Программирование на языке Python

Владислав Перлин

часть I

ОГЛАВЛЕНИЕ

I. БАЗОВЫЕ КОНСТРУКЦИИ PYTHON

1. Типы данных

В языке Python имеются около двух десятков встроенных типов данных. Важнейшие из них показаны в таблице I.1.

Таблица I.1: Важнейшие встроенные типы данных

имя	*значение*	*примеры*	
`int`	целое число	`1`, `25`, `-10`	(a)
`long`	большое целое число	`123456789012345678`	(a)
`float`	число с плав. точкой	`1.0`, `3.14159`, `2e5`	
`complex`	комплексное число	`2.0+1.0j`, `3.2-5.1j`	
`bool`	логический	`True`, `False`	
`unicode`	текстовая строка	`u'Hello!'`, `u'Вася Пупкин'`	(b) (c)
`str`	байтовая строка	`'abcdABCD'`	(b) (c)
`tuple`	кортеж	`( 1, 2.0, u'Hello!' )`	(d)
`list`	список	`[ 1.1, u'Текст' ]`	(d)
`dict`	словарь	`{ u'Вася': 1, u'Петя': 2 }`	(d)

(a) Преобразование между обычным целым и длинным целым Python выполняет автоматически — программисту не следует об этом заботиться.

(b) Имена типов `str` и `unicode` в Python 2.*x* сложились исторически. В Python 3.*x* они заменены на `byte` и `str` соответственно.

(c) Теоретически, Python может автоматически преобразовать байтовую строку в текстовую и наоборот. Но полагаться на эту возможность допустимо лишь в особых ситуациях.

(d) Кортежи, списки и словари подробно описаны в соотсетствующем разделе.

(e) Кроме перечисленных в таблице, в языке Python имеются и другие встроенные типы.

(f) Кроме встроенных типов, имеется еще множество стандартных библиотечных типов. Некоторые из их описаны в данном пособии.

2. Присваивание

Команда присваивания в языке Python выглядит традиционно:

```
X = 2*34.0
```

Порядок ее выполнения тоже традиционный — сначала вычисляется выражение, записанное справа от знака присваивания `=`, затем переменная становится равной ссылке на это значение.

Переменная рождается в тот момент, когда ей впервые присвоено значение. Как-либо специально объявлять переменную не требуется.

Все значения делятся на *изменяемые* и *неизменяемые*. Неизменяемыми значениями являются числа, строки и кортежи. Все остальные значения — изменяемые.

Переменная содержит не само значение, а ссылку на значение. Это становится важным, когда мы имеем дело с изменяемым значением.

```
L = [ 1, 2, 3 ] # L --- это ссылка на список
X = L           # X --- ссылка на тот же список, что и L
X[1] = 0
print L
```

Будет выведено:

```
[ 1, 0, 3 ]
```

то есть изменение переменной `X` неявно изменяет значение переменной `L`.

Замечание 1 (Принцип динамической типизации) *Тип значения никогда не изменяется. Переменная изменяет свой тип, подстраиваясь под тип значения.*

```
X = 1        # переменная X имеет тип int.
X = 1.0      # теперь тип переменной X -- float
X = u'Vasya' # А теперь ее тип -- unicode
```

3. Блок

Определение 1. Блок — это несколько команд, которые мы считаем за одну большую команду.

В большинстве языков программирования есть различные пометки, обозначающие начало и конец блока. Например, в языке C это фигурные скобки `{` и `}`, а в языке Pascal — это слова `begin` и `end`.

В языке Python никаких специальных знаков начала и конца блока нет. *Блок выделяется отступом на четыре пробела вправо.*

```
это команда
это другая команда
    это первая команда блока
    это вторая команда блока
    это третья команда блока
эта команда уже не в блоке
```

Блоки может быть вложенными:

```
команда
команда
    внутри блока
    внутри блока
        внутри вложенного
        внутри вложенного
        внутри вложенного
    внутри блока
команда
```

В отличие от большинства других языков программирования, в Python блок *не* может располагаться в любом месте программы: он обязательно является частью управляющей конструкции.

4. Управляющие конструкции

1. Ветвление

```
if A < 0 :
    эти команды
    выполняются,
    если условие ИСТИННО
else :
    эти команды
    выполняются,
```

```
    если условие ложно
эти команды выполняются
независимо от того,
истинно ли условие
```

Секция `else:` может быть опущена:

```
if A < 0 :
    эти команды
    выполняются,
    если условие
    ИСТИННО
эти команды выполняются
независимо от того,
истинно ли условие
```

2. Цикл

Бесконечный цикл

```
while True :
    эти команды
    выполняются
    много-много-много раз
    if A < 0 : break
    эти команды
    выполняются
    много-много-много раз
эти команды
будут выполняться
после сразу же
после команды break
```

Цикл по условию

```
while условие1 :
    эти команды повторяются
    до тех пор, пока условие1
    истинно
    if условие2 : break
    эти команды повторяются
    до тех пор, пока условие1
```

```
    истинно
else:
    эти команды будут выполнены
    один раз, когда условие станет
    ЛОЖНЫМ
    но не будут выполнены,
    если сработает break
эти команды будут выполняться
после окончания цикла
в любом случае
```

Цикл по итератору

```
for It in Seq :
    эти команды повторяются
    до тех пор, пока It
    не переберет все значения,
    содержащиеся в Seq
else:
    эти команды
    будут выполнены
    один раз, когда
    в Seq не останется
    ни одного элемента
эти команды
будут выполняться
после окончания
цикла
```

Цикл по счетчику

```
for K in range(0,N) :
    эти команды повторяются
    до тех пор, пока K
    не переберет все значения,
    от 0 до N-1
else:
    эти команды
    будут выполнены один раз,
    когда K станет равным N
эти команды
будут выполняться
после окончания
цикла
```

II. СПИСКИ, КОРТЕЖИ И СЛОВАРИ

1. Списки

Список представляет собою индексированную последовательность переменной длины, состоящую из значений, одинаковых по смыслу.

Элементы *последовательности* можно перебрать по одному от первого до последнего. В *индексированной* последовательности элементы к тому же имеют еще и индексы (порядковые номера) — то есть имеется возможность выбрать произвольный элемент списка, просто указав его индекс.

```
1 L = [ 1, 12.0, u"Vasya Pupkin", ]
```

После такого присваивания переменная `L` будет содержать список[1] из трех элементов, первый из которых — целое число, второй — вещественное число, третий — текстовая строка.

Чтобы перебрать элементы списка по одному, нужно написать так:

```
1 for el in L :
2     print el
```

В данном случае мы просто выводим элементы списка на консоль. Будет напечатано:

```
1
12.0
Vasya Pupkin
```

Чтобы получить элемент списка по индексу, нужно написать так:

```
1 print L[1]
```

Будет напечатано:

```
12.0
```

Обратите внимание, что индексы начинаются *с нуля*, то есть начальный элемент списка — это `L[0]`, следующий за ним — `L[1]`, далее идет `L[2]` и так далее.

В питоновском списке индексы могут быть отрицательными. Например:

[1] Точнее, *ссылку на список* — подробнее об этом см. стр. 4.

```
print L[-1]
```

Будет напечатано:

```
Vasya Pupkin
```

Срез (slice) списка — это выделенный из списка фрагмент. Срез задается указанием диапазона индексов, которые надо вырезать, причем существует правило: *Начало диапазоана в диапазон входит, а конец — не входит*. То есть:

```
L = [ 1, 2.0, u"Vasya Pupkin", [ 2, 1 ], 5752 ]
print L[1:3]
```

Будет напечатан список из *двух* (а не трех) элементов: `[ 2.0, u"Vasya Pupkin"]`.

(Кстати, обратите внимание, что `L` — это список из *пяти* (а не шести) элементов, причем `L[3]` — тоже список.)

Срез списка можно использовать и слева от знака присваивания, причем в результате длина списка может измениться:

```
L = [ 1, 2.0, u"Vasya Pupkin", [ 2, 1 ], 5752 ]
L[1:3] = [ 9, 8, 7 ]
print L
```

Будет напечатано

```
[ 1, 9, 8, 7, [ 2, 1 ], 5752 ]
```

Первый или последний индекс среза можно опустить. Это будет означать «с начала списка» и «до конца списка», соответственно:

Написано	Результат
`L[2:]`	`[ u"Vasya Pupkin", [ 2, 1 ], 5752 ]`
`L[:2]`	`[ 1, 2.0 ]`
`L[-2:]`	`[ [ 2, 1 ], 5752 ]`
`L[:-2]`	`[ 1, 2.0, u"Vasya Pupkin" ]`
`L[:]`	`[ 1, 2.0, u"Vasya Pupkin", [ 2, 1 ], 5752 ]`

Обратите внимание на отличие `L[:]` от `L`. Первая запись дает нам *копию* списка, а вторая — *ссылку* на список.

Удалить элемент из списка можно при помощи команды `del`

```
L = [ 1, 2.0, u"Vasya Pupkin", [ 2, 1 ], 5752 ]
del L[3]
print L
```

Будет выведено

```
[ 1, 2.0, [ 2, 1 ], 5752 ]
```

Операции сцепления и повтора списка

Написано	Результат
`[ 1, 2 ] + [ 3, 4 ]`	`[ 1, 2, 3, 4 ]`
`[ 1, 2 ] * 3`	`[ 1, 2, 1, 2, 1, 2 ]`
`4*[ 1, 2 ]`	`[ 1, 2, 1, 2, 1, 2, 1, 2 ]`

2. Кортежи

Кортеж — это *неизменяемый* список. То есть, с кортежем можно делать все те же операции, что и со списком, за исключением тех, которые изменяют кортеж. Например:

```
L = [ 1, 2.0, u"Vasya Pupkin", [ 2, 1 ], 5752 ] # список
C = ( 1, 2.0, u"Vasya Pupkin", [ 2, 1 ], 5752 ) # кортеж

print L[1] # нормально
print C[1] # нормально

print L[1:3] # нормально
print C[1:3] # нормально

L[1] = 0 # нормельно
C[1] = 0 # ОШИБКА!!! Нельзя изменять элемент кортежа

L[2:3] = [ 1, 2, 3 ] # нормально
C[2:3] = ( 1, 2, 3 ) # ОШИБКА!!! Нельзя изменять срез кортежа
```

Кроме того, есть еще операция, которая, хотя и применима для списков, предназначена в первую очередь для кортежей. Это операция *разборки кортежа*

Операция разборки кортежа позволяет разобрать значения элементов кортежа по отдельным переменным.

```
name = ( u'Иван', u'Иванович', u'Иванов' )
( firstname, middlename, lastname ) = name
```

Переменным `firstname`, `middlename` и `lastname` будут присвоены значения `u'Иван'`, `u'Иванович'` и `u'Иванов'` соответственно.

При этом количество переменных слева должно быть равно количеству элементов в кортеже:

```
name = ( u'Иван', u'Иванович', u'Иванов' )
( firstname, middlename ) = name                    # Ошибка!!!
( firstname, middlename, lastname, year ) = name  # Ошибка!!!
```

Операцию раборки кортежа можно использовать для обмена значений переменных.

```
a = 5
b = u'Hello'
( a, b ) = ( b, a )
```

Теперь переменная `a` равна `u'Hello'`, а переменная `b` равна `5`.

3. Словари

Словарем называется совокупность пар «ключ–значение», причем каждое значение определяется своим ключом. В роли ключа теоретически может выступать любое неизменяемое значение, но как правило в роли ключа выступает либо целое число, либо строка (текстовая либо байтовая). Значением может быть любое допустимое значение (точнее — ссылка на значение).

Операции со словарями очень похожи на операции со списками, за исключением того, что элементы словаря (в отличие от элемента списка) не упорядочены, и поэтому у словарей не бывает срезов, а также того, что при присваивании значения какому-нибудь ключу этот ключ автоматически создается в словаре, если его там до сих пор не было.

```
D = {
    u'Иванов'  : 1989 ,
    u'Петров'  : 1970 ,
    u'Сидоров' : 1969
}

print D[u'Иванов']    # будет выведено 1967
```

```
del D[u’Иванов’]      # удаляем ключ и соответствующее ему значение
D[u’Петров’] = 1971   # значение ключа u’Петров’ изменилось
D[u’Антонов’] = 1968  # создан новый ключ

print D[u’Алекс’]             # Ошибка!!! Такого ключа нет в словаре.
print D.get( u’Алекс’, u’?’ ) # Нормально. Будет выведен знак вопроса

for key in D :        # Перебираем все ключи в словаре
    print key

for ( key, val ) in D.items() : # перебираем попарно все ключи
    print key, val              # и соответствующие им значения

for val in D.values() :         # перебираем только значения
    print val                   # без ключей
```

III. ОБРАБОТКА ТЕКСТОВОЙ ИНФОРМАЦИИ

В языке Python имеются две разновидности строк: *текстовые* и *байтовые*. В Python 2.*x* текстовые строки имеют тип `unicode`, а байтовые — тип `str`.

Замечание 2 *Для представления текстовой информации следует пользоваться текстовыми строками.*

Существуют следующие способы записи текстовой строки:

```
A = u'Это текстовая строка'
B = u"Это тоже текстовая строка"

C = u'''
    Это очень длинная
    текстовая строка
'''

D = u"""
    Это тоже очень длинная
    текстовая строка
"""
```

Байтовые строки записываются аналогично, но без буквы `u` в начале.

```
A = 'Это БАЙТОВАЯ строка'
B = "Это тоже БАЙТОВАЯ строка"

C = '''
    Это очень длинная
    байтовая строка
'''

D = """
    Это тоже очень длинная
    байтовая строка
"""
```

В Python 2.*x* разрешается в особых случаях смешивать текстовые и байтовые строки, однако без особой необходимости этого лучше не делать.

Строка (как текстовая, так и байтовая) является неизменяемым объектом.

Строка (как текстовая, так и байтовая) является последовательностью, поэтому с ней можно выполнять все те же операции, что и с любой последовательностью.

программа

```
Str = u'Привет'

print Str[1:3]
print Str[:-1]

for letter in Str :
  print letter
```

напечатает

```
ри
Приве
П
р
и
в
е
т
```

1. Основные операции со строками

Сцепление строк

программа

```
print u'Вася' + u'Пупкин'
```

напечатает

```
ВасяПупкин
```

Обратите внимание на отсутствие пробела между `u'Вася'` и `u'Пупкин'` — этого пробела не было ни в одной из исходных строк.

Разбиение строки

программа

```
Str = u'Вася,Петя,Маша'
print Str.split(u',')
```

напечатает

```
[ u'Вася', u'Петя', u'Маша' ]
```

Длина строки

программа

```
print len( u'Привет!' )
```

напечатает

```
7
```

Поиск подстроки

программа

```
Str = u'Привет, Вася Пупкин!'
print u'Вася' in Str
print Str.find( u'Вася' )
```

напечатает

```
True
7
```

Функция `find()` возвращает позицию, начиная с которой подстрока, заданная в качестве параметра, входит в строку (в данном случае `Str`). Если подстрока не найдена, возвращает `-1`.

Операция `in` просто проверяет наличие подстроки в строке и возвращает `True` или `False` соответственно.

Обрасывание пробелов

программа

```
Str = u'     Привет!     '
print u'[' + Str.strip() + u']'
```

напечатает

```
[Привет!]
```

2. Форматирование строки

Кроме того, для строк существует операция *форматирования*. Она записывается одним из двух способов.

```
# форматирование с кортежем
Str = u'Сегодня %d день %s месяца'
print Str % ( 3, u'февраля' )

# форматирование со словарем
Str = u'Сегодня %(day)d день %(month)s месяца'
print Str % {
             'day'   : 3,
             'month' : u'февраля'
            }
```

Оба варианта выведут одно и то же:

```
Сегодня 3 день февраля месяца
```

3. Кодеки и кодировки

Кодек — это объект операционной системы, который содержит правила преобразования текстовой строки в байтовую и обратно. Для управления кодеками и преобразования текстовой строки в байтовую и обратно служит стандартный модуль `codecs`.

При этом текстовая строка рассматривается как своего рода «исходный вид» текста. Соответственно, преобразование текстовой строки в байтовую считается ее *за*кодированием, а байтовой строки в текстовую — *рас*кодированием.

```
import codecs

Utf8    = codecs.lookup( 'utf-8'        ) # запрашиваем кодек UTF-8
Win1251 = codecs.lookup( 'windows-1251' ) # запрашиваем кодек Win-1251
```

Полученный объект-кодек теперь можно использовать для закодирования и раскодирования строки.

```
Text = u'Привет'  # текстовая строка
( Bytes, Count ) = Utf-8.encode( Text )
```

Метод `encode()` принимает единственный параметр — текстовую строку, которую надо закодировать — и возвращает кортеж, первым элементом которого будет искомая байтовая строка, а вторым — целое число, обозначающее количество обработанных символов в исходной строке.

Аналогично выполняется раскодирование.

```
Bytes = 'Привет'  # байтовая строка
( Text, Count ) = Utf-8.encode( Text )
```

В этом случае переменная `Count` содержит количество байт, которые были обработаны в исходной байтовой строке.

4. Регулярные выражения

Регулярные выражения служат для более «интеллектуального» анализа текстовых строк, чем простое сравнение.

Схема использования регулярных выражений в целом такова:

a) создаем регулярное выражение;
b) сопоставляем регулярное выражение с текстовой строкой;
c) анализируем результат.

Таблица III.1: Основные специальные символы в регулярных выражениях

`.`	(точка) любой символ, кроме символа конца строки.
`[]`	любой символ из тех, что заключены в скобки; символы можно перечислить как напрямую `[abcdef]`, так и через дефис `[a-f]`
`^`	(крышечка) начало строки
`$`	конец строки
`*`	предыдущий символ или фрагмент повторяется 0 или более раз
`+`	предыдущий символ или фрагмент повторяется 1 или более раз
`?`	предыдущий символ или фрагмент может как присутствовать, так и отсутствовать
`{m}`	предыдущий символ или фрагмент повторяется ровно m раз
`{m,n}`	предыдущий символ или фрагмент повторяется от m до n раз
`\`	экранирует специальный символ (что позволяет использовать в регулярном выражении обычные точки, звездочки и проч.), или обозначает специальную последовательность
`()`	выделяет фрагмент (группу) в регулярном выражении
`(A|B)`	фрагмент (группа), соответствующий либо регулярному выражению `A`, либо регулярному выражению `B`
`\b`	пустая строка, обозначающая начало или конец слова
`\d`	любая цифра
`\s`	любой символ помежутка (пробел, знак табуляции, символ конца строки, символ возврата каретки)
`\S`	любой символ, *кроме* символа промежутка
`\w`	буква, цифра или знак подчеркивания

Таблица III.2: Важнейшие методы объекта соответствия

`group(No)`	Если `No` равен нулю, возвращает весь найденный фрагмент сроки целиком. Если это положительное число — возвращает фрагмент, выделенный круглыми скобками.
`start(No)`	Начало найденного фрагмента.
`end(No)`	Конец найденного фрагмента.

При необходимости пункты b–c повторяем несколько раз.

За регулярные выражения в языке Python отвечает модуль `re`.

программа

```
import re

RE.compile( ur'кошка' )

Str1 = u'Сиамская кошка мяукает'
Str2 = u'Сиамский кот мяукает'

M = RE.seacrh( Str1 )
if M :
    print u'Соответствие найдено'
else :
    print u'Нет соответствия'

M = RE.search( Str2 )
if M :
    print u'Соответствие найдено'
else :
    print u'Нет соответствия'
```

напечатает

```
Соответствие найдено
Нет соответствия
```

Функция `search()` проверяет текстовую строку, заданную в качестве параметра, на предмет *соответствия* сконструированному заранее регулярному выражению.

Строка считается соответствующей регулярному выражению, если это выражение встречается где-нибудь *внутри* этой строки. При этом некоторые символы имеют особый смысл (см. таблицу III.1). Если соответствие найдено, функция `search()` возвращает специальный *объект соответствия*, в противном случае она возвращает `None`.

Важнейшие методы объекта соответствия приведены в таблице III.2.

IV. ОБРАБОТКА ИСКЛЮЧЕНИЙ

Исключением называется особое значение, появление которого прерывает нормальное исполнение программы и передает управление на *блок обработки исключений*. Если подходящего блока не нашлось, программа завершается с соответствующим сообщением об ошибке.

Исключение может быть выброшено:

- интерпретатором Python при возникновении ошибки или иной исключительной ситуации;
- библиотечной функцией;
- командой `raise`.

Конструкция обработки исключений выглядит следующим образом.

```
try:
    блок, в котором
    мы ожидаем
    выброса исключения
finally:
    этот блок выполняется
    независимо от того,
    выброшено исключение
    или нет
except IndexError :
    этот блок выполняется,
    если было выброшено
    исключение типа IndexError
except ZeroDivisionError :
    этот блок выполняется,
    если было выброшено
    исключение типа IndexError
else:
    этот блок выполняется,
    если никакого исключения
    выброшено не было
```

Если никакого исключения выброшено не было, `try`-блок выполняется до конца, затем выполняется блок, помеченный меткой `finally` и затем — `else`-блок. Блоки, помеченные метками `except`, выполняться не будут.

Обработчиком исключения называется блок, помеченный меткой `except`. При этом указывается *тип* того исключения, которое будет обрабатываться этим блоком.

Если при выполнении `try`-блока было выброшено исключение, то выполнение `try`-блока прекращается на той команде, где оно было выброшено. Затем выполняется блок, помеченный меткой `except` с указанием типа выброшенного исключения. Если такого блока не нашлось — исключение не будет перехвачено в этом месте (но может быть перехвачено в другом месте — например, если вся эта конструкция находится внутри другого `try`-блока).

Например.

```
1  import math
2  try :
3      S = input( u'Площадь круга: ' )
4      R = math.sqrt ( S / math.pi )
5      A = 10.0 / R
6      print u'R =', R
7      print u'A =', A
8  finally:
9      print u'Расчет окончен'
10 except ValueError :
11     print u'Площадь круга не может быть отрицательной'
12 except ZeroDivisionError :
13     print u'Попытка деления на нуль'
14 else :
15     print u'Расчет окончен успешно'
```

Если пользователь в ответ на запрос введет $S = 5$, то будет выведено:

```
1.26156626101
7.92665459521
Расчет окончен
Расчет окончен успешно
```

Если пользователь введет $S = -1$, то будет выведено:

```
Расчет окончен
Площадь круга не может быть отрицательной
```

Обратите внимание, что строки 5–7 выполнены не будут.

Если пользователь введет $S = 0$, то будет выведено:

```
Расчет окончен
Попытка деления на нуль
```

Таблица IV.1: Важнейшие стандартные типы исключений

тип	когда выбрасывается
`Exception`	тип, общий для всех исключений
`ZeroDivisionError`	попытка деления на нуль
`ValueError`	попытка выполнить операцию, которую нельзя выполнить с этим значением, например, извлечь квадратный корень из отрицательного числа
`IndexError`	обращение к несуществующему элементу списка
`KeyError`	обращение к несуществующему элементу словаря
`ImportError`	ошибка импорта модуля
`AttributeError`	обращение к несуществующему атрибуту объекта
`KeyboardInterrupt`	пользователь нажал `Ctrl-C` на клавиатуре
`IndentationError`	неправильная расстановка отступов
`UnicodeError`	ошибка перекодирования текста
`TypeError`	недопустимый тип значения
`IndentationError`	неправильная расстановка отступов
`IOError`	ошибка ввода-вывода

В строке 4 приведенного примера исключение выбрасывается (если выбрасывается) библиотечной функцией `math.sqrt`. В строке 5 — интерпретатором. Кроме того, можно выбросить исключение определенного типа явно командой `raise`:

```
1 raise ValueError, u'Текст сообщения об ошибке'
```

V. ФУНКЦИИ

Функция в языке Python создается командой `def`. При этой указывается *имя функции*, а также список *формальных параметров*.

```
def NOD( A, B ) :
    '''
        Вычисление наибольшего общего чисел A и B
    '''
    if A < B :
        ( A, B ) = ( B, A )
    ( A, B )  = ( B, A % B )
    while B > 0 :
        ( A, B )  = ( B, A % B )
    return A
```

Использоваться эта функция может, например, так:

```
X = NOD( 25, 15 )
print X
```

Будет выведено

```
5
```

Фомальный параметр — это обыкновенная переменная, которая создается в момент вызова функции, и которой присваивается значение, указанное при вызове. В данном случае переменной `A` будет присвоено значение `25`, а переменной `B` — значение `15`.

Команда `return` выполняет *возврат* из функции в то место, откуда функция была вызвана и, кроме того, задает *возвращаемое значение* функции. Именно это значение будет использовано в точке вызова (в данном случае — присвоено переменной `X`).

Обратите внимание, что — как и вообще в языке Python — и формальные параметры, и возвращаемое значение содержат не само значение, а ссылку на значение, что может привести к любопытным эффектам.

Формальному параметру функции можно указать *значение по умолчанию*. В таком случае при вызове функции это значение можно будет не задавать — тогда будет использовано значение по умолчанию.

Строка, указанная сразу после заголовка функции — это *строка документации*. К этой строке можно получить доступ в самой программе, но как правило

она используется по прямому назначению — на ее основе формируется текстовый документ, содержащий описание функции (для этого есть специальные программные пакеты).

```
def connect( dbname, host=u'localhost' ) :
    '''
        Соединение с базой данных
    '''
    # тут будут команды
    # реализующие соединение
    return Connection
```

Функция `connect` осуществляет соединение с базой данных и возвращает дескриптор этого соединения. При этом имя базы данных `dbname` нужно указать обязательно, а имя сервера `host` можно не указывать. Если его все же указать — функция попытается соединиться с базой данных, расположенной на указанном сервере, а если не указать — она будет считать, что ей указано имя `u'localhost'` и попытается соединиться с базой на локальном компьютере.

```
Conn1 = connect( u'primary', u'center' ) # соединяемся с БД 'primary'
                                         # на сервере 'center'
Conn2 = connect( u'secondary' ) # соединяемся с базой 'secondary'
                                # на сервере 'localhost'
Conn3 = connect()          # ОШИБКА!!! имя базы данных обязательно
                           # должно быть указано
```

Параметры можно передавать с `ключевыми словами`. Тогда параметры могут следовать в иной последовательности, чем в объявлении функции. Например, функцию `connect` из предыдущего примера можно вызвать так:

```
Conn4 = connect ( dbname = u'primary', host = u'main'   )
Conn5 = connect ( host = u'center', dbname = u'primary' )
```

1. Функция с переменным числом параметров

Функцию можно создать таким образом, что ее можно будет вызывать с произвольными параметрами.

```
def connect ( dbname, host=u'localhost', *args, **kwargs ) :
    # тут будут команды
    # реализующие соединение
    return Connection
```

При вызове этой функции первый параметр (или параметр с ключевым словом `dbname`) будет присвоен формальному параметру `dbname`, второй параметр (или параметр с ключевым словом `host`) будет присвоен формальному параметру `host`.

Все остальные параметры *без ключевых слов* будут собраны в кортеж `args`, а все остальные параметры *с ключевыми словами* — в словарь `kwargs`.

Например, если функцию `connect` вызвать так:

```
Conn = connect( u'one', u'main', 12, 'auto',
                autocommit = True, timeout = 30 )
```

то кортеж `args` стенет равным `( 12, 'auto' )`, а словарь `kwargs` — равным `{ 'autocommit' : True, 'timeout' : 30 }`.

2. Функция как объект первого класса

Имя функции в языке Python на самом деле представляет собой обыкновенную перемененную, которая — согласно общему правилу — содержит *ссылку на функцию*. Соответственно, с этой переменной можно обращаться так же, как и с любой другой.

```
def func1 ( a, b, c ) : # создаем функцию
    команды,
    составляющие
    функцию.

func2 = func1        # теперь func2 содержит ссылку
                     # на ту же функцию, что и func1

func1 ( 1, 2, 3 )    # и их можно вызывать
func2 ( 1, 2, 3 )    # одинаковым способом

func1 = 5            # теперь func1 --- это больше не функция,
A = 2 + func1        # а обычное чисто (A стало равным 7)

func2( 4, 5, 6 )     # но функцию по-прежнему можно вызвать
                     # через переменную func2
```

Функцию можно также передать в качестве параметра в другую функцию.

```
def primary ( a, b ) : # создаем функцию
    команды,
    составляющие
    функцию

def secondary ( n, function ) : # второй параметр этой функции --
    function( 1, n )            # тоже функция, ее можно вызвать

secondary( 3, primary )   # а эту функцию вызываем так,
                          # передав функцию, как параметр
```

Иногда бывает так, что функция-параметр очень примитивна и сводится к простому выражению. Например, так:

```
def summa ( a, b ) :
    return a + b

secondary( 1, summa )
```

Такую простую функцию не обязательно создавать отдельно. В таком случае можно воспользоваться λ-функцией.[1]

```
secondary( 1, lambda a, b : a + b )
```

Функцию можно создать внутри другой функции и вернуть в качестве возвращаемого значения.

```
def general ( a ) :
    def function ( b ) :
        return a + b
    return function
```

Функция `general` создает внутри себя функцию и возвращает ее. Она будет создавать новую функцию всякий раз, когда она будет вызываться.

```
func1 = general(  1 )     # создаем одну функцию
func2 = general( 10 )     # создаем другую функцию

print func1( 1 )
print func2( 1 )
```

Будет выведено

[1] Читается «лямбда-функция».

```
2
11
```

3. Декораторы

Декоратором называется функция, которая имеет единственный параметр — другую функцию, и возвращает тоже функцию.

Допустим, у нас есть функция, и нам необходимо убедиться, что первый ее параметр имеет тип `int`. Это можно было бы сделать так:

```
1  def funcone ( a, b, c ) :
2      if not isinstance ( a, int ) :
3          raise TypeError
4      # полезная
5      # часть
6      # функции
```

Однако этот способ слишком неудобен — если такая же проверка потребуется в другой функции, команды в строках 2–3 придется повторять и там. Вместо этого можно поступить так:

```
1  def first_is_int ( function ) :
2      def checked_func ( *args, **kwargs ) :
3          if isinstance( args[0], int ) :
4              raise TypeError
5          return function ( *args, **kwargs )
6      return checked_func
7
8  def func1 ( a, b, c ) :
9      # полезная
10     # часть
11     # функции
12
13 func1 = first_is_int ( func1 )
```

То есть, мы написали функцию, которая создает функцию-обертку (wrapper function) для той функции, которая передана ей в качестве параметра. Затем мы создаем нужную нам функцию, а затем — «обертываем» ее. Полученная обертка сначала роверяет правильность первого параметра, а затем — либо выбрасывает исключение, либо вызывает нашу функцию.

Для такого случая — применения обертки — существует более удобная форма записи:

```
@first_is_int
def func1 ( a, b, c ) :
    # полезная
    # часть
    # функции
```

Таким способом можо обернуть — декорировать — столько функций, сколько нам нужно.

```
@first_is_int
def func1 ( a, b, c ) :
    # полезная
    # часть
    # функции

@first_is_int
def func2 ( a ) :
    # полезная
    # часть
    # функции
```

Обратите внимание, что для каждой из наших функций создается своя оберт-ка.

VI. ФАЙЛЫ И ФАЙЛОВАЯ СИСТЕМА

Открытие файла выполняется функцией `open()`.

```
1 SRC = open( u'myfile.txt', 'rt' )
```

Первый параметр функции `open` — это текстовая строка, задающая путь к файлу, который требуется открыть. Это может быть относительный или абсолютный путь — по общим правилам, принятым в операционной системе.

Второй параметр — это байтовая[1] строка, состоящая из двух букв, и задающая *режим открытия* файла.

Первая буква — это либо буква `'r'` (и тогда файл будет открыт для чтения), либо буква `'w'` или `'o'` (и тогда файл будет открыт на запись).

При попытке открыть для чтения несуществующий файл будет выброшено исключение `IOError`.

При открытии несуществующего файла на запись (безразлично, в режиме `'w'` или `'a'`) будет сделана попытка создать этот файл. Если создать файл не получится, будет выброшено исключение `IOError`.

Различие между режимами `'w'` и `'a'` проявляется при попытке открыть на запись *существующий* файл. В режиме `'w'` файл будет удален и создан заново. В режиме `'a'` он будет открыт таким образом, что новые данные будут дописываться в конец файла.

Вторая буква второго параметра — это либо буква `'t'` (и тогда файл будет открыт как текстовый), либо буква `'b'` (и тогда файл будет открыт как двоичный).

В случае, если файл открыт успешно, функция `open())` возвратит *дескриптор* открытого файла.

После окончания работы с файлом он должен быть **закрыт**. Для этого служит функция `close()`:

```
1 SRC.close()
```

1. Ввод и вывод

Замечание 3 *Все операции ввода и вывода работают только с* байтовыми *(а не с текстовыми) строками.*

[1] Указывать здесь именно байтовую строку — наследие стандарта POSIX. Однако, указать текстовую строку также не будет ошибкой.

1. Чтение из файла

Текстовый файл представляет собой *последовательность строк*. Обратите внимание, что — так как кодировка текстового файла нам заранее не известна — это будет последовательность *байтовых* строк. Строки отделяются друг от друга символом конца строки `'\n'`, который также считается частью строки (автоматически он не отбрасывается). Удалить этот символ можно простой командой:

```
if L[-1] == '\n' : L = L[:-1]
```

Просто (без проверки) удалять последний символ строки рискованно, потому что последняя строка файла может заканчиваться и другим символом.

Таким образом, стандартная схема чтения текстового файла будет такой:

```
import codecs

Codec = codecs.lookup( 'utf-8' )  # загружаем нужный кодек

SRC = open ( u'myfile.txt', 'rt' )
for Line in SRC :
    if Line[-1] == \n' : Line = Line[:-1]
    Line = Codec.decode( Line )[0]
    # Теперь обрабатываем
    # полученную текстовую строку
SRC.close()
```

Чтение **двоичного файла** выполняется функцией `read()`[2] Эта функция может использоваться в двух вариантах: без параметра для чтения файла целиком (что не рекомендуется) или с параметром для чтения файла по частям. При достижении конца файла функция `read()` возвращает `None`.

Стандартная схема чтения двоичного файла получится такой:

```
SRC = open ( u'myfile.dat', 'rb' )
while Data = SRC.read(200) :  # читаем фрагментами по 200 байт
    # Здесь обрабатывем
    # полученные данные
SRC.close()
```

Обратите внимание, что в заголовке цикла употреблена команда присваивания, а не сравнения — в данном случае это не ошибка.

[2]На самом деле эту функцию можно использовать и для текстового файла.

Заметим, что такое «лобовое» чтение двоичного файла на практике применяется редко. В большинстве случаев для каждого конкретного двоичного формата данных используется соответсвующая библиотека.

2. Запись в файл

Запись в файл (безразлично, текстовый или двоичный) выполняется функцией `write()`, которая принимает единственный параметр — байтовую строку, которую необходимо записать в файл. При записи в текстовый файл эта строка обычно получена перекодированием текстовой строки в соответствующую кодировку.

```
1 TRG = open( u'myfile.file', 'wb' )
2 TRG.write( Data )
3 TRG.close()
```

2. Основные операции с путями к файлам

Путь к файлу задается текстовой строкой. Python автоматически определяет екущую кодировку файловой системы и соответственно преобразует текстовую строку к байтовой при необходимости.

В принципе, с путями к файлам можно работать и как с обычной текстовой строкой. Но этого делать не рекомендуется: в стандартной библиотеке имеется стандартный модуль `os.path`, который выполняет ряд операций с путями, учитывая при этом особенности операционной системы (например, различие между разделителями `\` и `/`).

Важнейшие функции модуля `os.path` перечислены далее.

`normpath(path)`
: Возвращает «нормализованный» путь к файлу. В нормализованном пути во-первых, разделители папок исправлены на правильные (для данной операционной системы), а во-вторых, удалены «циклические» пути — например, путь `A/foo/../B` превращается в `A/B`.

`abspath(path)`
: Преобразует относительный путь в абсолютный, считая относительный путь от текущей рабочей папки.

`relpath(path)`
: Преобразует абсолютный путь в относильный, считая последний от текущей рабочей папки.

exists(path)
: Возвращает `True`, если указанный файл или папка существует.

split(path)
: Возвращает кортеж, первый элемент которого — путь к папке, а второй — имя файла.

splitdrive(path)
: Возвращает кортеж, первый элемент которого — буква устройства, а второй — путь к файлу внутри устройства.

splitext(path)
: Возвращает кортеж, второй элемент которого — расширение файла, а первый — все остальное.

join(path1,path2,path3,…)
: «Склеивает» путь к файлу из заданных частей, вставляя между ними символ-разделитель папок, принятый в данной операционной системе.

3. Рекурсивный обход папки

Иногда возникает необходимость рекурсивно перебрать все файлы и папки, находящиеся внутри заданной папки. Для этого служит генератор-функция `os.walk()`. Эта функция принимает параметр — путь к папке, которую необходимо просмотреть. На каждом шаге итерации функция возвращает кортеж `(path,dirs,files)` из трех элементов:

- `path` — пусть к папке, в которую мы только что зашли;
- `dirs` — список папок, непосредственно содержащихся в папке `path` (но не в ее подпапках);
- `files` — список файлов, непосредственно содержащихся в папке `path`, но не в ее подпапках.

Например, чтобы вывести полный список файлов и папок, содержащихся в корне диска `C:`, можно так:

```
import os
import os.path

for ( path, dirs, files ) in os.walk( u'C:\' ) :
    print path
    for file in files :
        print os.path.join( path, file )
```

Обратите внимание, что, как и вообще в языке Python списки `dirs` и `files` — это не копии списков, а ссылки на списки, причем сами списки — это внутренние переменные функции `os.walk()`. Этим фактом можно воспользоваться, чтобы заставить ее пропустить при просмотре ту или иную папку. Для этого пропускаемую папку нужно просто удалить из списка `dirs`.

Например, чтобы пропустить при просмотре папку `Program Files`, можно поступить так:

```
import os
import os.path

for ( path, dirs, files ) in os.walk( u'C:\' ) :
    print path
    for file in files :
        print os.path.join( path, file )
    if path == u'C:\' and u'Program Files' in dirs :
        dirs.remove( u'Program Files' )
```

VII. ПАКЕТЫ И МОДУЛИ

Чтобы создать модуль, нужно создать обычный файл с кодом на язяке Python, поместив в этот файл нужные функции. Например, в файл `mod1.py` запишем:

mod1.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

def func1 ( x ) :
    print u'Функция вызвана с параметром', x
```

Затем, вызовем эту функцию из файла `prog.py`. Это делается так:

prog.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# импортируем модуль
import mod1

# вызываем функцию из модуля
mod1.func1 ( 5 )
```

Чтобы собрать несколько модулей в пакет, нужно:

- сохранить эти модули в папку;
- создать в этой папке файл `__init__.py`;
- в файле `__init__.py` импортировать все нужные модули.

Например, пусть у нас кроме модуля `mod1.py` есть еще модуль `mod2.py`:

mod2.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

def func2 ( x ) :
    print u'Другая функция вызвана с параметром', x
```

Переместим оба этих модуля в папку `pack`. В этой папке создадим файл `__init__.py`, в котором напишем:

pack/__init__.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from mod1 import func1
from mod2 import func2
```

И вызовем эти функции подобно тому, как вызывали функцию из модуля:

prog.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# импортируем пакет
import pack

# вызываем первую функцию из пакета
pack.func1 ( 5 )

# вызываем вторую функцию из пакета
pack.func2 ( 100 )
```

1. Важнейшие стандартные пакеты

re — обработка регулярных выражений.

codecs — кодеки и кодировки, преобразование из текстовой строки в байтовую и обратно.

datetime — работа с датой и временем.

weakref — «слабые» ссылки.

math — математические операции с вещественными числами.

cmath — математические операции с комплексными числами.

decimal — арифметические операции с фиксированной точкой.

random — генерация псевдослучайных чисел.

os.path — работа с путями к файлам.

sqlite3 — интерфейс к базе данных SQLite.

zipfile — работа с ZIP-архивами.

csv — работа с файлом в формате CSV (текст, разделенный запятыми).

hashlib — вычисление контрольных сумм.

os — различные системные операции.

argparse — разбор параметров командной строки.

logging — ведение журнала приложения.

subprocess — запуск одной программы из другой (дочерний процесс).

htmllib — синтаксический разбор HTML-страниц.

xml.dom и **xml.sax** — синтаксический разбор XML-документов.

httplib — клиент протокола HTTP.

SimpleHTTPServer — простой сервер протокола HTTP.

sys — доступ к параметрам и функциям операционной системы.

1. Некоторые важные нестандартные пакеты

psycopg2 — интерфейс к базе данных PostgreSQL.

numpy и **scipy** — пакеты для научных вычислений.

matplotlib — пакет научной графики.

PyQt4 и **PyCore** — два различных интерфейса к GUI-библиотеке Qt4.

pygame — пакет для создания простых компьютерных игр.

pyogre — интерфейс к пакету трехмерной визуализации OGRE.

cv — интерфейс к библиотеке компьютерного зрения OpenCV.

VIII. ВЕДЕНИЕ ЖУРНАЛА СОБЫТИЙ ПРИЛОЖЕНИЯ

Журнал событий приложения обычно ведут в виде стандартного текстового файла. Для этой цели служит библиотечный модуль `logging`.

При ведении журнала различают пять степеней серьезности:

- `DEBUG` (отладочное сообщение) — подробная информация, обыкновенно интересная только разработчику программы для поиска возможных ошибок.
- `INFO` (уведомление) — сообщение о событии, произошедшем в системе именно так, как предполагалось. Может применяться для фиксации учетных транзакций или ведения журнала действий пользователя.
- `WARNING` (предупреждение) — сообщение о том, что произошло что-то непредвиденное, или что не исключены какие-то проблемы в близком будущем (например, осталочь мало места на диске), но программа все еще штатно выполняет сови функции.
- `ERROR` (ошибка) — возникла проблема, в результате которой программа не может выполнить ту или иную операцию.
- `CRITICAL` (фатальная ошибка) — возникла проблема столь серьезная, что программа вообще не может дальше работать.

Каждому сообщению при отправке в журнал приписывается определенный *уровень сообщения*. Кроме того, всему журналу в целом устанавливается *уровень журналирования*. При этому в журнал реально записываются только сообщения заданной степени серьезности или выше. По умолчанию установлен уровень журналирования `WARNING`.

Отправка сообщений в журнал приложения выполняется следующим образом:

```
1 import logging
2
3 logging.debug( u'Это отладочное сообщение' )
4 logging.info( u'Информационное сообщение' )
5 logging.warning( u'Предупреждение' )
6 logging.error ( u'Произошла ошибка' )
7 logging.critical( u'Фатальная ошибка' )
```

При этом реально в журнал приложения попадут сообщения, соответствующие установленному уровню журналирования (по умолчанию — третье, четвертое и пятое).

Настройка журналирования выполняется функцией `logging.basicConfig()`. Эту функцию следует вызывать только один раз на протяжении работы программы. Все последующие ее вызовы будут проигнороированы.

Функция `basicConfig()` имеет следующие параметры:

- `level` — уровень журналирования;
- `filename` — путь к файлу, в который пишется журнал;
- `format` — формат записи в в журнал приложения.

Возможные значения параметра `level` соответствуют степеням серьезности сообщений журнала.

Значение параметра `format` — это текстовая строка в такой форме, акая предусмотрена для форматирования строк со словарем (см. §III.2, стр. 15). В параметре `format` возможны следующие ключи форматирования:

- `%(asctime)s` — дата и время, когда была сделана запись в журнал в форме, удобной для чтения человеком;
- `%(filename)s` — файл, из которого была сделана запись в журнал;
- `%(funcName)s` — имя функции, из которой была сделана запись в журнал;
- `%(levelname)s` — уровень серьезности сообщения;
- `%(module)s` — имя модуля, из которого была сделана запись;
- `%(message)s` — текст сообщения в журнал приложения.