не влияют на возвращаемое значение. Таким образом, замыкания позволяют захватить из внешнего контекста значение и сохранить в функции для дальнейшего использования.

Стоит обратить внимание на поведение функции в случае, когда параметру по умолчанию присваивается изменяемый объект, например, список. В таком случае список создается один раз при создании функции и все последующие действия будут производится с этим экземпляром списка, т. е. список будет вести себя как статическая переменная.

```
>>> def f(x, l=[]):
...     l.append(x)
...     print l
...
>>> f(1)
[1]
>>> f(2)
[1, 2]
```

Обычно такое поведение нежелательно, поэтому присутствие в параметрах по умолчанию изменяемых объектов, таких как список или словарь, обычно является признаком ошибки в логике программы. Если все-таки при отсутствии входного параметра необходимо

создавать изменяемый объект по умолчанию, иногда используется следующий подход.

```
>>> def get_smth_or_list(x=None):
...     if x is None:
...         return []
...     return x
>>> get_smth_or_list()
[]
>>> get_smth_or_list(123)
123
>>> get_smth_or_list(None)
[]
```

Приведенная в примере функция get_smth_or_list() создает и возвращает список в случае, когда она вызвана без параметров. Если функция явно получает в качестве параметра None, как в последней строке примера, то результат будет также равен списку, а не None, как можно было бы ожидать. Для того, чтобы отличать явно переданный параметр от параметра по умолчанию, возможно использование сторожевого значения.

```
>>> sentinel = object()
>>> def get_smth_or_list(x=sentinel):
...     if x is sentinel:
...         return []
...     return x
>>> get_smth_or_list(123)
123
>>> get_smth_or_list(None)
>>> get_smth_or_list()
[]
```

Сторожевое значение sentinel инициализируется экземпляром класса object, который будет иметь уникальный идентификатор. В этом случае некорректное поведение функции возможно лишь при явной передаче в качестве параметра значения sentinel, что крайне маловероятно в логике обычной программы.

В Python есть возможность определения функций с произвольным количеством параметров как именованных, так и позиционных. При этом все дополнительные позиционные значения упаковываются в кортеж, а именованные — в словарь.

```
>>> def f(a, b, *args, **kwargs):
...     print "a = %d" % a
```

```
...     print "b = %d" % b
...     for arg in args:
...             print arg
...     for key, value in kwargs.items():
...             print "%s = %d" % (key, value)
...
>>> f(1, 2, 3, 4, 5, c = 6, d = 7, e = 8)
a = 1
b = 2
3
4
5
c = 6
e = 8
d = 7
```

Функция из примера получает два обязательных параметра, а затем произвольное количество позиционных и именованных параметров. Идентификаторы args и kwargs в данном случае не относятся к ключевым словам, но являются общепринятыми. Функции со списками args и kwargs часто используются при перегрузке функций классов сторонних библиотек, когда необходимо выполнить некоторый набор дополнительных действий, а затем вызвать оригинальную библиотечную реализацию. Также данный механизм используется при создании функций оберток, которые не зависят от всего набора параметров оборачиваемой функции, например при описании декораторов. Примером использования списков параметров произвольной длины может быть следующая функция.

```
 >>> def sum_(*args):
...     s = 0
...     for arg in args:
...         s += arg
...     return s
>>> sum_(1, 2, 5)
8
>>> sum_(2, 3, 4, 5, 6, 7, 8)
35
>>> sum_()
0
```

Функция sum_() получает произвольное количество входных параметров и возвращает их сумму. Пример вызова функции без параметров возвращает не очевидный результат, т. к. сумма пустого

списка не должна быть равна 0. Для того, чтобы предотвратить такие вызовы функция может быть определена с одним обязательным параметром.

```
>>> def sum_(first, *args):
...     s = first
...     for arg in args:
...         s += arg
...     return s
>>> sum_()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sum_() missing 1 required positional argument: 'first'
>>> sum_(1)
1
>>> sum_(1, 2, 3)
6
```

Как видно из примера данная функция требует минимум один параметр для вызова.

Аналогично для передачи в функцию последовательности значений из контейнера, можно его распаковать прямо при вызове. Для этого, перед именем контейнера необходимо поместить символ '*'.

```
>>> l = [1, 2, 3]
>>> def f(a, b, c):
...     return a + b + c
...
>>> f(*l)
6
```

Аналогично, в качестве набора параметров могут быть распакованы словари.

```
>>> d = {'a': 1,'b': 2,'c': 3}
>>> f(*d)
'acb'
>>> f(**d)
6
```

В частности, эта особенность может быть использована для того, чтобы из списка списков получить списки первых элементов, вторых элементов и т. д., т. е. для транспонирования матриц.

```
>>> zip(*[[1, 2], [3, 4], [5, 6]])
[(1, 3, 5), (2, 4, 6)]
```

В данном случае, внешний список будет распакован в набор вложенных списков, для которых затем будет выполнена функция zip().

Важной особенностью функций в Python – то, что они являются объектами первого класса. Это означает, что функции можно присваивать переменным, передавать в качестве параметра в другие функции и возвращать из функций, т. е. делать с ними все то же, что можно делать с данными любых других типов языка.

```
>>> def double(x):
...     return 2 * x
>>> f = double
>>> f(2)
4
>>> def execute(f, x):
...     return f(x)
>>> execute(double, 2)
4
>>> def get_double():
...     def double(x):
...             return 2 * x
...     return double
>>> get_double()(2)
4
```

В приведенных выше примерах создается функция double() и присваивается переменной. Поскольку переменная содержит ссылку на функцию, ее можно вызывать по имени переменной. Далее создается функция execute(), первым параметром которой также является функция. Наконец, в третьем примере функция get_double() возвращает созданную внутри нее функцию double(). Соответственно, последний вызов примера приводит к возвращению функции, которая затем вызывается с параметром 2.

Возможность сохранять функции в переменные позволяет реализовать в Python многовариантное ветвление подобно оператору switch в операторах C++ или Java. Для этого функции, которые необходимо вызывать при определенных значениях переменной-переключателя, заносятся в словарь, а ключами словаря выбираются эти значения:

```
>>> def f():
```

```
...     print('f()')
>>> def g():
...     print('g()')
>>> d = {'key1': f, 'key2': g}
>>> s = 'key1'
>>> d[s]()
f()
```

В данном случае значение переменной s будет определять, какая функция будет вызвана. Данный пример можно дополнить реализацией вызова функции по умолчанию в случае, если значение переменной-переключателя не предусмотрено в словаре.

```
>>> def default():
...     print('default')
...
>>> d.get('key3', default)()
default
```

Здесь используется метод словаря get(), который позволяет получить значение по ключу, а в случае отсутствия ключа, вернуть значение по умолчанию, а именно функцию default() в приведенном примере.

В свою очередь, при создании функции внутри другой функции, если внутренняя функция зависит от параметров внешней функции, производится захват и сохранение значений этих параметров в локальном контексте. Таким образом происходит неявное замыкание.

```
>>> def f(x):
...     def g(y):
...         return x + y
...     return g
>>> a = 1
>>> new_g = f(a)
>>> new_g(1)
2
>>> a = 3
>>> new_g(1)
2
```

Как видно из примера, функция f() возвращает новую функцию, в которой неявно сохраняется значение входного параметра x. Это происходит каждый раз, когда вложенная функция зависит от локальных параметров внешней функции. Т. к. значения локальных

параметров исчезнут после завершения выполнения внешней функции, они сохраняются во внутренних структурах данных функции. Примером использования замыканий могут быть фабрики функций.

```
>>> def multiply(x):
...     def f(y):
...         return x * y
...     return f
>>> double = multiply(2)
>>> triple = multiply(3)
>>> double(2)
4
>>> triple(3)
9
```

Функция multiply() позволяет генерировать новые функции, которые умножают входной параметр на различные константы. При этом сама константа сохраняется при помощи неявного замыкания.

Описанные выше свойства функций позволяют создавать разного рода обобщенные алгоритмы. В частности, в Python реализованы встроенные функции map(), filter() и reduce(), которые в качестве первого параметра принимают функции. Функция map() вторым параметром получает контейнер, к каждому элементу которого применяет переданную функцию.

```
>>> def double(x):
...     return 2 * x
>>> map(double, [1, 2, 3, 4, 5])
[2, 4, 6, 8, 10]
```

map() применяет double() к каждому элементу переданного списка и возвращает список удвоенных значений. Первым параметром функции filter() является функция, которая также применяется ко всем значениям контейнера, переданного во втором параметре. Результат ее выполнения приводится к типу bool. Если он истинен, то элемент контейнера попадает в результирующий список, иначе он отбрасывается.

```
>>> def odd(x):
...     return x % 2
>>> filter(odd, [1, 2, 3, 4, 5])
[1, 3, 5]
```

Функция odd() возвращает остаток от деления входного параметра на 2, т. е. для четных чисел — 0, который приводится к False. Соответственно, все четные числа отбрасываются и в результате получается список нечетных чисел. Функция reduce() в первом параметре получает функцию от двух аргументов. Первым из них является аккумулятор, который накапливает результат выполнения заданной функции с самим собой и каждым следующим элементом контейнера, переданным во втором параметре функции reduce().

```
>>> def add(a, x):
...     return a + x
...
>>> reduce(add, [1, 2, 3, 4, 5])
15
>>> def mul(a, x):
...     return a * x
...
>>> reduce(mul, [1, 2, 3, 4, 5])
120
>>> reduce(add, [[1, 2, 3], [4, 5]])
[1, 2, 3, 4, 5]
>>> reduce(mul, [1, 2, 3, 4, 5], 0)
0
```

В последнем примере в функцию reduce() передан третий параметр, который задает начальное значение аккумулятора. В Python 3 функции map() и filter() реализованы как генераторы, т. е. они не создают весь результирующий список сразу, а возвращают каждое следующее значение по запросу. Функция reduce() в Python3 не является встроенной и находится в модуле functools.

Как видно из приведенных примеров, для использования описанных функций необходимо предварительно описать функции, которые в последствии выступят в качестве параметров. Это бывает неудобно в случае, когда подобные функции используются один раз для конкретной задачи. В таких ситуациях удобно создавать функции непосредственно в месте вызова при помощи оператора lambda.

```
>>> map(lambda x: 2 * x, range(10))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
>>> reduce(lambda a, x: a + x, range(10))
45
```

Оператор lambda фактически создает анонимную функцию, т. е. функцию без имени, которая уничтожается сразу после ее

выполнения. Анонимные функции не могут быть длиннее одной строки. Кроме того, анонимные функции могут быть использованы для создания функций с именем.

```
>>> f = lambda x: x ** 2
>>> f(3)
9
```

В PEP8 данный способ использовать не рекомендуется, поскольку в таком случае более очевидным является применение ключевого слова def.

В некоторых случаях функции передаются в другие функции для настройки их работы. Например, функцию sort(), определенную для списков, можно настроить так, чтобы она сортировала списки кортежей или словарей по заданному критерию. Вообще, функция sort() упорядочивает кортежи в соответствии с порядком элементов — сначала сортировка производится по 0-евым элементам, затем по 1-ым и т. д., например:

```
>>> l = [(5, 2, 1), (3, 1), (5, 1), (2, 3), (5, 2)]
>>> l.sort()
>>> l
[(2, 3), (3, 1), (5, 1), (5, 2), (5, 2, 1)]
```

Если же возникает необходимость отсортировать по значениям элементов с определенным индексом, можно воспользоваться именованным параметром key. Данный параметр принимает функцию, которая вызывается для каждого элемента сортируемого списка, и должна возвращать вес этого элемента, которые в последствии используется для сравнения элементов.

```
>>> l.sort(key=lambda x: x[1])
>>> l
[(3, 1), (5, 1), (5, 2), (5, 2, 1), (2, 3)]
```

lambda, определенная в этом примере, возвращает элементы с индексом 1, а затем на основе их значений выполняется сортировка. Аналогичного эффекта можно добиться используя функцию itemgetter() из модуля operator стандартной библиотеки. Эта функция возвращает элемент контейнера по индексу.

```
>>> from random import shuffle
>>> shuffle(l)
>>> l
```

```
[(2, 3), (5, 2), (5, 2, 1), (3, 1), (5, 1)]
>>> from operator import itemgetter
>>> l.sort(key=itemgetter(1))
>>> l
[(3, 1), (5, 1), (5, 2), (5, 2, 1), (2, 3)]
```

Чтобы убедиться, что сортировка действительно работает, предварительно было выполнено перемешивания элементов списка при помощи функции shuffle() из модуля random. Кроме параметра key можно передавать еще параметр cmp, который получает функцию сравнения с двумя аргументами. Эта функция должна возвращать -1, 0 или 1 в зависимости от того меньше ли, первый элемент второго, равен или больше. Данный способ сравнения считается устаревшим и, на сегодняшний день, рекомендуется использовать параметр key.

Также параметр key позволяет в некоторых случаях сортировать несравниваемые типы данных, что бывает актуально при использовании Python 3. В следующем примере список элементов разных типов упорядочивается в соответствии с их строковым представлением.

```
>>> l = ['abc', 1, 2]
>>> l.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: int() < str()
>>> l.sort(key=str)
>>> l
[1, 2, 'abc']
```

В некоторых случаях бывает удобным создавать новые функции на основе уже существующих. Для этого, как правило, используется частичное определение параметров и композиция функций. В модуле functools стандартной библиотеки описана функция partial, которая на основе заданной функции и набора параметров возвращает новую функцию (на самом деле объект типа partial) с меньшим числом параметров.

```
>>> from functools import partial
>>> def mul(n, x):
...     return n * x
>>> double = partial(mul, 2)
>>> double(2)
4
```

Приведенный пример может быть переписан короче с использованием модуля встроенной библиотеки operator. В этом модуле определены синонимы для всех операций языка.

```
>>> from operator import mul
>>> double = partial(mul, 2)
>>> double(2)
4
```

Композиция функций представляет собой новую функцию, которая последовательно вызывает переданные функции так, что результат предыдущей становится параметром следующей. К сожалению, в стандартной библиотеке нет функции для создания композиции, но ее несложно написать самому. Например, для композиции двух функций, каждая из которых принимает один параметр:

```
>>> def compose(f, g):
...     return lambda x: f(g(x))
```

Теперь, с помощью функции-композиции можно строить новые функции из уже существующих.

```
>>> quadruple = compose(double, double)
>>> quadruple(2)
8
>>> eight_times = compose(quadruple, double)
>>> eight_times(2)
16
```

С использованием partial() и compose() можно получать новые полезные функции, например из функции, проверяющей на нечетность можно получить новую, проверяющую на четность.

```
>>> odd = lambda x: x % 2
>>> from operator import add
>>> even = compose(partial(add, 1), odd)
>>> even(2)
1
```

В крупных программных проектах часто встречаются одинаковые фрагменты кода, которыми необходимо оборачивать большое количество функций. Такие фрагменты могут выполнять соединение с базой данных на входе в функцию и закрытие соединения на выходе,

логирование действий пользователя, проверку на наличие прав доступа к функции у данного пользователя и т. д. Чтобы такой код не дублировался многократно, его можно поместить функцию, которой затем оборачивать другие функции. Например, код функции-обертки, которая в моменты входа и выхода из функции выводит сообщения в консоль, может выглядеть следующим образом.

```
>>> def decor(f):
...     def wrapper():
...             print "in"
...             res = f()
...             print "out"
...             return res
...     return wrapper
>>> def f():
...     print "f"
...     return 0
>>> f = decor(f)
>>> f()
in
f
out
0
```

В данном примере функция decor() создает и возвращает функцию-обертку. Далее определяется функция f(), которая затем оборачивается. Для этого вызывается функция decor() с параметром f. В результате после вызова decor() переменная f будет равна новой функции, которая выводит необходимое сообщение, затем вызывает функцию f() и снова выводит сообщение. Получается, что функция f() сохраняет свое имя и предназначение, но при этом оборачивается в дополнительную функциональность. Поскольку данный способ достаточно популярен, в Python для него предусмотрена специальная конструкция, которая называется декоратором.

```
>>> @decor
... def f():
...     print "f"
...     return 0
...
>>> f()
in
f
out
0
```

Первая строка данного примера эквивалентна предпоследнему вызову из предыдущего примера.

В некоторых случаях необходимо иметь возможность дополнительной настройки декоратора при помощи параметров. Для этого необходимо создать функцию, которая вернет декоратор, которая, в свою очередь, вернет обертку. Например, данный декоратор позволяет умножить входной параметр на заданный коэффициент.

```
def scale(x):
...     def inner(f):
...             def wrapper(y):
...                     return f(x * y)
...             return wrapper
...     return inner
 >>> @scale(10)
... def f(x):
...     return x
>>> f(2)
20
```

В данном случае круглые скобки после имени функции scale требуют от интерпретатора ее вызова. Она возвращает функцию inner(), которая и является на самом деле декоратором. Подобный способ декорирования аналогичен следующему вызову:

```
f = scale(10)(f)
```

Файлы

Время жизни данных, которые обрабатывает интерпретатор ограничено временем работы самого интерпретатора. После завершения программы на языке Python, память, которая была использована для хранения данных, освобождается и, соответственно, значения всех переменных будут утеряны. Чтобы сохранить информацию для последующего использования, ее помещают в долговременную память, т. е. на диск. Единицей хранения данных на диске является файл. Для его идентификации используется имя, которое представляет собой строковую константу как правило в кодировке UTF-8.

Для получения доступа к конкретному файлу на диске в Python необходимо создать объект типа file, для чего служит встроенная функция open():

```
>>> f = open("file.txt", "w")
```

```
>>> f
<open file 'file.txt', mode 'w' at 0xb747f128>
```

Встроенная функция open() запрашивает у операционной системы доступ к файлу на диске и возвращает объект типа file. Через этот объект выполняются все дальнейшие операции с файлом. Первым параметром функции open() является имя открываемого файла, причем оно может включать в себя как относительный, так и абсолютный путь. Второй параметр определяет режим открытия файла и может принимать значения:

'w' — открытие файла для записи, данная операция фактически приводит к созданию нового файла с указанным именем;

'r' — открытие существующего файла для чтения;

'a' — открытие существующего файла для добавления;

'x' — открытие файла для записи, если файл уже существует, выдается сообщение об ошибке;

'r+', 'w+', 'a+' — открытие файла для редактирования, т. е. для чтения и записи одновременно.

Традиционно файл считается ресурсом с последовательным доступом. Это означает, что при его открытии создается внутренний указатель, который отмечает последний считанный или записанный байт. Каждая операция чтения или записи смещает указатель на определенное количество байтов и следующая операция выполняется начиная с новой позиции указателя. Таким образом, данные обрабатываются строго последовательно.

Данный подход был удобен в старых вычислительных системах, в которых в качестве носителей информации использовались накопители на магнитных лентах. Обращение к произвольной позиции в файле требовало перемотки ленты, что занимало достаточно много времени. Несмотря на то, что современные жесткие диски и другие носители информации позволяют находить необходимую позицию достаточно быстро, большинство функций для работы с файлами рассматривают их как последовательный ресурс.

Открытие файла для чтения и записи приводит к перемещению файлового указателя на начало файла, а открытие для добавления — в конец. Запись в файл производится с помощью метода write():

```
>>> f.tell()
0L
>>> f.write("String1\n")
>>> f.tell()
8L
>>> f.write("String2\n")
>>> f.write("String3\n")
```

```
>>> f.tell()
24L
>>> f.close()
```

Сразу после открытия файла для записи файловый указатель установлен на нулевой байт о чем свидетельствует результат, возвращаемый функцией tell(). В данном примере в ранее открытый файл будут последовательно записаны 3 строки. После каждой операции записи файловый указатель будет установлен на следующий байт за последним записанным. В результате, следующая операция записи поместит данные сразу за предыдущими. Поскольку в примере длина одной записываемой строки равна 8 символам, три последовательных записи поместили файловый указатель на 24-й байт начиная с нуля. После этого содержимое файла будет иметь следующий вид:

```
String1
String2
String3
```

Функция close() закрывает файл, тем самым освобождая ресурсы, которые были с ним связаны и сообщая операционной системе, что файл больше не используется.

Если открыть данный файл для добавления, файловый указатель окажется в конце файла и запись будет продолжаться с этого места.

```
>>> f = open("file.txt", "a")
>>> f.tell()
24L
```

Следует обратить внимание, что если просмотреть содержимое файла до его закрытия, последние записанные строки могут отсутствовать. Это происходит потому, что операционная система, для того, чтобы не заставлять пользователя ожидать завершения длительных дисковых операций, сохраняет данные в буфер, расположенный в оперативной памяти. Соответственно операция записи на диск выполняется в случае заполнения буфера или закрытия файла. Чтобы выполнить принудительную запись данных из буфера закрытия файла можно выполнить функцию flush():

```
>>> f.flush()
```

Данная функция бывает полезна, например, при логировании ошибок и в любых других случаях, когда необходимо обеспечить

доступность данных другим программам в реальном времени. Кроме этого, буферизацию можно отключить вообще, установив третий параметр функции open() в 0:

```
>>> f = open("file.txt", 'w', 0)
```

Чтение файла может быть выполнено побайтово или построчно.

```
>>> f = open("file.txt", "r")
>>> s = f.read()
>>> s
'String1\nString2\nString3\n'
>>> f.tell()
24L
>>> s = f.read()
>>> s
''
>>> f.close()
```

В данном случае функция read() прочитала все содержимое файла в строку, что видно по состоянию файлового указателя. Последующие вызовы read() для этого файла будут возвращать пустую строку, т. к. файловый указатель уже достиг конца файла.

```
>>> f = open("file.txt", "r")
>>> s = f.read(5)
>>> s
'Strin'
>>> f.tell()
5L
>>> s = f.read(5)
>>> s
'g1\nSt'
>>> f.tell()
10L
>>> f.close()
```

Здесь чтение выполняется по 5 байт за один вызов функции read(). Для того, чтобы прочитать из файла сразу целую строку можно воспользоваться функцией readline():

```
>>> f = open("file.txt", "r")
>>> s = f.readline()
>>> s
'String1\n'
```

```
>>> s = f.readline()
>>> s
'String2\n'
>>> f.close()
```

Также можно считать файл построчно при помощи цикла for:

```
>>> f = open("file.txt", "r")
>>> for line in f:
...     print line
...
String1

String2

String3

>>> f.close()
```

Пустые строки после каждой строки связаны с тем, что дополнительный символ перевода каретки добавляет оператор print.

Несмотря на то, что в большинстве случаев файлы обрабатываются последовательно, современные операционные системы предоставляют также возможность произвольного доступа к данным на диске. Подобный способ обычно используется при работе с файлами очень больших размеров, чтение в память полностью для которых нецелесообразно. При этом файл, как правило, открывается в режиме для чтения и записи — «r+», «a+» или «w+». Перемещение файлового указателя в этом случае выполняется с помощью функции seek().

```
>>> f = open("file.txt", "r+")
>>> f.tell()
0L
>>> f.seek(5)
>>> f.tell()
5L
>>> f.seek(-5, 2)
>>> f.tell()
19L
>>> f.seek(1, 1)
>>> f.tell()
20L
```

Как видно из примера, первый параметр функции seek()

определяет насколько байтов сместится указатель, а второй — позицию, относительно которой будет выполнено смещение. Данный параметр может принимать значения 0 (по умолчанию) — от начала файла; 1 — относительно текущего указателя; 2 — от конца.

Если при обращении к файлу происходит ошибка, интерпретатор сообщает об этом путем генерации исключения IOError.

```
>>> f = open("unknown.txt", "r")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError:  [Errno  2]  No  such  file  or  directory:
'unknown.txt'
```

В данном примере произведена попытка открытия несуществующего файла для чтения, что привело к выбросу исключения. Поскольку корректное выполнение файловых операций зависит от внешних факторов, любое обращение к файлам желательно оборачивать в оператор try/except.

Вообще говоря, при завершении процесса интерпретатора все файлы закрываются автоматически, поэтому при работе в интерактивном режиме вызывать функцию close() не обязательно. В остальных случаях закрытие файлов является необходимым, поскольку накопление неиспользуемых и незакрытых файлов может привести к нехватке ресурсов и краху программы. В крупных проектах, когда работа с файлами сопряжена со сложной программной логикой, а модификация кода производится через продолжительные временные промежутки, существует высокая вероятность появления веток программы, в которых по ошибке закрытие файла опущено. Чтобы избежать подобных проблем желательно позаботиться о закрытии файла во время его открытия. Для решения этой задачи в Python предусмотрен оператор with. Этот оператор позволяет получить доступ к ресурсу и при помощи операторного блока указать диапазон его использования. Сразу после выхода из блока ресурс будет освобожден.

```
>>> f = None
>>> def read_int(filename):
...     global f
...     with open(filename, 'r') as f:
...             try:
...                     i = int(f.readline())
...             except ValueError:
...                     return 0
...             return i
```

```
...
>>> read_int('file.txt')
0
>>> f
<closed file 'file.txt', mode 'r' at 0xb7412e90>
```

В данном примере функция read_int() выполняет попытку чтения из файла целого числа. Для того, чтобы получить информацию о состоянии файловой переменной за пределами функции, она объявлена глобальной. Функция имеет несколько операторов return, поэтому в случае отсутствия оператора with, перед каждым из них необходимо было бы вызвать close(), что само по себе является дублированием и потенциально приводит к ошибкам. Оператор with выполняет функцию open(), помещает результат в переменную f и гарантирует, что когда интерпретатор покинет блок, определенный данным оператором, обязательно выполнится закрытие файла.

Для работы с объектами файловой системы, такими как файлы и директории, в Python предусмотрен ряд дополнительных функций, определенных в модуле os стандартной библиотеки. Следующий пример демонстрирует использование ряда таких функций.

```
>>> import os
>>> os.mkdir('dir1')
```

Создана директория dir1.

```
>>> os.chdir('dir1')
```

Выполнен переход в эту директорию.

```
>>> os.getcwd()
'/home/z/dir1'
```

Отображение абсолютного пути к текущей директории.

```
>>> os.listdir('.')
[]
```

Просмотр содержимого директории, в данном случае текущей.

```
>>> f = open('file1', 'w')
>>> f.close()
>>> f = open('file2', 'w')
>>> f.close()
```

```
>>> os.listdir('.')
['file2', 'file1']
```

Создано два файла file1 и file2

```
>>> os.rename('file1', 'file3')
>>> os.listdir('.')
['file2', 'file3']
```

Файл file1 переименован в файл file3.

```
>>> os.remove('file2')
>>> os.listdir('.')
['file3']
>>> os.unlink('file1')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 2] No such file or directory: 'file1'
>>> os.unlink('file3')
>>> os.listdir('.')
[]
```

Оба созданных файла удалены. Функции remove() и unlink() являются синонимами. При возникновении ошибок во время выполнения функций модуля os выбрасывается исключение OSError.

```
 >>> os.chdir('..')
>>> os.rmdir('dir1')
```

Созданная директория удалена.

Нередко возникает необходимость обработать не только содержимое заданной директории, но и всех директорий, вложенных в нее. Эту операцию можно выполнить при помощи функции os.walk(), которая при каждом вызове последовательно возвращает путь к найденной директории, список всех вложенных директорий и список всех содержащихся файлов.

```
>>> for root, dirs, files in os.walk('/usr/lib/gcc'):
...     spaces = ' ' * root.count('/')
...     print "{}Root: {}".format(spaces, root)
...     print "{}Dirs: {}".format(spaces, dirs)
...     print "{}Files: {}".format(spaces, files)
   Root: /usr/lib/gcc
   Dirs: ['i686-linux-gnu']
   Files: []
```

```
    Root: /usr/lib/gcc/i686-linux-gnu
    Dirs:  ['4.8.4',  '4.9',  '4.6',  '4.6.4',  '4.8',
'4.9.1']
    Files: []
     Root: /usr/lib/gcc/i686-linux-gnu/4.9
     Dirs: []
     Files: []
     Root: /usr/lib/gcc/i686-linux-gnu/4.6
     Dirs: ['include-fixed', 'include']
     …
```

В данном примере выполняется обход всех директорий вложенных в /usr/lib/gcc. Переменная spaces содержит столько пробелов, сколько символов '/' содержится в текущем обрабатываемом пути. Это позволяет при выводе сдвинуть каждый новый уровень вложенности на один пробел вправо.

Не менее полезным при работе с файлами является модуль os.path, предназначенный для работы с путями. В частности, для получения абсолютного пути используется следующая функция:

```
>>> import os.path
>>> os.path.abspath('.')
'/home/z'
```

Проверка существования пути.

```
>>> os.path.exists('/usr/lib/gcc/')
True
```

Определение размера файла.

```
>>> os.path.getsize('/etc/passwd')
2151L
```

Проверка, существует ли файл. Данная функция в данном контексте предпочтительнее exists(), т. к. гарантирует наличие файла, а не директории.

```
>>> os.path.isfile('/etc/passwd')
True
```

и директории

```
>>> os.path.isdir('/etc/passwd')
```

```
False
```

Генерация полного пути. Данная функция является кроссплатформенной, т. к. использует разделитель директорий, который применяется в данной операционной системе.

```
>>> os.path.join('/', 'usr', 'lib')
'/usr/lib'
```

Функции для выделения имени файла и пути к нему.

```
>>> os.path.dirname('/usr/bin/python')
'/usr/bin'
>>> os.path.split('/usr/bin/python')
('/usr/bin', 'python')
```

Иногда возникает необходимость создания временных файлов, например, для хранения промежуточных значений вычислений. Такие файлы могут быть созданы с помощью модуля tempfile стандартной библиотеки, что значительно упрощает их дальнейшую поддержку.

```
>>> import tempfile
>>> f = tempfile.NamedTemporaryFile()
>>> f
<open file '<fdopen>', mode 'w+b' at 0xb74c0b20>
>>> f.name
'/tmp/tmpJ4OXpg'
>>> f.close()
>>> os.path.isfile(f.name)
False
>>> f = tempfile.NamedTemporaryFile(delete=False)
>>> f.name
'/tmp/tmpPwxrlw'
>>> f.close()
>>> os.path.isfile(f.name)
True
>>> f = tempfile.TemporaryFile()
>>> f.name
'<fdopen>'
```

В данном примере последовательно создаются два именованных временных файла и один неименованный. Именованные файлы видимы как файловые объекты в файловой системе и другие процессы могут получать к ним доступ. Неименованный файл доступен только создавшему его процессу. Время жизни именованных временных

файлов ограничено моментом их закрытия, если при их создании не указан дополнительный параметр delete=True.

Работа с файлами предполагает возможность сохранения и восстановления данных произвольных типов, в том числе составных объектов, включающих, например, контейнеры или экземпляры классов. Для таких данных обычно несложно получить строковое представление, однако реализация обратной операции может оказаться достаточно непростой. Поэтому сохранение подобных объектов выполняется при помощи средств сериализации. В частности, в стандартной библиотеке существует два основных способа сериализации данных — модули pickle и json.

Модуль pickle является традиционным для Python, соответственно, он корректно работает со всеми объектами языка. Существует возможность сериализации как в строку, так и непосредственно в файл.

```
>>> l = {'a': [1, 2, 3], True: ('abc', (0, 2))}
>>> import pickle
>>> pickle.dumps(l)
"(dp0\nS'a'\np1\n(lp2\nI1\naI2\naI3\nasI01\n(S'abc'\np
3\n(I0\nI2\ntp4\ntp5\ns."
>>> pickle.loads(pickle.dumps(l)) == l
True
>>> pickle.dump(l, open('l.bin', 'w'))
>>> del l
>>> l
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'l' is not defined
>>> l = pickle.load(open('l.bin', 'r'))
>>> l
{'a': [1, 2, 3], True: ('abc', (0, 2))}
```

В данном примере объект, имеющий сложную структуру был сериализован сначала в строку, а затем в файл. После этого исходный объект был удален и восстановлен из файла. Видно, что исходный и восстановленный объекты совпадают. Следует обратить внимание, что в Python 2 сериализация средствами pickle выполняется в текстовом формате, а в Python 3 — в бинарном, соответственно в каждом случае необходимо выбирать правильный режим открытия файла.

Pickle применим в том случае, когда и сериализация и десериализация будут выполняться при помощи средств языка Python. В случае, когда данные необходимо передавать между приложениями,

написанными на разных языках программирования, удобно применять язык JSON — JavaScript Object Notation (см. http://json.org).

```
>>> import json
>>> json.dumps(l)
'{"a": [1, 2, 3], "True": ["abc", [0, 2]]}'
>>> json.loads(json.dumps(l))
{u'a': [1, 2, 3], u'True': [u'abc', [0, 2]]}
>>> json.dump(l, open('l.json', 'w'))
>>> del l
>>> l = json.load(open('l.json', 'r'))
>>> l
{u'a': [1, 2, 3], u'true': [u'abc', [0, 2]]}
```

Как видно из примера, интерфейс модуля json полностью идентичен интерфейсу pickle. В Python 2 особенностью json является конвертация всех строковых объектов в unicode.

Кроме сериализации, стандартная библиотека языка включает в себя поддержку ряда других форматов файлов. В частности, модуль zipfile позволяет архивировать и разархивировать файлы с использованием формата zip.

```
>>> os.path.getsize('longfile.txt')
594933L
>>>  with  zipfile.ZipFile('longfile.zip',  'w')  as
myzip:
…        myzip.write('pg1661.txt',
             compress_type=zipfile.ZIP_DEFLATED)
>>> os.path.getsize('longfile.zip')
227008L
>>> zipfile.ZipFile('longfile.zip').extract(
 'longfile.txt',
 path='tmp'
)
'tmp/longfile.txt'
>>> import filecmp
>>> filecmp.cmp('longfile.txt', 'tmp/longfile.txt')
True
```

Из примера видно, что объем архивированного файла уменьшился более чем в 2 раза. При этом функция cmp модуля filecmp, которая позволяет сравнивать два файловых объекта показала, что входной и разархивированный файлы идентичны.

Классы

Как говорилось ранее, любые данные, обрабатываемые интерпретатором языка Python, являются объектами или, еще говорят, экземплярами классов. Класс по своей сути является типом, причем могут существовать как встроенные классы, так и классы, созданные пользователем. Тип, как указывалось выше, хранит в себе информацию о данных, которые содержатся в объектах типа и об операциях, которые можно к значениям данного типа применять.

Python позволяет определять пользовательские типы, которые принято называть классами, что позволяет использовать объектно-ориентированную парадигму программирования. Следует отметить, что несмотря на то, что на Python можно писать код в стиле традиционных объектно-ориентированных языков программирования, таких как, например, Java или C++, внутреннее представление классов и объектов в Python существенно отличается.

В первую очередь отличие состоит в том, что в компилируемых языках программирования классы представляют собой сущности времени компиляции и в выполняемом коде представляются, как правило, исключительно экземплярами. В Python классы являются такими же объектами, как и любые другие объекты языка и существуют во время выполнения программы, т. е. они могут быть помещены в переменную, переданы в качестве параметра в функцию, созданы и изменены динамически. В отличие от традиционных языков, классы не определяют напрямую свойства и методы экземпляров. Единственной связью между классом и экземпляром класса является переменная экземпляра __class__, которая определяет порядок поиска методов и свойств: сначала интерпретатор ищет атрибут в экземпляре, а затем, если его там нет, поиск продолжается в классе, на который указывает переменная __class__.

Класс в Python создается при помощи ключевого слова class и может включать в себя набор свойств и методов. Следует отметить, что свойства класса относятся непосредственно к классу и не копируются в экземпляр.

```
>>> class A:
...     a = 1
...     b = 2
>>> A
<class '__main__.A'>
```

В данном случае, создан класс с именем A, который содержит две переменные класса a и b. Доступ к этим переменным можно получить

через имя класса, используя символ «.»:

```
>>> A.a
1
```

При помощи класса может быть создан его экземпляр, связанный с породившим его классом, как уже было сказано выше, переменной __class__. Создание экземпляра синтаксически выглядит как выполнение класса как функции.

```
>>> a = A()
>>> a
<__main__.A object at 0xb70b888c>
>>> a.__class__
<class '__main__.A'>
```

Аналогично можно создавать и экземпляры встроенных типов, например:

```
>>> i = int(1)
>>> i
1
>>> s = str(23)
>>> s
'23'
```

В данном примере создаются два объекта встроенных типов int и str. На самом деле, в случае со значением 1, новый объект скорее всего создан не будет, а будет использовано уже существующее значение в памяти, но это работает только с некоторыми объектами неизменяемых встроенных типов.

Полученный выше экземпляр класса a не содержит собственных свойств, в чем можно убедиться путем вызова функции vars(), которая предназначена для вывода на экран атрибутов объекта.

```
>>> vars(a)
{}
```

Как видно из примера, атрибуты экземпляра класса хранятся в словаре. Этот словарь называется __dict__ и к нему можно получить доступ непосредственно.

```
>>> a.__dict__
{}
```

Однако при попытке получить доступ к атрибуту a экземпляра класса a интерпретатор вместо ошибки выведет значение.

```
>>> a.a
1
```

Т. к. словарь атрибутов экземпляра пуст, можно сделать вывод, что данное значение принадлежит классу, а не экземпляру. На самом деле, при обращении к отсутствующему полю или методу экземпляра класса, интерпретатор пытается найти его в классе, используя переменную __class__. Подобный поиск производится каждый раз, когда происходит обращение к атрибутам класса, в чем можно убедиться следующим образом.

```
>>> a = A()
>>> a.a
1
>>> class A:
...     a = 5
>>> a = A()
>>> a.a
5
>>> class B:
...     a = 3
>>> a.__class__ = B
>>> a.a
3
```

Как видно из приведенного примера, экземпляр класса A во время выполнения программы может превратиться в экземпляр класса B.

Кроме атрибутов класса, экземпляр может содержать собственные атрибуты. Добавить атрибут классу можно при помощи синтаксической конструкции, содержащей символ «.».

```
>>> class A:
...     a = 1
>>> a = A()
>>> a.a = 2
>>> a.a
2
>>> A.a
1
```

Теперь и класс и экземпляр содержат атрибуты с именами a, но с разными значениями. Новый экземпляр класса A не будет иметь

собственного атрибута a, поэтому, при попытке обращения к нему, будет возвращать значение класса.

```
>>> b = A()
>>> b.a
1
```

Как было указано выше, атрибуты экземпляра размещаются в словаре __dict__, который, как и любой другой словарь, является изменяемым, что позволяет во время выполнения программы добавлять и удалять атрибуты.

```
>>> a.x = 1
>>> a.x
1
>>> del a.x
>>> vars(a)
{'a': 2}
```

В процессе выполнения программы количество и типы переменных экземпляра класса могут значительно изменится, и отличаться о тех, которые определены в классе, который породил экземпляр. Подобная гибкость экземпляров известна как «утиная» (duck) типизация, которая является одной их характерных особенностей языка Python. Термин «утиная» типизация происходит от известной английской фразы «если птица плавает как утка, ходит как утка и крякает как утка, то она, вероятно, является уткой». В данном случае имеется в виду, что тип объекта определяется его поведением и свойствами, а не принадлежностью к тому или иному классу.

Класс, кроме собственных атрибутов-переменных также может содержать атрибуты-функции, которые принято называть методами.

```
>>> class A:
...     def f(self):
...             print(self)
...             self.x = 1
>>> a = A()
>>> a.f()
<__main__.A object at 0xb701f90c>
>>> a
<__main__.A object at 0xb701f90c>
>>> a.x
1
```

Методы всегда получают обязательный первый параметр, который принято называть self. Несмотря на то, что для первого параметра можно использовать произвольный допустимый идентификатор, этого делать не рекомендуется, чтобы не вводить других разработчиков в заблуждение. Данный параметр содержит указатель на экземпляр класса, от имени которого вызывается метод. Как видно из примера, значения self и переменной a совпадают. Соответственно, оператор присваивания a.x = 1 эквивалентен оператору self.x = 1 внутри класса, т. е. методы класса могут использовать атрибуты экземпляра.

Если рассматривать метод как атрибут класса, в котором он определен, то он будет представлять собой обыкновенную функцию, которой можно передать произвольный параметр, в том числе экземпляр другого класса.

```
>>> class A:
...     def f(self):
...             self.x = 1
...
>>> A.f
<function A.f at 0xb7053adc>
>>> class B:
...     pass
...
>>> b = B()
>>> A.f(b)
>>> b.x
1
```

Но в рамках экземпляра класса, благодаря специальным методам, которые реализуют протокол дескрипторов, рассматриваемый ниже, такая функция превращается в метод.

```
>>> a = A()
>>> a.f
<bound method A.f of <__main__.A object at
0xb7068b0c>>
>>> a.f()
>>> a.x
1
```

Метод привязан к экземпляру при помощи параметра self, неявно передаваемого при вызове.

Т. к. классы, как правило, являются моделями определенных

сущностей из предметной области, для которой выполняется разработка, все экземпляры определенного класса должны иметь определенный набор свойств, чтобы описывать эти сущности. Поскольку в Python сам класс ничего не дает своему экземпляру, кроме ссылки на самого себя, задачу инициализации переменных берет на себя специальный метод __init__(). Этот метод обязательно вызывается каждый раз при создании экземпляра класса.

```
>>> class A:
...     def __init__(self, a):
...             print "Init was called"
...             self.a = a
>>> a = A(1)
Init was called
>>> a.a
1
>>> b = A()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() takes exactly 2 arguments (1
given)
```

Из данного примера видно, что функция __init__() была вызвана неявно при создании экземпляра, причем параметр, переданный при вызове A, стал параметром __init__(). При создании второго экземпляра возникла ошибка потому что для функции __init__() было передано недостаточно параметров. Таким образом гарантируется, что созданный экземпляр будет иметь все необходимые для дальнейшей работы данные. Вообще, методы классов, имена которых обернуты с обеих сторон 2-мя символами подчеркивания, как правило, имеют специальное значение в языке и вызываются интерпретатором в особых случаях, поэтому их принято называть «магическими» (magic methods).

Таким образом, метод __init__() является местом, в котором в большинстве случаев экземпляр класса получает свои атрибуты. В предыдущих примерах классы не имели метода __init__(), но при этом их экземпляры все равно можно было создать. Это объясняется наличием в объектной модели языка Python механизма наследования. На самом деле, любой пользовательский класс обязательно имеет один или несколько классов-родителей, от которых он отнаследован. Традиционно, наследование в программировании понимают как статическое средство, позволяющее классу-наследнику получить все атрибуты и методы от родителя. В Python наследование выглядит

похожим образом, но реализовано динамически. Как и в случае с экземпляром, который содержит ссылку на породивший его класс, класс-наследник содержит ссылки на своих родителей. При обращении к методу или свойству экземпляра, Python сначала пытается найти его в словаре экземпляра, затем в словаре класса, а затем последовательно обходит все родительские классы, пока не найдет искомый атрибут.

В вершине иерархии классов Python находится класс object и в версии Python 3 он автоматически становится родителем любого класса, если у него нет других родителей. В случае создания экземпляра класса, который не имеет __init__(), используется __init__() из класса-родителя, в частности, в предыдущих примерах из класса object.

При определении пользовательского класса список наследования указывается после имени класса.

```
>>> class A:
...     a = 1
>>> class B(A):
...     pass
>>> a = B()
>>> a.a
1
```

Здесь был создан класс B, унаследованный от класса A. Ни экземпляр a, ни класс B не содержат атрибута a, поэтому данный атрибут был найден в классе A. Наследование позволяет существенно сократить объем дублирования кода, позволяя повторно использовать код классов-родителей.

Следует отметить, что в Python 2 существует два вида классов — старого и нового стиля. Классы старого стиля использовались до версии 2.2 и были оставлены в языке для обратной совместимости. Отличием классов нового стиля, кроме расширенной функциональности, является явное наследование от класса object.

Поиск атрибутов при наследовании, так же как и при инстанциировании, выполняется во время выполнения. Аналогично атрибуту __class__ экземпляра класса, у каждого класса есть атрибут __bases__, который определяет кортеж прямых родителей. При его изменении изменяется поведение и класса-наследника.

```
>>> class A:
...     def f(self):
...             print('A')
>>> class B:
```

```
...     def f(self):
...             print('B')
>>> class C(A):
...     pass
>>> a = C()
>>> a.f()
A
>>> C.__bases__
(<class '__main__.A'>,)
>>> C.__bases__ = (B, )
>>> a.f()
B
```

В Python разрешено использование множественного наследования, благодаря значительно упрощается повторное использование кода.

```
>>> class Terrestrial:
...     def run(self):
...             print("%s is running" % self.name)
>>> class Natatorial:
...     def swim(self):
...             print("%s is swimming" % self.name)
>>> class Creature(Terrestrial, Natatorial):
...     def __init__(self, name):
...             self.name = name
>>> Bill = Creature('Bill')
>>> Bill.run(); Bill.swim(); Bill.run()
Bill is running
Bill is swimming
Bill is running
```

В данном примере класс-наследник получает все возможности, которые доступны в базовых классах. Однако, если в нескольких базовых классах определены методы с одинаковыми именами, который из них будет вызван определяется порядком следования классов в списке наследования.

```
>>> class A:
...     def f(self):
...             print('A')
>>> class B:
...     def f(self):
...             print('B')
>>> class AB(A, B):
```

```
...     pass
>>> ab = AB()
>>> class BA(B, A):
...     pass
>>> ba = BA()
>>> ab.f()
A
>>> ba.f()
B
```

Когда иерархия наследования глубокая и разветвленная, порядок поиска методов (method resolution order, MRO) в классах-наследниках далеко не всегда очевиден. Для его однозначного определения используется алгоритм линеаризации C3. Данный алгоритм строит такую последовательность классов, в которой: порядок классов всегда соответствует порядку следования в списке наследования (слева направо); порядок классов у класса-наследника всегда будет совпадать с порядком у его классов-родителей.

Неформально работу данного алгоритма можно описать так: классы в последовательность поиска добавляются по возможности снизу вверх, слева направо, но так, чтобы класс-родитель не мог идти за классом-наследником. Например, рассмотрим следующую иерархию.

```
class A: pass
class B(A): pass
class C(A): pass
class D(A): pass
class E(B, C): pass
class F(D): pass
class G(E, F): pass
```

Для класса G порядок поиска методов будет следующий. Двигаясь снизу вверх и слева направо мы сначала попадаем в класс E, а затем в B. Дальше вверх двигаться нельзя, т. к. класс A является родителем для других родителей класса G, которые мы не рассмотрели. Поэтому следующим будет класс C. Из него мы снова попадаем в A, но поскольку туда идти не можем, сначала рассматриваем F и D, а только затем A.

Получить информацию об MRO для данного класса можно путем вызова метода mro().

```
>>> G.mro()
[<class '__main__.G'>, <class '__main__.E'>, <class
```

```
'__main__.B'>,     <class     '__main__.C'>,     <class
'__main__.F'>,     <class     '__main__.D'>,     <class
'__main__.A'>, <class 'object'>]
```

В некоторых случаях, когда для заданной иерархии наследования невозможно определить однозначный порядок вызова методов, создание класса может привести к ошибке.

```
>>> class A:
...     def f(self):
...             print 'A'
>>> class B(A):
...     def f(self):
...             print 'B'
>>> class C(A, B):
...     pass
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Error when calling the metaclass bases
    Cannot create a consistent method resolution
order (MRO) for bases B, A
```

В то же время, если поменять местами классы A и B в списке наследования, класс будет создан без ошибок.

При использовании наследования в классах-потомках часто используется переопределение методов. В этом случае бывает необходимо расширить существующее в базовом классе поведение, а не полностью его переопределить.

Допустим, для описания произвольной фигуры на плоскости определен класс Shape. Атрибуты x и y представляют центр фигуры.

```
>>> class Shape:
...     def __init__(self, x, y):
...             self.x = x
...             self.y = y
```

Для того, чтобы определить класс для описания конкретного типа фигуры, например, круга, можно отнаследовать класс круга от класса фигуры. При этом класс круга Circle получит доступ к методу __init__() класса Shape, который определяет в экземпляре атрибуты для представления координат. Однако класс круга требует дополнительного параметра для хранения радиуса, который необходимо также инициализировать в методе __init__(), что

вынуждает переопределить его в классе Circle. Чтобы не дублировать код определения координат __init__() класса Circle может вызывать __init__() из Shape.

```
>>> class Circle(Shape):
...     def __init__(self, x, y, r):
...             Shape.__init__(self, x, y)
...             self.r = r
>>> c = Circle(1, 2, 3)
>>> vars(c)
{'y': 2, 'x': 1, 'r': 3}
```

Данный код работает, но имеет один недостаток. Имя класса-родителя дублируется в списке наследования и при вызове __init__() родительского класса. В случае необходимости внести изменения в иерархию классов код придется редактировать в двух местах, что может привести к последующим ошибкам. Следует также обратить внимание, что в данном случае метод __init__() вызывается как функция от имени класса и, поэтому, требует явного указания self.

Для того, чтобы избежать дублирования для вызова методов родительского класса используют метод super(). При обращении к методу класса через super() метод возвращается уже связанным с текущим экземпляром класса, что позволяет не передавать в него self.

```
>>> class Circle(Shape):
...     def __init__(self, x, y, r):
...             super().__init__(x, y)
...             self.r = r
>>> c = Circle(1, 2, 3)
>>> vars(c)
{'y': 2, 'x': 1, 'r': 3}
```

В Python 2 метод super() требует указания двух параметров: первый — класс, для которого необходимо определить родительский класс, а второй — экземпляр класса, с которым будет связан метод базового класса.

Поведение метода super() становится не очевидным при множественном наследовании. Рассмотрим случай, когда класс отнасделован от двух классов одновременно.

```
>>> class A:
...     def __init__(self):
...             print('A __init__()')
...             super().__init__()
>>> class B:
```

```
...     def __init__(self):
...             print('B __init__()')
...             super().__init__()
>>> class C(A, B):
...     def __init__(self):
...             print('C __init__()')
...             super().__init__()
>>> C()
C __init__()
A __init__()
B __init__()
<__main__.C object at 0xb7068cec>
```

В данном случае, при создании экземпляра класса C первым делом вызывается его метод __init__(), о чем свидетельствует выведенное сообщение. Далее, благодаря вызову метода super(), происходит обращение к методу __init__() класса A, что вполне ожидаемо, поскольку он встречается первым в списке наследования. Однако, далее следовало ожидать обращение к методу __init__() класса object, т. к. он является родителем класса A, но на самом деле вызывается __init__() класса B, хотя он вообще не является предком A.

С другой стороны такое поведение соответствует логике программы, т. к. в Python инициализация полей класса происходит в методе __init__(). Чтобы класс-наследник получил все необходимые поля, требуется вызвать методы-инициализаторы всех родительских классов.

Чтобы понять, каким образом super() на каждом этапе возвращает родительский класс, необходимо рассмотреть MRO класса C.

```
>>> C.mro()
[<class '__main__.C'>, <class '__main__.A'>, <class
'__main__.B'>, <class 'object'>]
```

Как видно, порядок поиска методов класса совпадает с последовательностью вызова инициализаторов. Такой подход позволяет вызвать инициализаторы всех классов в иерархии и, при этом, однозначно определяет порядок их вызова.

Использование MRO для порядка вызова super() приводит к важному следствию: при создании любого класса, который в последствии может стать классом родителем при использовании множественного наследования, необходимо позаботиться о правильном вызове возможных перегруженных методов. В частности, если класс наследник вызывает __init__() через super(), то он желает

чтобы класс родитель после собственной инициализации передал управление всем остальным классам-родителям. Поэтому, появление super() в любом из перегруженных методов требует наличия вызова super() во всех соответствующих методах вверх по иерархии. Если в предыдущем примере опустить вызов super() в классе A, что кажется вполне уместным, т. к. он находится на вершине иерархии, то инициализатор класса B не вызовется и, в результате, класс C окажется не полностью инициализированным.

```
>>> class A:
...     def __init__(self):
...             print('A __init__()')
>>> class B:
...     def __init__(self):
...             print('B __init__()')
...             super().__init__()
>>> class C(A, B):
...     def __init__(self):
...             print('C __init__()')
...             super().__init__()
>>> C()
C __init__()
A __init__()
<__main__.C object at 0xb70987cc>
```

Обязательное использование super() во всех переопределенных методах приводит к еще одной сложности: каждый переопределенный метод должен нести ответственность за передаваемые и получаемые параметры для всех остальных методов. В частности, пример, приведенный ниже, не будет работать, т. к. класс C передает в super().__init__() два параметра для того, чтобы инициализировать собственное состояния на основе инициализаторов классов-родителей. При этом класс A не знает о том, что для него в данной цепочке наследования super-классом окажется класс B и, поэтому ожидает только один, необходимый ему параметр.

```
class A:
    def __init__(self, x):
        self.x = x
        super().__init__()
class B:
    def __init__(self, y):
        self.y = y
        super().__init__()
class C(A, B):
```

```
    def __init__(self, x, y, z):
        self.z = z
        super().__init__(x, y)
```

Решением данной проблемы может быть модификация класса A так, чтобы он получал параметры и для себя и для B.

```
class A:
    def __init__(self, x, y):
        self.x = x
        super().__init__(y)
```

Однако, такое решение не будет работать в случае, если понадобится создать экземпляр класса A или появится класс D, имеющий в списке наследования класс A совместно с другими классами. Универсальным решением является описание функции __init__() так, чтобы она получала произвольное количество аргументов.

```
class A:
    def __init__(self, *args, **kwargs):
        self.x = kwargs['x']
        super().__init__(*args, **kwargs)
class B:
    def __init__(self, *args, **kwargs):
        self.y = kwargs['y']
        super().__init__(*args, **kwargs)
class C(A, B):
    def __init__(self, *args, **kwargs):
        self.z = kwargs['z']
        super().__init__(*args, **kwargs)
```

К сожалению, такой подход тоже не будет работать.

```
>>> C(x=1, y=2, z=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in __init__
  File "<stdin>", line 4, in __init__
  File "<stdin>", line 4, in __init__
TypeError: object.__init__() takes no parameters
```

В данном случае, поскольку все инициализаторы получают полный набор аргументов и передают его далее, последний в цепочке инициализатор передаст этот же набор инициализатору класса object,

который не ожидает дополнительных параметров. Есть два способа решения данной проблемы. Во-первых, каждый метод в цепочке может удалять нужные ему параметры.

```
class A:
    def __init__(self, *args, **kwargs):
        self.x = kwargs.pop('x')
        super().__init__(*args, **kwargs)
class B:
    def __init__(self, *args, **kwargs):
        self.y = kwargs.pop('y')
        super().__init__(*args, **kwargs)
class C(A, B):
    def __init__(self, *args, **kwargs):
        self.z = kwargs.pop('z')
        super().__init__(*args, **kwargs)
>>> c = C(x=1, y=2, z=3)
>>> vars(c)
{'y': 2, 'z': 3, 'x': 1}
```

К моменту вызова __init__() класса object в списке не остается ни одного параметра, поэтому создание экземпляра происходит без ошибок. Второй вариант решения проблемы состоит в создании базового класса, от которого наследуются все классы, находящиеся на вершине иерархии.

```
class Base:
    def __init__(self, *args, **kwargs):
        pass
class A(Base):
    def __init__(self, *args, **kwargs):
        self.x = kwargs['x']
        super().__init__(*args, **kwargs)
class B(Base):
    def __init__(self, *args, **kwargs):
        self.y = kwargs['y']
        super().__init__(*args, **kwargs)
class C(A, B):
    def __init__(self, *args, **kwargs):
        self.z = kwargs['z']
        super().__init__(*args, **kwargs)
>>> c = C(x=1, y=2, z=3)
>>> vars(c)
{'y': 2, 'z': 3, 'x': 1}
```

Второе решение менее изящно, т. к. навязывает разработчику необходимость следить за тем, чтобы каждый созданный класс имел среди родителей класс Base, однако, такой подход может быть полезен в случае, когда цепочка вызовов super() применяется для методов, отсутствующих в классе object.

Необходимо отметить, что в Python отсутствуют модификаторы доступа к атрибутам класса. Т. е. поле или метод нельзя сделать недоступным за пределами класса, все атрибуты являются публичными. Однако, существует ряд соглашений, которые позволяют ограничить доступ к данным. Так, имя переменной или метода, которое начинается с символа _ считается внутренним и используется для собственных нужд класса. Когда разработчик использует подобное имя, он как бы сообщает другим разработчикам, что использование этого атрибута может привести к неожиданным последствиям. Имена, начинающиеся с двух символов _ хранятся в __dict__ в особом виде и, поэтому, обращение к ним приводит к ошибке.

```
>>> class A:
...     def __init__(self):
...             self.__secret = 0
>>> a = A()
>>> a.__secret
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute '__secret'
>>> vars(a)
{'_A__secret': 0}
>>> a._A__secret
0
```

Как видно из примера, подобные имена преобразуются к виду _имя-класса__имя-переменной. При этом, внутри класса обращение к этим переменным проблем не вызывает. Данное свойство может быть полезно при наследовании, когда классы на каждом уровне иерархии имеют свое значение определенной переменной.

```
>>> class A:
...     def __init__(self):
...             self.__secret = 0
...     def tell_A_secret(self):
...             print(self.__secret)
>>> class B(A):
...     def __init__(self):
...             self.__secret = 1
...             super().__init__()
```

```
...     def tell_B_secret(self):
...             print(self.__secret)
>>> A().tell_A_secret()
0
>>> b = B()
>>> b.tell_A_secret()
0
>>> b.tell_B_secret()
1
```

Как видно из примера, несмотря на то, что переменная __secret переопределена в классе B, метод tell_A_secret() использует значение переменной из класса A. Таким образом, методы классов на каждом уровне иерархии могут привязаны к переменным на том же уровне иерархии.

Также, иногда при создании класса бывают полезны так называемые статические переменные. Как было указано выше, переменная класса доступна без создания экземпляра. Это означает, что такая переменная будет доступна и в классах, и в экземплярах классов. Примером использования статических переменных является класс, который считает количество собственных экземпляров.

```
>>> class A:
...     counter = 0
...     def __init__(self):
...             A.counter += 1
>>> a = A()
>>> b = A()
>>> A.counter
2
```

В данном случае, переменная класса conuter является статической и содержит количество созданных экземпляров.

Для доступа к статическим переменным бывают полезны статические методы. Также, такие методы бывают полезными в случаях, когда метод не требует доступа к экземпляру класса. Функции-декораторы classmethod и staticmethod позволяют создавать такие методы и выполнять их без создания экземпляра класса, причем методы класса получают первым параметром класс, от имени которого они вызываются, а статические методы вообще не получают обязательных параметров.

```
>>> class A:
...     a = 1
```

```
...     def __init__(self):
...         self.a = 2
...     @staticmethod
...     def m1():
...           print("Static")
...     @classmethod
...     def m2(cls):
...           print(cls.a)
...     def m3(self):
...         print(self.a)
>>> A.m1()
Static
>>> A.m2()
1
>>> A.m3()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError:   m3()   missing   1   required   positional
argument: 'self'
>>> a = A()
>>> a.m3()
2
>>> a.m1()
Static
>>> a.m2()
1
```

Здесь методы m1() и m2() запускаются как для класса, так и для его экземпляра, в то время как m3() требует обязательного наличия экземпляра.

Статические методы и методы класса ведут себя по разному в случае наследования. В частности, метод класса привязан к статическим переменным текущего класса и в классах-наследниках будет пользоваться состоянием наследников. Статические методы вообще не привязаны к классу и могут получать доступ к переменным класса только через явное указание его имени.

```
>>> class A:
...     a = 1
...     @classmethod
...     def f(cls):
...             print(cls.a)
...     @staticmethod
...     def g():
...             print(A.a)
```

```
>>> class B(A):
...     a = 2
>>> A.f()
1
>>> A.g()
1
>>> B.f()
2
>>> B.g()
1
```

В данном примере метод f() использует переменную того класса, в контексте которого он вызван. В то же время метод g() всегда использует переменные класса A.

Пользователь может определять собственные типы (классы), указывая набор переменных (их еще называют полями или свойствами), содержащихся в объектах и функций (методов), которые можно к объектам применять. Кроме этого, типы можно наследовать от уже существующих типов. В таком случае объекты данных типов будут содержать в себе все собственные свойства и методы, а также свойства и методы всех родительских типов. Типы – это такие же объекты языка, как и все остальные объекты, соответственно, для создания новых типов можно воспользоваться встроенной функцией-конструктором type(), которая принимает 3 параметра. Следует обратить внимание, что функция type() с одним параметром просто возвращает его тип.

Например:

```
>>> Class = type(
     'Class',
     (object, ),
     {'a': 1, 'b': 2}
)
```

создаст новый пользовательский тип Class, именем которого будет 'Class', который унаследован от встроенного типа object и который содержит два поля a и b со значениям 1 и 2 соответственно. На самом деле существует более простой и читабельный способ создания пользовательских типов, основанный на использовании ключевого слова class:

```
class Class(object):
     a = 1
     b = 2
```

В обоих случаях, теперь с помощью нового класса можно создавать его экземпляры:

```
>>> c = Class()
>>> c
<__main__.Class object at 0xb70f4cac>
>>> c.a
1
>>> c.b
2
```

c является экземпляром класса Class, а его поля доступны через символ '.'. Строго говоря, поля a и b являются полями класса, а не объекта, но поскольку в самом объекте поля с такими именами не определены, интерпретатор пробует найти их в породившем классе.

В приведенном примере класс был унаследован от класса object. Вообще, в Python существуют классы, которые унаследованы от object и, соответственно, которые не унаследованы. Первые называются классами нового стиля, а вторые – классами старого стиля. Поскольку классы старого стиля использовать не рекомендуется, а в Python 3 от них вообще отказались, мы будем рассматривать только классы нового стиля.

Поскольку классы создаются путем вызова конструктора типа type(), что было показано в начале главы, классом, породившим класс, является type:

```
>>> class A(object):
...     pass
>>> A.__class__
<type 'type'>
```

Также в языке существует возможность более тонкой настройки способов доступа к атрибутам класса. В частности, для модификации, удаления и чтения значений атрибутов в базовом классе object предусмотрены специальные методы __setattr__(), __delattr__(), __getattribute__() и __getattr__(). Любой класс, унаследованный от object, вызывает эти методы для взаимодействия со словарем __dict__. При модификации атрибута вызывается метод __setattr__(), при удалении — __delattr__(), а при чтении алгоритм вызова несколько более сложный. Для повышения производительности программного кода, при попытке получить значение атрибута сначала вызывается

метод __getattribute__(), который пытается найти необходимое имя. Если это не удалось, вызывается метод __getattr__(), который выбрасывает исключение AttributeError. Любой из этих методов можно переопределить:

```
>>> class Verbose(object):
...     def __setattr__(self, name, value):
...      print "Set %s" % name
...      object.__setattr__(self, name, value)
...     def __delattr__(self, name):
...      print "Del %s" % name
...      object.__delattr__(self, name)
...     def __getattribute__(self, name):
...      print "Get %s" % name
...      return object.__getattribute__(self, name)
...     def __getattr__(self, name):
...      print "%s doesn't exist" % name
...      raise AttributeError
>>> v = Verbose()
>>> v.a = 1
Set a
>>> v.a
Get a
1
>>> del v.a
Del a
>>> v.a
Get a
a doesn't exist
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 13, in __getattr__
AttributeError
```

В данном примере переопределены все четыре рассмотренных метода так, что каждый из них выводит сообщение при вызове. Следует обратить внимание на то, что при добавлении новой функциональности, для сохранения работоспособности метода, как правило, необходимо дополнительно вызывать метод базового класса. В частности, реализация, приведенная в примере ниже, приведет к бесконечной рекурсии при попытке установить значение атрибута.

```
class Verbose(object):
    def __setattr__(self, name, value):
            print "Set %s" % name
```

```
            self.name = value
```

Это связано с тем, что оператор присваивания для атрибута экземпляра класса снова вызовет функцию __setattr__().

Следует также обратить внимание на встроенные функции getattr(), setattr(), delattr() и hasattr(), которые позволяют получить, установить, удалить значение и проверить наличие атрибута объекта соответственно за пределами этого объекта. С помощью этих функций можно, например, делегировать вызов методов экземпляру класса, который агрегирован другим классом.

```
>>> class Inner(object):
...     def m1(self):
...         print "m1"
...     def m2(self):
...         print "m2"
>>> class Outer(object):
...     def __init__(self):
...         self.inner = Inner()
...     def m3(self):
...         print "m3"
...     def __getattr__(self, name):
...         return getattr(self.inner, name)
>>> o = Outer()
>>> o.m1()
m1
>>> o.m2()
m2
>>> o.m3()
m3
```

Здесь в классе Outer реализован только один метод m3(), а также он агрегирует класс Inner. Вызов всех методов, которые отсутствуют в Outer, приводит в вызову __getattr__(), который, в свою очередь, делегирует его классу Inner.

Рассмотрим еще один пример с использованием функции hasattr().

```
>>> class Singleton(object):
...   def __new__(cls):
...     if not hasattr(cls, 'instance'):
...       cls.instance = object.__new__(cls)
...     return cls.instance
>>> s1 = Singleton()
>>> s2 = Singleton()
```

```
>>> s1 is s2
True
```

Для класса Singleton существует возможность создать только один экземпляр. Из примера видно, что любая повторная попытка создания экземпляра возвращает одно и то же значение. Для этого в методе __new__() с помощью функции hasattr() проверяется наличие атрибута класса instance и только при его отсутствии создается новый экземпляр.

К недостаткам переопределения методов доступа к атрибутам можно считать невозможность их повторного использования в других классах без применения наследования, а также не всегда очевидный программный код, т. к. переопределение поведения одного атрибута может повлиять на доступ к остальным. Этих недостатков лишен другой способ управления атрибутами, а именно протокол дескрипторов.

Чтобы воспользоваться этой возможностью необходимо обернуть значение атрибута в класс и реализовать в нем методы __get__(), __set__() и, при необходимости __delete__(). Если эти методы присутствуют, обращение к ним выполняется в первую очередь.

```
>>> class Length(object):
...     def __set__(self, obj, val):
...         print "Set length to %d" % val
...         obj._length = val / 100
...     def __get__(self, obj, objtype):
...         print "Get length"
...         return obj._length * 100
>>> class Line(object):
...     def __init__(self):
...             self._length = 0
...     length = Length()
>>> l = Line()
>>> l.length = 100
Set length to 100
>>> l.length
Get length
100
>>> l._length
1
```

В данном примере значение длины хранится во внутреннем представлении в метрах, но выводится и устанавливается в сантиметрах благодаря дескрипторам. Для хранения внутреннего

представления в методе __init__() класса Line создается переменная экземпляра класса _length. Переменная length класса Line, как видно из определения класса, является статической, т. е. имеет только одно значение для всех экземпляров. Для того, чтобы иметь возможность обратиться к переменным конкретного экземпляра, метод __set__() получает три параметра: первый — ссылка на экземпляр класса Length самого дескриптора (self); второй — ссылка на экземпляр класса Line, для которого производится присвоение (obj) и третий — собственно значение, которое необходимо присвоить (val). Соответственно внутри метода __set__() новое значение присваивается переменной экземпляра класса Line путем обращения к параметру obj, а не параметру self. Аналогично, метод __get__() принимает параметры self и obj, а также дополнительный параметр objtype, значение которого определяет класс, для которого вызывается данный метод, в данном случае это будет Line.

Третья встроенная функция — property() — предназначена для управления доступом ко значениям переменных и фактически является высокоуровневым способом определения дескрипторов. С ее помощью можно создавать поля класса, которые в объектно-ориентированных языках принято называть свойствами. Свойства извне выглядят как обычные переменные экземпляра класса, но при обращении к ним выполняется дополнительный, определенный пользователем, программный код. В следующем примере определен класс, который работает аналогично классу Line в примере с дескриптором, приведенным выше.

```
>>> class Line(object):
...     def __init__(self):
...             self._length = 0
...     def set_l(self, val):
...             self._length = val / 100
...     def get_l(self):
...             return self._length * 100
...     length = property(get_l, set_l)
>>> l = Line()
>>> l.length = 100
>>> l.length
100
>>> l._length
1
```

Функция property() также может принимать третий параметр, определяющий функцию, вызываемую при удалении атрибута. Использование данной функции в виде декоратора выглядит еще более

компактно:

```
>>> class Line(object):
...     def __init__(self):
...             self._length = 0
...     @property
...     def length(self):
...             return self._length * 100
...     @length.setter
...     def length(self, val):
...             self._length = val / 100
>>> l = Line()
>>> l.length = 100
>>> l.length
100
>>> l._length
1
```

Порядок создания экземпляра класса перед функцией __init__() предполагает вызов еще одной функции __new__(), которая возвращает новый экземпляр. Изменив ее поведение можно получить класс совершенно другого типа.

```
>>> class New(object):
...     def __new__(cls):
...             return 1
...
>>> a = New()
>>> a
1
>>> type(a)
<type 'int'>
```

В данном примере функция __new__() вместо экземпляра требуемого класса вернула целочисленное значение. Данное значение полностью соответствует типу int и не несет в себе никакой информации о типе New. Более реальным примером использования переопределения функции __new__() является создание фабрики объектов, которая позволяет на основе строкового ключа создавать объекты разных типов.

```
 >>> class Truck(object):
...     def go(self):
...      print "RRRR"
```

```
>>> class Car(object):
...     def go(self):
...      print "Bzzz"
>>> class Vehicle(object):
...     def __new__(cls, desc):
...      if desc == 'Truck':
...       return super(Vehicle, cls).__new__(Truck)
...      if desc == 'Car':
...       return super(Vehicle, cls).__new__(Car)
...      raise ValueError
>>> v1 = Vehicle('Truck')
>>> v2 = Vehicle('Car')
>>> v3 = Vehicle('Plane')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in __new__
ValueError
>>> v1.go()
RRRR
>>> v2.go()
Bzzz
>>> type(v1)
<class '__main__.Truck'>
>>> type(v2)
<class '__main__.Car'>
```

Здесь были созданы два класса, для каждого из которых определен метод go(). Для создания экземпляров данного класса используется третий класс с обобщающим именем Vehicle. Этот класс получает строковое значение, от которого зависит тип объекта, который будет создан. Функция __new__() получает не ссылку на экземпляр объекта, такой как self, которого на самом деле еще не существует, а класс, объект которого должен быть создан. Как было указано выше, классы являются типами, а типы — это тоже объекты языка. Благодаря этому, в Python есть возможность создания метаклассов, т. е. классов, порождающих новые классы.

Поскольку, функция type() от трех аргументов является конструктором класса, то чтобы класс стал метаклассом его необходимо унаследовать не от object, а от type.

```
>>> class Meta(type):
...  def __new__(cls, name, parents, attrs):
...   new = {}
...   for key, value in attrs.items():
...    if not key.startswith('unused_'):
```

```
...     new[key] = value
... return type.__new__(cls, name, parents, new)
>>> class A(object):
...     __metaclass__ = Meta
...     a = 0
...     unused_a = 0
>>> A.unused_a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'A' has no attribute
'unused_a'
>>> A.a
0
```

В данном примере создается метакласс, который анализирует атрибуты порождаемого класса и удаляет те, имена которых начинаются со строки 'unused_'. Далее создается класс A, для которого указан соответствующий метакласс. В результате, несмотря на то, что в классе A была объявлена переменная unused_a, в созданном классе она отсутствует. Таким образом, метаклассы могут влиять на процесс создания новых классов, определенным образом настраивая их поведение.

На механизме метаклассов основан модуль стандартной библиотеки abc (Abstract Base Classes). Этот модуль позволяет описывать абстрактные классы, т. е. классы, экземпляры которых нельзя создавать, но они обязательно требуют определения определенных в них методов в классах-наследниках. Создание таких классов позволяет гарантировать наличие необходимых методов и, таким образом, соответствие определенному интерфейсу.

```
>>> import abc
>>> class Printer(object):
...     __metaclass__ = abc.ABCMeta
...     @abc.abstractmethod
...     def print_a(self):
...             pass
>>> p = Printer()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Printer
with abstract methods print_a
>>> class CPrinter(Printer):
...     def print_b(self):
...             print 'b'
```

```
>>> c = CPrinter()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class CPrinter
with abstract methods print_a
>>> class CPrinter(Printer):
...     def print_a(self):
...             print 'a'
>>> c = CPrinter()
>>> c.print_a()
a
```

Из примера видно, что ни для базового класса, ни для классов-наследников, в которых отсутствует реализация абстрактного метода создание экземпляров невозможно.

Классы в Python могут иметь ряд методов, которые неявно вызываются в особых ситуациях. В частности, методы __str__() и __repr__() используются для получения строкового представления класса. Причем __str__() чаще применяется для вывода информации об экземпляре класса пользователю, в то время как __repr__() — для отладочной информации. В частности оператор print сначала пытается вызвать __str__(), если он не определен, тогда вызывается __repr__(), и, наконец, в случае неудачи вызываются аналогичные методы базового класса.

```
>>> class A(object):
...     pass
...
>>> a = A()
>>> print a
<__main__.A object at 0xb704ca8c>
>>> A.__repr__ = lambda self: 'repr A'
>>> print a
repr A
>>> A.__str__ = lambda self: 'str A'
>>> print a
str A
>>> `a`
'repr A'
```

В данном примере в класс динамически добавляются методы __repr__() и __str__(). При их отсутствии попытка вывода содержимого экземпляра класса приводит к стандартному для класса object поведению. Запись `a` в Python 2 эквивалентна вызову функции

__repr__().

Другими важными методами, которые можно определить для класса являются __eq__(), __ne__(), __gt__(), __lt__(), __ge__(), __le__(). Эти методы вызываются при попытке сравнения экземпляров класса.

```
>>> class Person(object):
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...     def __eq__(self, other):
...         return self.name == other.name
...     def __lt__(self, other):
...         return self.name < other.name
...     def __repr__(self):
...         return "%s, %d" % (self.name, self.age)
>>> p1 = Person('Bill', 25)
>>> p2 = Person('Bob', 25)
>>> p3 = Person('Bill', 26)
>>> p1 == p3
True
>>> p1 == p2
False
>>> p3 < p2
True
>>> p3 > p2
False
>>> l = [p1, p2, p3]
>>> l.sort()
>>> l
[Bill, 25, Bill, 26, Bob, 25]
```

После определения двух операций — == и < интерпретатор способен самостоятельно определить остальные операции сравнения при помощи декоратора total_ordering, определенного в модуле functools. Кроме того, определения данных операций достаточно для того, чтобы функция сортировки работала в соответствии с установленными правилами.

Вообще, возможность переопределения существует для всех операций языка. Например, для рассмотренного выше класса Person можно переопределить операцию сложения, хотя это и не имеет смысла для данной предметной области.

```
>>> Person.__add__ = lambda self, other: self.age +
```

```
other.age
>>> p1 + p2
50
```

На самом деле, чтобы заставить встроенные операторы полноценно работать с экземплярами пользовательских классов реализации одного метода недостаточно. Рассмотрим пример класса, который инкапсулирует в себе числовое значение.

```
>>> class Number(object):
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return "Number: {}".format(self.value)
...     def __add__(self, other):
...         return Number(self.value + other.value)
>>> a = Number(1)
>>> b = Number(2)
>>> a + b
Number: 3
```

При сложении экземпляров класса Number проблем не возникает, но при попытке прибавить к экземпляру данного класса число возникнет ошибка:

```
>>> a + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in __add__
AttributeError: 'int' object has no attribute 'value'
```

Причиной данной ошибки является то, что метод __add__() считает, что оба входных параметра относятся к классу Number и, поэтому пытается получить значение value для числа. Чтобы избежать этой ошибки, необходимо проверить тип второго параметра, и если это число, выполнить соответствующее преобразование.

Определение типа значение возможно несколькими способами. Наиболее очевидным является способ сравнения результата функции type() с именем типа.

```
>>> type(1) == int
True
```

При всей простоте данного метода, он не позволяет получить

удовлетворительный результат для любых чисел кроме целых.

```
>>> type(1l) == int
False
>>> type(1.0) == int
False
```

Другим способом проверить относится ли заданное значение к определенному типу является использование встроенной функции isinstance(), которая выполняет не просто сравнение, а принимает во внимание наследование.

```
>>> import numbers
>>> isinstance(1, numbers.Number)
True
>>> isinstance(1.0, numbers.Number)
True
>>> isinstance(1l, numbers.Number)
True
>>> isinstance(1 + 2j, numbers.Number)
True
```

В данном примере было выполнено импортирование наиболее общего класса чисел из встроенного модуля number. Как видно, для любого значения, относящегося к числовым типам isinstance() возвращает True. Кроме того, вторым параметром функции isinstance() может быть кортеж типов.

```
>>> isinstance(1l, (int, long))
True
```

Таким образом, метод __add__() в классе Number может быть переписан следующим образом.

```
...     def __add__(self, other):
...         if isinstance(other, numbers.Number):
...             other = Number(other)
...         return Number(self.value + other.value)
>>> a = Number(1)
>>> a + 1
Number: 2
```

Следующая проблема возникает при попытке сложить число с экземпляром класса Number.

```
>>> 1 + a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int'
and 'Number'
```

При попытке выполнить сложение двух значений Python сначала пытается найти реализацию метода __add__() у левого операнда. Если такой метод отсутствует или не поддерживает сложение со значениями, имеющими тип правого операнда, производится попытка найти у правого операнда метод __radd__(). Таким образом, чтобы для данного примера сложение работало корректно, необходимо добавить реализацию метода __radd__() в класс Number.

```
...     def __radd__(self, other):
...         return self.__add__(other)
>>> a = Number(1)
>>> 1 + a
Number: 2
```

Реализация метода __add__() автоматически добавляет возможность использования оператора +=.

```
>>> a += 3
>>> a
Number: 4
```

Однако, существует возможность переопределения поведения и этого оператора.

```
...     def __iadd__(self, other):
...         raise ValueError
>>> a = Number(1)
>>> a += 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 13, in __iadd__
ValueError
```

Другим полезным методом классов является метод __hash__(), которая используется для определения хеш-кода объекта. Хеш-коды используются при добавлении объектов в словари и множества. Переопределение метода __hash__() может быть полезно для указания свойств объекта, определяющих его уникальность. Переопределим

данный метод для класса Person из предыдущего примера.

```
>>> set([p1, p2, p3])
set([Bill, 26, Bob, 25, Bill, 25])
>>> Person.__hash__ = lambda self: hash(self.name)
>>> set([p1, p2, p3])
set([Bob, 25, Bill, 25])
```

В данном случае для вычисления хеш-кода используется встроенная функция hash(), но аргументом этой функции выступает только поле name, а не оба поля класса, поэтому экземпляры p1 и p3 оказываются идентичными для множества.

Не менее полезной является возможность создания функторов, т. е. классов, экземпляры которых можно вызывать подобно функциям. Для реализации подобной функциональности необходимо определить метод __call__().

```
>>> class Multiplier(object):
...     def __init__(self, x):
...             self.x = x
...     def __call__(self, y):
...             return self.x * y
>>> double = Multiplier(2)
>>> double(3)
6
```

Функторы иногда используются для создания декораторов с параметрами.

```
>>> class scale(object):
...     def __init__(self, x):
...             self.x = x
...     def __call__(self, f):
...             def wrapper(y):
...                     return f(self.x * y)
...             return wrapper
>>> @scale(10)
... def f(x):
...     return 2 * x
>>> f(2)
40
```

В данном примере в качестве декоратора используется экземпляр класса, который сначала создается путем вызова конструктора с параметром, а затем сразу же вызывается в момент декорирования.

Часто классы, создаваемые пользователем, имеют характерное для контейнеров поведение. Существует два способа организации подобных классов. В первом случае новый класс-контейнер наследуется от существующего контейнера с определением дополнительных методов.

```
>>> class SumList(list):
...     def sum(self):
...             s = 0
...             for i in self:
...                     s += i
...             return s
>>> l = SumList([1, 2, 3])
>>> l.sum()
6
```

Новый класс ведет себя также как список, но имеет дополнительный метод для подсчета суммы элементов. Такой подход имеет существенный недостаток: новый класс полностью копирует интерфейс базового контейнера, что в большинстве случаев может быть излишним. Поэтому чаще используется второй вариант организации, основанный на агрегации контейнера. Для того, чтобы имитировать поведение контейнера необходимо определить три метода — __contains__(), который вызывается при использовании оператора in; __len__() для определения количества элементов и __getitem__() для получения элемента по индексу.

```
>>> class MyList(object):
...     def __init__(self, *args):
...             self.l = args
...     def __contains__(self, el):
...         return el in self.l
...     def __len__(self):
...             return len(self.l)
...     def __getitem__(self, index):
...             return self.l[index]
>>> l = MyList(1, 2, 3, 4)
>>> l[0]
1
>>> 3 in l
True
>>> 7 in l
False
>>> len(l)
4
```

```
>>> l[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 9, in __getitem__
IndexError: tuple index out of range
```

Видно, что поведение нового списка полностью идентично поведению традиционного списка, однако модификация содержимого находится под полным контролем автора класса. В принципе обход нового контейнера с помощью цикла for также возможен, но он основан на устаревшем протоколе итерирования.

```
>>> for i in l:
...     print i
...
1
2
3
4
```

В данном случае интерпретатор вызывает метод __getitem__() для последовательности индексов начиная с 0 и до получения исключения IndexError. Такой подход работает только в случае, если __getitem__() определен и индексы в контейнере идут строго последовательно. Поэтому, современные реализации интерпретатора для прохода по значениям контейнера используют протокол итераторов. Итератором является объект, который реализует два метода — iter() для получения ссылки на самого себя и next() для получения следующего элемента.

```
>>> class Iter(object):
...     def __init__(self):
...         self.i = 0
...     def __iter__(self):
...         return self
...     def next(self):
...         if self.i < 5:
...             self.i += 1
...             return self.i
...         raise StopIteration
>>> for i in Iter():
...     print i
1
2
3
4
```

```
5
```

В данном примере класс Iter имитирует контейнер. Аналогичного эффекта можно достичь при помощи старого протокола.

```
>>> class OldStyle(object):
...     def __init__(self):
...         self.i = 0
...     def __getitem__(self, index):
...         if self.i < 5:
...             self.i += 1
...             return self.i
...         raise IndexError
...
>>> for i in OldStyle():
...     print i
...
1
2
3
4
5
```

Генераторы

Протокол итератора для классов-контейнеров может быть гораздо проще реализован с помощью объектов-генераторов. Для создания генератора необходимо определить функцию, которая содержит ключевое слово yield. Например,

```
>>> def f():
...     for i in range(5):
...             yield i
...             print("I'm here")
```

Наличие оператора yield в коде превращает данную функцию в фабрику генераторов. Результатом выполнения такой функции будет генераторный объект, который, в том числе, соответствует протоколу итератора.

```
>>> gen = f()
>>> dir(gen)
['__class__',  '__del__',  '__delattr__',  '__dir__',
'__doc__',    '__eq__',    '__format__',    '__ge__',
```

```
'__getattribute__', '__gt__', '__hash__', '__init__',
'__iter__', '__le__', '__lt__', '__name__', '__ne__',
'__new__', '__next__', '__qualname__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', 'close',
'gi_code', 'gi_frame', 'gi_running', 'gi_yieldfrom',
'send', 'throw']
```

Поскольку полученный генератор реализует метод __next__(), он может быть активирован при помощи встроенной функции next():

```
>>> next(gen)
0
>>> next(gen)
I'm here
1
>>> next(gen)
I'm here
2
>>> next(gen)
I'm here
3
>>> next(gen)
I'm here
4
>>> next(gen)
I'm here
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

В результате первого вызова next() начинается выполнение программного кода функции: происходит инициализация цикла for и затем отрабатывает оператор yield. В данном случае, он ведет себя как оператор return, что приводит к выходу из функции и возвращению значения переменной i равного 0. При повторном вызове next() выполнение продолжается с оператора, следующего непосредственно за yield, о чем свидетельствует сообщение «I'm here». Как видно из примера, каждый следующий вызов приводит к выполнению следующей итерации цикла и так происходит до тех пор, пока цикл не завершится. Выход из цикла приводит к выходу из генератора, что приводит возбуждению исключения StopIteration, которое

сигнализирует о завершении итерирования в соответствии с протоколом итератора. Аналогично, генераторные объекты могут быть использованы в цикле for.

```
>>> for i in f():
...     print(i)
0
I'm here
1
I'm here
2
I'm here
3
I'm here
4
I'm here
```

Генераторные объекты могут быть использованы в программном коде для реализации как минимум трех оригинальных подходов, а именно:

1) итераторов для классов-контейнеров

2) ленивых вычислений

3) сопрограмм (coroutine)

Как было указано выше, класс-контейнер, удовлетворяющий протоколу итератора может быть реализован очень простым способом:

```
>>> class MyList:
...     def __init__(self, l=[]):
...             self._l = list(l)
...     def __iter__(self):
...             for i in self._l:
...                     yield i
>>> l = MyList([1,2,3,4])
>>> for i in l:
...     print(i)
1
2
3
4
```

Магический метод __iter__() в данном случае представляет собой функцию, содержащую оператор yield, которая, в свою очередь, при вызове возвращает объект, содержащий метод __next__(). Это позволяет выполнять итерирование по экземпляру класса-контейнера в цикле for.

В последнем случае поведение итератора фактически дублирует поведение итератора внутреннего списка. Когда возникает необходимость делегирования поведения генератора другому генератору, можно использовать специальную форму оператора yield — yield from.

```
>>> class MyList:
...     def __init__(self, l=[]):
...             self._l = list(l)
...     def __iter__(self):
...             yield from self._l
>>> l = MyList([1, 2, 3])
>>> for i in l:
...     print(i)
1
2
3
```

Второй способ использования генераторных объектов состоит в реализации ленивых вычислений. Данный подход характерен для функциональных языков программирования, где функции не имеют побочных эффектов. Суть ленивых вычислений состоит в том, что значения выражений вычисляются только в случае необходимости. Например, допустим, что есть некая функция f(), требующая значительных ресурсов для вычисления. Допустим также, что существует функция g() следующего вида:

```
def g(x, y):
    if x < 0:
        return y
    return 0
```

При вызове данной функции следующим способом:

```
g(1, f(...))
```

неленивые языки, такие как Python, будут вычислять значение f(), даже если, как в данном случае, оно не требуется. В свою очередь ленивый язык отложит вычисление функции f() до тех пор, пока оно действительно не понадобится, а в данном случае вообще не будет его выполнять.

Императивные языки как правило не могут самостоятельно откладывать вычисления функций, т. к. они допускают возможность доступа к глобальному пространству имен, что позволяет функциям

иметь побочные эффекты. Если вычисление функции f(), которая обращается к глобальным переменным, было отложено, то на момент ее выполнения состояние глобального пространства имен может измениться и результат вычислений окажется неправильным. Тем не менее Python позволяет создавать ленивые конструкции явно при помощи генераторных объектов.

Например, допустим необходимо найти положительные целые числа, квадраты которых меньше 100. Если заранее не известно, сколько таких чисел будет, можно создать ленивый бесконечный список, который будет служить источником данных для кода, вычисляющего квадраты:

```
>>> def infinity_list():
...     i = 1
...     while True:
...             yield i
...             i += 1
```

Данный генератор при каждом обращении к нему будет возвращать следующее целое число.

```
>>> for i in infinity_list():
...     if i*i >= 100:
...             break
...     print(i)
1
2
3
4
5
6
7
8
9
```

В данном случае, итерирование по бесконечному списку прерывается при получении первого числа, удовлетворяющего искомому условию.

Большое количество функций, работающих с объектами, которые реализуют протокол итератора, включено в модуль стандартной библиотеки itertools. В частности, предыдущий пример с бесконечным списком может быть компактно переписан с использованием функций этого модуля.

```
>>> import itertools
```

```
>>> list(itertools.takewhile(lambda x: x*x <= 100,
itertools.count(1)))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Функция itertools.count() формирует бесконечный список, начиная с заданного значения, а функция itertools.takewhile() выполняет итерирование по входному объекту, пока функция, переданная в качестве первого параметра не вернет False.

Другим классическим примером использования генераторов является построение решета Эратосфена для вычисления последовательностей простых чисел. Сложность задачи генерации простых чисел определяется отсутствием аналитического способа нахождения числа с произвольным номером в последовательности. Заранее нельзя определить, до какого числа необходимо производить вычисления, чтобы получить заданное количество чисел.

Решето Эратосфена представляет собой бесконечную последовательность натуральных чисел, из которого последовательно вычеркиваются значения, кратные уже найденным простым числам. Для этого необходимо найти первое невычеркнутое число — оно будет простым, а затем вычеркнуть все кратные ему. При повторении процесса будет получено следующее простое число. Процесс вычеркивания чисел можно записать следующим генераторным объектом.

```
>>> def filter_mult(n, gen):
...     for i in gen:
...             if i % n:
...                     yield i
```

В данном случае, на вход функции приходит простое число и последовательность невычеркнутых чисел. Генератор возвращает новую последовательность, содержащую только числа, которые не делятся на n.

Получение следующего простого числа может быть построено также на генераторе.

```
>>> def get_prime():
...     c = itertools.count(2)
...     while True:
...             prime = next(c)
...             c = filter_mult(prime, c)
...             yield prime
```

Здесь в качестве начальной последовательности строится бесконечный список, начинающийся с 2. Далее в цикле последовательно извлекается первое число из последовательности, а затем последовательность фильтруется при помощи ранее определенной функции. Таким образом строится рекурсивная последовательность вызовов генераторных объектов. Получить первые 10 простых чисел начиная с 2, можно при помощи функции itertools.islice(), которая возвращает срез-итератор, что позволяет избавиться от необходимости вычисления бесконечных последовательностей:

```
>>> list(itertools.islice(get_prime(), 0, 10))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Другой полезной функцией из модуля itertools является groupby(). Она позволяет сгруппировать последовательности одинаковых значений в отдельные итерируемые объекты. Эта функция для любого объекта-контейнера возвращает итератор, который по запросу формирует пару: следующее значение из контейнера и итератор, который повторяет данное значение столько раз, сколько оно последовательно встречается в контейнере.

```
>>> [(k, list(v)) for k, v in itertools.groupby([0, 0,
0, 1, 2, 2, 0, 0, 2, 2, 2])]
[(0, [0, 0, 0]), (1, [1]), (2, [2, 2]), (0, [0, 0]),
(2, [2, 2, 2])]
```

Данная функция позволяет реализовать алгоритм сжатия Run-length Encoding (RLE) одной строкой кода. Этот алгоритм хорошо подходит для сжатия векторных изображений, которые содержат длинные последовательности одинаковых значений. Идея алгоритма состоит в том, чтобы для каждого набора повторяющихся значений хранить их количество и само значение. С помощью groupby() подобное представление можно получить следующим образом.

```
>>> [(len(list(v)), k) for k, v in
itertools.groupby([0, 0, 0, 1, 2, 2, 0, 0, 2, 2, 2])]
[(3, 0), (1, 1), (2, 2), (2, 0), (3, 2)]
```

Также для генерации различных комбинаций наборов значений бывают полезными комбинаторные функции модуля itertools, такие как product() для получения декартового произведения входных

последовательностей, permutations() для получения всех возможных комбинаций элементов входного набора заданной длины, combinations() и combinations_with_replacement() для получения всех возможных комбинаций с соблюдением порядка следования с повторениями и без.

```
>>> list(itertools.product('ABC', 'DEF'))
[('A', 'D'), ('A', 'E'), ('A', 'F'), ('B', 'D'), ('B',
'E'), ('B', 'F'), ('C', 'D'), ('C', 'E'), ('C', 'F')]
>>> list(itertools.permutations('ABC', 2))
[('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C',
'A'), ('C', 'B')]
>>> list(itertools.combinations('ABC', 2))
[('A', 'B'), ('A', 'C'), ('B', 'C')]
>>>
list(itertools.combinations_with_replacement('ABC',
2))
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'B'), ('B',
'C'), ('C', 'C')]
```

Следует обратить внимание на еще один способ создания генераторного объекта с помощью списочного выражения в круглых скобках.

```
>>> i = (x*x for x in range(5))
>>> i
<generator object <genexpr> at 0xb6ff547c>
>>> for j in i:
...     print(j)
0
1
4
9
16
```

В данном случае создается объект, который позволяет вычислить значения квадратов чисел от 0 до 4, но делает это не сразу, а по соответствующему запросу.

Как было упомянуто выше, генераторные объекты также могут быть использованы для организации особого механизма взаимодействия между функциями, известного как сопрограммы. Данный механизм предполагает возможность многократной передачи

управления от одной функции к другой и назад. Фактически такой подход эмулирует кооперативную многозадачность, когда два псевдопараллельных потока время от времени уступают управление друг другу. Механизм сопрограмм используется при асинхронном взаимодействии и имеет готовую реализацию в модуле asyncio, а начиная с версии языка 3.5 поддерживается в синтаксисе языка при помощи специальных ключевых слов.

Для обмена данными с сопрограммой используется оператор yield и метод генераторного объекта send(). Оператор yield позволяет вернуть значение из сопрограммы, приостановив ее выполнение, а метод send() запускает приостановленную сопрограмму и передает ей параметры. Рассмотрим пример сопрограммы, которая выполняется псевдопараллельно с главной функцией и обменивается с ней данными.

```
@coroutine
def f():
    print('f start')
    i = yield
    print('f:', i)
    i = yield i + 1
    print('f:', i)
    i = yield i + 1
```

Наличие в данной функции оператора yield превращает ее в фабрику генераторов. Для того, чтобы из фабрики получить генератор используется декоратор coroutine.

```
def coroutine(f):
    gen = f()
    next(gen)
    return gen
```

В первой строке этого декоратора вызывается функция f() для того, чтобы получить генератор. Далее данный генератор выполняется до первого оператора yield и возвращает управление. В этот момент генератор готов к выполнению в режиме сопрограммы и ждет вызова send() чтобы продолжить вычисления. Последняя строка декоратора возвращает генератор, позволяя ему подменить функцию f() собой, так чтобы главная функция смогла обращаться к сопрограмме по имени f. Оставшаяся часть кода может выглядеть так.

```
def main():
    print('main start')
```

```
    i = f.send(1)
    print('main:', i)
    i = f.send(i + 1)
    print('main:', i)


print('--------------')
main()
```

Результат выполнения данного примера будет выглядеть следующим образом.

```
f start
--------------
main start
f: 1
main: 2
f: 3
main: 4
```

Как видно, выполнение функции f() начинается еще до вызова main() поскольку декоратор выполняется при создании функции. Вызов функции main() приводит к первому вызову send(), который передает в сопрограмму значение 1 и запускает ее выполнения начиная с первого оператора yield. Значение, переданное в send() помещается в переменную i, выводится на экран, увеличивается на 1 и возвращается в main(). Функция main() выводит полученное значение, снова его увеличивает и опять передает в f() теперь уже в строку с вторым оператором yield. Таким образом, данные функции выполняются попеременно, постоянно передавая управление друг другу. Данный подход используется при асинхронном взаимодействии с внешними ресурсами, когда одна сопрограмма отправляет неблокирующий запрос и отдает управление до тех пор, пока не придет ответ, что позволяет в то же время исполнять другие сопрограммы.

Кроме метода send() генераторные объекты поддерживают методы throw() и close(). Метод throw() позволяет сгенерировать определенное исключение в внутри генератора. Благодаря этому становится возможным предусмотреть специальное поведение внутри генератора в особых случаях, например, когда отправляемые данные некорректны. Метод close() генерирует внутри генератора специальное исключение GeneratorExit, предназначенное для завершения работы с генератором и позволяющее ему выполнить завершающие действия, например, освободить ресурсы. Рассмотрим

пример, в котором генератор используется как сопрограмма для получения данных и их последующего вывода на экран. Подобная сопрограмма может быть использована, например, для инкапсуляции процесса отправки данных другому узлу сети через сокеты. В данном случае, данные, полученные через оператор yield печатаются на экран определенным образом. Если входные данные по каким-либо причинам не соответствуют требованиям бизнес-логики, и вызывающая функция не должна знать о представлении ошибок при выводе, она может просто возбудить исключение через метод throw(). Это исключение должна перехватить сопрограмма и сформировать соответствующее выводимое сообщение. Также в данном генераторе предусмотрена возможность реагирования на метод close() путем обработки исключения GeneratorExit.

```
>>> def writer():
...     while True:
...             try:
...                     i = yield
...                     print(i)
...             except ValueError:
...                     print('***')
...             except GeneratorExit:
...                     print('Goodbye')
...                     break
>>> w = writer()
>>> next(w)
>>> w.send(1)
1
>>> w.throw(ValueError)
***
>>> w.send(1)
1
>>> w.close()
Goodbye
>>> w.send(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Следует обратить внимание на то, что проигнорировать исключение GeneratorExit нельзя, т. к. это приведет к ошибке во время выполнения.

```
>>> def f():
...     while True:
```

```
...             try:
...                     i = yield
...                     print(i)
...             except:
...                     pass
...
>>> g = f()
>>> next(g)
>>> g.close()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: generator ignored GeneratorExit
```

При попытке реализации обертки для генератора writer() из примера, приведенного выше, весь код генератора должен быть продублирован, поскольку весь набор обрабатываемых исключений должен быть перехвачен и в обертке и в самом writer(). Чтобы избежать подобного дублирования также может быть использована конструкция yield from.

```
>>> w = writer()
>>> next(w)
>>> def gen():
...     yield from w
>>> g = gen()
>>> next(g)
None
>>> g.send(1)
1
>>> g.throw(ValueError)
***
>>> g.close()
Goodbye
```

В данном случае, все действия, выполняемые с генератором gen() перенаправляются генератору writer(). Как видно из примеров, конструкция yield from может быть использована как двунаправленный канал, позволяющий через генератор-обертку передавать данные как из так и в сопрограмму.

Модули

Любой текстовый файл, содержащий код на языке Python и имеющий расширение .py, является модулем. Это означает, что такой файл может быть запущен из командной строки при помощи

интерпретатора Python или использован в других модулях путем импортирования. Выделение части программного кода в отдельный модуль позволяет получить ряд преимуществ. В первую очередь модули позволяют сосредоточить в одном месте переменные, функции и классы, предназначенные для решения определенной задачи, и, таким образом, уменьшить количество зависимостей. Во-вторых, модули формируют пространства имен, благодаря чему появляется возможность одновременного использования одноименных объектов в разных контекстах. Кроме того, модули упрощают повторное использование кода.

К важным характеристикам модулей, определяющим удобство их использования и модификации, относятся связность (coupling) и сцепленность (cohesion). Под связностью обычно понимают то, насколько данный модуль зависит или использует объекты других модулей, причем связность может быть сильной, когда один модуль непосредственно изменяет значения переменных другого, и слабой, когда из другого модуля только вызываются функции. Сцепленность в свою очередь определяет, насколько компоненты модуля связаны между собой по функциональности: низкая или случайная сцепленность характерна для модулей, в которых собраны сервисные функции из разных частей проекта просто потому, что их больше некуда поместить, а высокая сцепленность предполагает, что все функции модуля предназначены для решения одной и той же задачи. Хорошо спроектированные модули обладают низкой связностью и высокой сцепленностью.

Как было указано выше, любой модуль может быть запущен как отдельная программа и проимпортирован из других модулей. В обоих случаях интерпретатор последовательно выполнит все операторы, которые содержатся в модуле. Например, для модуля hello.py :

```
# hello.py
print "Hello!"
```

запуск из командной строки будет выглядеть следующим образом:

```
~ >python hello.py
Hello!
```

Импортирование из другого модуля:

```
>>> import hello
Hello!
```

В последнем случае, несмотря на то, что код импортируется в интерактивном режиме интерпретатора, все равно он исполняется в контексте модуля. На самом деле, при запуске интерпретатора автоматически создается модуль с именем __main__, который определяет глобальное пространство имен. Все определения глобальных переменных и функций будут относиться к этому модулю. Также в контексте этого модуля выполняются программы, запущенные из командной строки. Для того, чтобы определить, в каком контексте запущен модуль, можно воспользоваться встроенной переменной __name__. При выполнении кода в интерактивном режиме или при запуске модуля как самостоятельной программы данная переменная будет иметь значение __main__, в то время как в случае импортирования она примет значение, равное имени проимпортированного модуля.

```
# test_name.py
print __name__
…
```

Из командной строки:

```
~ >python test_name.py
__main__
…
```

Из интерпретатора:

```
>>> import test_name
test_name
```

Как видно из примера, разные способы запуска приводят к получению разных значений __name__. На этом факте основан способ определения контекста запуска, который часто используется в модулях различных библиотек.

```
…
# переменные и функции библиотеки
…
if __name__ == '__main__':
    # тестовый программный код
```

В данном случае, при импорте модуля тестовый код выполнен не будет, т. к. переменная __name__ будет иметь значение имени модуля. В противном случае, при запуске из командной строки, тесты будут

запущены.

Для импорта модулей существует несколько различных способов. В частности, наиболее популярным является использование ключевого слова import. Допустим, в модуле m1.py содержится следующий код:

```
# m1.py

a = 1
b = 2

print "m1 imported"
```

Результат его импорта будет выглядет так.

```
>>> import m1
m1 imported
>>> m1.a
1
>>> m1.b
2
```

Как было указано выше, при импорте модуля последовательно выполняются все описанные в нем инструкции. Соответственно, в данном случае, в глобальном контексте модуля m1 были созданы две переменные a и b, а затем был выполнен оператор print. Результат выполнения модуля, а именно все объекты, которые были в нем определенны, помещаются в объект с именем m1, и, соответственно, доступ к этим объектам можно получить через это имя. Объект модуля служит пространством имен для всех определенных в нем объектов и в то же время выполняет роль своеобразного кеша. Это выражается в том, что при повторном импортировании модуля его код не выполняется, а используются полученные ранее объекты.

```
>>> m1.a = 2
>>> import m1
>>> m1.a
2
```

В данном случае, состояние переменной модуля было изменено. Если бы код модуля при повторном импорте выполнялся, переменная m1.a приняла бы первоначальное значение, чего, как видно, не произошло. Также не был выполнен оператор print. Для определения модулей, которые уже были импортированы используется внутренний

словарь, доступ к которому можно получить через модуль sys.

```
>>> import sys
>>> sys.modules
{'copy_reg':       <module       'copy_reg'       from
'/usr/lib/python2.7/copy_reg.pyc'>,  …  '__builtin__':
<module  '__builtin__'  (built-in)>,  ...  '__main__':
<module '__main__' (built-in)>, … 'm1': <module 'm1'
from 'm1.py'>, … 'sys': <module 'sys' (built-in)>, …
'_weakref': <module '_weakref' (built-in)>}
```

Вывод содержимого словаря весьма обширен и может отличаться от версии к версии, поэтому в данном примере часть модулей была опущена. Тем не менее видно, что в словаре содержатся модули __builtin__ с определениями встроенных имен, __main__, соответствующий глобальному контексту и проимпортированные в текущей сессии модули sys и m1. Поскольку модуль __main__ отвечает за глобальный контекст, встроенная функция globals() возвращает его содержимое.

```
>>> sys.modules['__main__'].__dict__ == globals()
True
>>> sys.modules['__main__'].__dict__['c'] = 5
>>> c
5
>>> globals()['c'] = 4
>>> c
4
```

Данная особенность может быть использована для доступа к глобальным переменным внутри функций и даже для экспорта локальных переменных в глобальный контекст.

```
>>> def f(x):
...     globals()['x'] = x
...
>>> f(5)
>>> x
5
```

Следует обратить внимание на то, что функции, определенные в модуле при обращении к глобальным переменным используют

контекст модуля, а не глобальный контекст.

```
# global_state.py
a = 1
def f():
    return a
```

Здесь в модуле global_state определяется функция f(), которая возвращает значение глобальной переменной.

```
>>> import global_state
>>> a = 5
>>> global_state.f()
1
```

Несмотря на то, что в глобальном контексте объявлена переменная a, при вызове функции обращение выполняется к переменной, определенной в модуле. На самом деле, каждая импортированная функция содержит ссылку на словарь глобального контекста модуля, к которому, в случае необходимости происходит обращение.

```
>>> global_state.f.__globals__
{'a': 1, 'f': <function f at 0xb748d79c>,
'__builtins__': ...
```

При импорте модуля, оператор import последовательно формирует из имени модуля и расширений .py, .pyc и других из списка допустимых расширений имя файла и делает попытку найти файл с полученным именем в текущем каталоге, в случае неудачи в каталогах, указанных в переменных окружения PYTHONPATH и далее в библиотечных каталогах интерпретатора. Полный список каталогов поиска в порядке, соответствующем порядку поиска, хранится в переменной path модуля sys.

```
>>> sys.path
['', '/usr/lib/python2.7',
'/usr/lib/python2.7/plat-i386-linux-gnu',
'/usr/lib/python2.7/lib-tk',
'/usr/lib/python2.7/lib-old',
'/usr/lib/python2.7/lib-dynload',
'/usr/local/lib/python2.7/dist-packages',
'/usr/lib/python2.7/dist-packages',
'/usr/lib/python2.7/dist-packages/PILcompat',
```

```
'/usr/lib/python2.7/dist-packages/gtk-2.0',
'/usr/lib/pymodules/python2.7',
'/usr/lib/python2.7/dist-packages/ubuntu-sso-client']
```

При необходимости данный список может быть модифицирован во время выполнения программы.

Когда имя модуля, который необходимо проимпортировать, заранее неизвестно, может быть использована функция __import__(). Эта функция получает имя модуля в виде строковой константы.

```
>>> def serialize(x):
...     return serializer.dumps(x)
>>> serializer = __import__('pickle')
>>> serialize({'a': 1})
"(dp0\nS'a'\np1\nI1\ns."
>>> serializer = __import__('json')
>>> serialize({'a': 1})
'{"a": 1}'
```

В данном примере при помощи строковой константы, содержащей имя модуля выбирается способ сериализации. Подобное решение возможно в том случае, когда все импортируемые модули имеют схожие интерфейсы.

Еще одним способом импортирования модулей является использование оператора from … import. Этот оператор позволяет скопировать объекты модуля в глобальный контекст. Подобно оператору import, при первом обращении данный оператор запускает код, содержащийся в модуле, и определенные в нем объекты помещаются sys.modules. Однако, в глобальный контекст помещается не весь объект модуля, а только копии объектов, указанные в инструкции from … import.

```
>>> from m1 import a
m1 imported
>>> import sys
>>> 'm1' in sys.modules
True
>>> 'm1' in globals()
False
>>> a
1
```

В данном случае в глобальном пространстве имен будет находится копия переменной a, а оригинальное значение все равно

останется в объекте модуля.

```
>>> import m1
m1 imported
>>> from m1 import a
>>> m1.a
1
>>> a
1
>>> a = 2
>>> m1.a
1
>>> a
2
```

Из примера видно, что переменная модуля и глобальная переменная являются разными значениями. В операторе from … import возможно указать сразу несколько имен для копирования через запятую, а также при помощи ключевого слова as переименовать эти переменные. Переименование может быть удобно в том случае, когда в нескольких импортируемых модулях находятся объекты с одинаковыми именами или импортируемое имя слишком длинное.

```
>>> from m1 import a as m1_a, b as m1_b
>>> m1_a, m1_b
(1, 2)
```

Существует также специальная форма оператора from … import *, которая позволяет скопировать в глобальный контекст сразу все объекты модуля. Обычно такой способ импорта является небезопасным, поскольку имена объектов в импортированном модуле могут совпадать с уже существующими именами в глобальном пространстве имен. В таком случае, имена из модуля подменят глобальные имена, причем это произойдет незаметно для программиста, что в результате может привести к трудно находимым ошибкам.