Разработка научных приложений
Введение в Python

1 Общее описание языка

Python — интерпретируемый, объектно–ориентированный высокоуровневый язык программирования с динамической семантикой. Встроенные высокоуровневые структуры данных в сочетании с динамическими типизацией и связыванием делают язык привлекательным для быстрой разработки приложений. Кроме того, его можно использовать в качестве сценарного языка для связи программных компонентов. Синтаксис Python прост в изучении, в нем придается особое значение читаемости кода, а это сокращает затраты на сопровождение программных продуктов. Python поддерживает модули и пакеты, поощряя модульность и повторное использование кода. Интерпретатор Python и большая стандартная библиотека доступны бесплатно в виде исходных и исполняемых кодов для всех основных платформ и могут свободно распространяться.

Python — это универсальный язык программирования. Он имеет свои преимущества и недостатки, а также сферы применения. В поставку Python входит обширная стандартная библиотека для решения широкого круга задач. В Интернете доступны качественные библиотеки для Python по различным предметным областям: средства обработки текстов и технологии Интернет, обработка изображений, инструменты для создания приложений, механизмы доступа к базам данных, пакеты для научных вычислений, библиотеки построения графического интерфейса и т.п. Кроме того, Python имеет достаточно простые средства для интеграции с языками C, C++ (и Java) как путем встраивания (embedding) интерпретатора в программы на этих языках, так и наоборот, посредством использования библиотек, написанных на этих языках, в Python–программах. Язык Python поддерживает несколько парадигм программирования: **императивное** (процедурный, структурный, модульный подходы), **объектно–ориентированное** и **функциональное** программирование.

Можно считать, что Python — это целая технология для создания программных продуктов (и их прототипов). Она доступна почти на всех современных платформах (как 32-битных, так и на 64-битных) с компилятором C и на платформе Java.

Программа на языке Python может состоять из одного или нескольких модулей. Каждый модуль представляет собой текстовый файл в кодировке, совместимой с 7-битной кодировкой ASCII. Для кодировок, использующих старший бит, необходимо явно указывать название кодировки. Например, модуль, комментарии или строковые литералы которого записаны в кодировке CP-1251 (она же windows-1251), должен иметь в первой или второй строке следующую спецификацию:

```
# -*- coding: cp-1251 -*-
```

Благодаря этой спецификации интерпретатор Python будет знать, как корректно переводить символы литералов Unicode–строк в Unicode. Без этой строки новые версии Python будут выдавать предупреждение на каждый модуль, в котором встречаются коды с установленным восьмым битом.

В примерах ниже используются как фрагменты модулей, записанных в файл, так и фрагменты диалога с интерпретатором Python. Последние отличаются характерным приглашением >>>. Символ решетка (#) отмечает комментарий до конца строки.

Программа на Python, с точки зрения интерпретатора, состоит из логических строк. Одна логическая строка, как правило, располагается в одной физической, но длинные логические строки можно явно (с помощью обратной косой черты) или неявно (внутри скобок) разбить на несколько физических:

```
print a, " - очень длинная строка, которая не помещается в", \
   80, "знакоместах"
```

2 Алгоритмические конструкции

2.1 Последовательность действий

Последовательные действия описываются последовательными строками программы. В программах важны отступы, поэтому все операторы, входящие в одну последовательность действий, должны иметь один и тот же отступ:

```
a = 1
b = 2
a = a + b
b = a - b
a = a - b
print a, b
```

Проверить этот пример можно с помощью интерактивного режима интерпретатора Python. При работе с Python в интерактивном режиме как бы вводится одна большая программа, состоящая из последовательных действий. В примере выше использованы операторы присваивания и оператор **print**.

2.2 Операторы выбора и ветвления

Ветвление действий в зависимости от выполнения некоторого логического условия осуществляется следующим образом:

```
if a > b :
    c = a
else:
    c = b
```

Этот кусок кода на Python интуитивно понятен каждому, кто помнит, что **if** по-английски значит «если», а **else** — «иначе». Оператор ветвления имеет в данном случае две части, операторы каждой из которых записываются с отступом вправо относительно оператора ветвления. Более общий случай — оператор выбора — можно записать с помощью следующего синтаксиса (пример вычисления знака числа):

```
if a < 0:
    s = -1
elif a == 0:
    s = 0
```

```
else:
    s = 1
```

Стоит заметить, что **elif** — это сокращенный **else if**. Без сокращения пришлось бы применять вложенный оператор ветвления:

```
if a < 0:
    s = -1
else:
    if a == 0:
        s = 0
    else:
        s = 1
```

В отличие от оператора **print**, оператор **if-else** — составной оператор.

2.3 Операторы цикла

В Python имеются два вида циклов: цикл **ПОКА** (выполняется некоторое действие) и цикл **ДЛЯ** (всех значений последовательности). Следующий пример иллюстрирует цикл **ПОКА** на Python:

```
s = "abcdefghijklmnop"
while s != "":
    print s
    s = s[1:-1]
```

Оператор **while** говорит интерпретатору Python: пока верно условие цикла, выполнять тело цикла. В языке Python тело цикла выделяется отступом. Каждое исполнение тела цикла будет называться итерацией. В приведенном примере убирается первый и последний символ строки до тех пор, пока не останется пустая строка.

Для большей гибкости при организации циклов применяются операторы **break** (прервать) и **continue** (продолжить). Первый позволяет прервать цикл, а второй - продолжить цикл, перейдя к следующей итерации (если, конечно, выполняется условие цикла). Следующий пример читает строки из файла и выводит те, у которых длина больше 5:

```
f = open("file.txt", "r")
while 1:
    l = f.readline()
    if not l:
        break
    if len(l) > 5:
    print l,
f.close()
```

В этом примере организован бесконечный цикл, который прерывается только при получении из файла пустой строки (l), что обозначает конец файла.

В языке Python логическое значение несет каждый объект: нули, пустые строки и последовательности, специальный объект **None** и логический литерал **False** имеют значение «ложь», а прочие объекты значение «истина». Для обозначения истины обычно используется 1 или **True**.

Цикл **ДЛЯ** выполняет тело цикла для каждого элемента последовательности. В следующем примере выводится таблица умножения:

```
for i in range(1, 10):
    for j in range(1, 10):
        print "%2i" % (i*j),
    print
```

Здесь циклы **for** являются вложенными. Функция **range()** порождает список целых чисел из полуоткрытого диапазона [1, 10). Перед каждой итерацией **счетчик цикла** получает очередное значение из этого списка. Полуоткрытые диапазоны общеприняты в Python. Считается, что их использование более удобно и вызывает меньше программистских ошибок. Например, **range(len(s))** порождает список индексов для списка s (в Python–последовательности первый элемент имеет индекс 0). Для красивого вывода таблицы умножения применена операция форматирования % (для целых чисел тот же символ используется для обозначения операции взятия остатка от деления). Строка форматирования (задается слева) строится почти как строка форматирования для **printf** из C.

2.4 Определение функций

Программист может определять собственные функции двумя способами: с помощью оператора **def** или прямо в выражении, посредством **lambda**.

Определение функции должно содержать список формальных параметров и тело определения функции. В случае с оператором **def** функции также задается некоторое имя. Формальные параметры являются локальными именами внутри тела определения функции, а при вызове функции они оказываются связанными с объектами, переданными как фактические параметры. Значения по умолчанию вычисляются в момент выполнения оператора **def**, и потому в них можно использовать видимые на момент определения имена. Вызов функции синтаксически выглядит как объект–функция (фактические параметры). Обычно объект–функция — это просто имя функции, хотя это может быть и любое выражение, которое в результате вычисления дает исполняемый объект. Функция одного аргумента:

```
def swapcase(s):
    return s.swapcase()

print swapcase('ABC')
```

Функция двух аргументов, один из которых необязателен и имеет значение по умолчанию:

```
def inc(n, delta=1):
    return n + delta

print inc(12)
print inc(12, 2)
```

Функция с одним обязательным аргументом, с одним, имеющим значение по умолчанию и неопределенным числом именованных аргументов:

```
def wrap(text, width=70, **kwargs):
    from textwrap import TextWrapper
    # kwargs - словарь с именами и значениями аргументов
    w = TextWrapper(width=width, **kwargs)
    return w.wrap(text)
```

```
print wrap('My long text ...', width=4)
```

Функция произвольного числа аргументов:

```
def max_min(*args):
    # args - список аргументов в порядке их указания при вызове
    return max(args), min(args)

print max_min(1, 2, -1, 5, 3)
```

Функция с обычными (позиционными) и именованными аргументами:

```
def swiss_knife(arg1, *args, **kwargs):
    print arg1
    print args
    print kwargs
    return None

print swiss_knife(1)
print swiss_knife(1, 2, 3, 4, 5)
print swiss_knife(1, 2, 3, a='abc', b='sdf')
# print swiss_knife(1, a='abc', 3, 4) # !!! ошибка

lst = [2, 3, 4, 5]
dct = {'a': 'abc', 'b': 'sdf'}
print swiss_knife(1, *lst, **dct)
```

Приведем теперь пример определения функции с помощью **lambda**–выражения дан ниже:

```
func = lambda x, y: x + y
```

В результате **lambda**–выражения получается безымянный объект-функция, которая затем используется, например, для того, чтобы связать с ней некоторое имя. Однако, как правило, определяемые **lambda**–выражением функции, применяются в качестве параметров функций.

В языке Python функция может возвратить только одно значение, которое может быть кортежем. В следующем примере видно, как стандартная функция **divmod()** возвращает частное и остаток от деления двух чисел:

```
def bin(n):
    '''Цифры двоичного представления натурального числа'''
    digits = []
    while n > 0:
        n, d = divmod(n, 2)
        digits = [d] + digits
    return digits

print bin(69)
```

Важно понять, что за именем функции стоит объект. Этот объект можно связать с другим именем:

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

```
    # теперь addition и add - разные
    # имена одного и того же объекта
    addition = add
```

Ниже приводится пример, в котором в качестве значения по умолчанию аргумента функции используется изменчивый объект (список). Этот объект — один и тот же для всех вызовов функций, что может привести к казусам:

```
def mylist(val, lst=[]):
    lst.append(val)
    return lst

print mylist(1),
print mylist(2)
```

Вместо ожидаемого [1] [2] получается [1] [1, 2], так как добавляются элементы к «значению по умолчанию». Правильный вариант решения будет, например, таким:

```
def mylist(val, lst=None):
    lst = lst or []
    lst.append(val)
    return lst
```

Конечно, приведенная выше форма может использоваться для хранения в функции некоторого состояния между ее вызовами, однако, практически всегда вместо функции с таким побочным эффектом лучше написать класс и использовать его экземпляр.

2.5 Обработка исключений

В современных программах передача управления происходит не всегда так гладко, как в описанных выше конструкциях. Для обработки особых ситуаций (таких как деление на ноль или попытка чтения из несуществующего файла) применяется механизм исключений. Лучше всего пояснить синтаксис оператора **try-except** следующим примером:

```
try:
    res = int(open('a.txt').read()) / int(open('c.txt').read())
    print res
except IOError:
    print "Ошибка ввода-вывода"
except ZeroDivisionError:
    print "Деление на 0"
except KeyboardInterrupt:
    print "Прерывание с клавиатуры"
except:
    print "Ошибка"
```

В этом примере берутся числа из двух файлов и делятся одно на другое. В результате этих нехитрых действий может возникнуть несколько исключительных ситуаций, некоторые из них отмечены в частях **except** (здесь использованы стандартные встроенные исключения Python). Последняя часть **except** в этом примере улавливает все другие исключения, которые не были пойманы выше. Например, если хотя бы в одном из файлов находится нечисловое значение, функция **int()** возбудит исключение *ValueError*. Его-то и сможет отловить последняя часть

except. Разумеется, выполнение части **try** в случае возникновения ошибки уже не продолжается после выполнения одной из частей **except**.

В отличие от других языков программирования, в Python исключения нередко служат для упрощения алгоритмов. Записывая оператор **try-except**, программист может думать так: "попробую, а если сорвется — выполнится код в except". Особенно часто это используется для выражений, в которых значение получается по ключу из отображения:

```
try:
    value = dict[key]
except:
    value = default_value
```

Вместо

```
if dict.has_key(key):
    value = dict[key]
else:
    value = default_value
```

Заметим, что в современном Python лучше писать так

```
value = dict.get(key, default_value)
```

Исключения можно возбуждать и из программы. Для этого служит оператор **raise**. Заодно следующий пример показывает канонический способ определения собственного исключения:

```
class MyError(Exception):
    pass

try:
    ...
    raise MyError, "My error 1"
    ...
except MyError, x:
    print "Ошибка:", x
```

Все исключения выстроены в иерархию классов, поэтому *ZeroDivisionError* может быть поймана как *ArithmeticError*, если соответствующая часть **except** будет идти раньше. Для утверждений применяется специальный оператор **assert**. Он возбуждает *AssertionError*, если заданное в нем условие неверно. Этот оператор используют для самопроверки программы. В оптимизированном коде он не выполняется, поэтому строить на нем логику алгоритма нельзя. Пример:

```
c = a + b
assert c == a + b
```

Кроме описанной формы оператора, есть еще форма **try-finally** для гарантированного выполнения некоторых действий при передаче управления изнутри оператора **try-finally** вовне. Он может применяться для освобождения занятых ресурсов, что требует обязательного выполнения, независимо от произошедших внутри катаклизмов:

```
try:
    ...
finally:
    print "Обработка гарантированно завершена"
```

Смешивать вместе формы **try-except** и **try-finally** нельзя.

3 Встроенные типы данных

Все данные в Python — объекты в смысле ООП. Имена являются ссылками на объекты и не несут никакой информации о его типе. Тип определяется во время исполнения кода, поэтому вместо «присваивания значения переменной» лучше говорить о «связывании значения с некоторым именем». Способ динамической типизации, применяемый в Python, называется утиной (неявной) типизацией. Название происходит от английского шуточного «утиного теста»:

Если это выглядит как утка, плавает как утка и крякает как утка, то, вероятно, это утка.

Объекты могут быть **изменчивыми** и **неизменчивыми**. К примеру, строки являются неизменчивыми, так что при операциях над строками создаются новые строки, а не изменяются старые. Тип каждого объекта можно узнать при помощи функции **type()**.

3.1 Численные типы

Существует четыре встроенных типа для представления чисел. Один из них — **complex** — служит для представления комплексных чисел (арифметические операции встроены). Реализуется это добавлением **j** в качестве суффикса к мнимой части числа:

```
>>> -1j * -1j
(-1-0j)
```

Еще два типа: **int** и **long** служат моделью для представления целых чисел. Первый соответствует типу long в компиляторе C для используемой архитектуры. Второй тип реализует представление для целых чисел **произвольной точности**. Числовые литералы можно записать в системах счисления с основанием 8, 10 или 16:

```
# В этих литералах записано число 10
print 10, 012, 0xA, 10L
```

Набор операций над числами — достаточно стандартный как по семантике, так и по обозначениям:

```
>>> print 1 + 1, 3 - 2, 2*2, 7/4, 5%3
2 1 4 1 2
>>> print 2L ** 1000
10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371418779541821530464749835819412673987675591655439460770629145711964776865421676604298316526243868372056680693769
>>> print 3 < 4 < 6,
3 >= 5,
4 &=&  4,
4 != 4 # сравнения
```

```
True False True False
>>> print 1 << 8,
4 >> 2,
~4
# побитовые сдвиги и инверсия
256 1 -5
>>> for i, j in (0, 0), (0, 1), (1, 0), (1, 1):
    ...
    # побитовые операции
    print i, j, ":", i & j, i | j, i ^ j
    ...
0 0 : 0 0 0
0 1 : 0 1 1
1 0 : 0 1 1
1 1 : 1 1 0
```

Значения типа **int** должны покрывать диапазон от -2147483648 до 2147483647, а точность целых произвольной точности зависит от объема доступной памяти.

Стоит заметить, что если в результате операции получается значение, выходящее за рамки допустимого, тип **int** может быть неявно преобразован в **long**:

```
>>> type(-2147483648)
<type 'int'>
>>> type(-2147483649)
<type 'long'>
```

Также нужно быть осторожным при записи констант. Ноли в начале числа — признак восьмеричной системы счисления, в которой нет цифры 8:

```
>>> 008
  File "<stdin>", line 1
    008
      ^
SyntaxError: invalid token
```

Наконец, тип **float** служит для представления чисел с плавающей точкой. Он соответствует C–типу **double** для используемой архитектуры. Записывается вполне традиционным способом либо через точку, либо в экспоненциальной форме:

```
>>> pi = 3.1415926535897931
>>> pi ** 40
7.6912142205156999e+19
```

Кроме арифметических операций, можно использовать операции из модуля **math**. Для этого типа есть встроенные функции: **round()**, **abs()**.

3.2 Логический тип

Существует специальный подтип целочисленного типа для «канонического» обозначения логических величин. Два значения: **True** (истина) и **False** (ложь) — все, что принадлежит этому типу. Как уже говорилось, любой объект Python имеет истинностное значение, логические операции можно проиллюстрировать с помощью логического типа:

```
>>> for i in (False, True):
...     for j in (False, True):
...     print i, j, ":", i and j, i or j, not i
...
...
False False : False False True
False True : False True True
True False : False True False
True True : True True False
```

Следует отметить, что Python не вычисляет второй операнд операции **and** или **or**, если ее исход ясен по первому операнду. Например, если первый операнд истинен, он и возвращается как результат **or**, в противном случае возвращается второй операнд.

3.3 Строковый тип

В Python строки бывают двух типов: обычные и Unicode-строки. Фактически строка — это последовательность символов (в случае обычных строк можно сказать «последовательность байтов»). Строки-константы можно задать в программе с помощью строковых литералов. Для литералов наравне используются как апострофы (’), так и обычные двойные кавычки (”). Для многострочных литералов можно использовать утроенные апострофы или утроенные кавычки. Управляющие последовательности внутри строковых литералов задаются обратной косой чертой. Примеры написания строковых литералов:

```
s1 = "строка1"
s2 = 'строка2\nс переводом строки внутри'
s3 = '''строка3
с переводом строки внутри'''
u1 = u'\u043f\u0440\u0438\u0432\u0435\u0442' # привет
u2 = u'Еще пример' # не забудьте про coding!
```

Для строк имеется еще одна разновидность: необработанные строковые литералы. В этих литералах обратная косая черта и следующие за ней символы не интерпретируются как спецсимволы, а вставляются в строку «как есть»:

```
my_re = r"(\d)=\1"
```

Обычно такие строки требуются для записи регулярных выражений (о них пойдет речь в лекции, посвященной обработке текстовой информации).

Набор операций над строками включает конкатенацию ’+’, повтор ’*’, форматирование ’%’. Также строки имеют большое количество методов, некоторые из которых приведены ниже. Полный набор методов (и их необязательных аргументов) можно получить в документации по Python.

```
>>> "A" + "B"
'AB'
>>> "A"*10
'AAAAAAAAAA'
>>> ''%s %i'' % (''abc'', 12)
'abc 12'
```

3.4 Составные типы

3.4.1 Кортеж

Для представления константной последовательности (разнородных) объектов используется тип **кортеж** (**tuple**). Литерал кортежа обычно записывается в круглых скобках. Однако, если не возникают неоднозначности, можно писать и без них. Примеры записи кортежей:

```
p = (1.2, 3.4, 0.9) # точка в трехмерном пространстве
for s in 'one', 'two', 'three': # цикл по значениям кортежа
    print s
one_item = (1,)
empty = ()
p1 = 1, 3, 9  # без скобок
p2 = 3, 8, 5, # запятая в конце игнорируется
```

Использовать синтаксис кортежей можно и в левой части оператора присваивания. В этом случае на основе вычисленных справа значений формируется кортеж и связывается один в один с именами в левой части. Поэтому обмен значениями записывается очень изящно:

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

3.4.2 Список

В «чистом» Python нет массивов с произвольным типом элемента. Вместо них используются **списки** (**list**). Их можно задать с помощью литералов, записываемых в квадратных скобках, или посредством списковых включений. Варианты задания списка приведены ниже:

```
lst1 = [1, 2, 3,]
lst2 = [x**2 for x in range(10) if x % 2 == 1]
lst3 = list('abcde')
```

Для работы со списками существует несколько методов, дополнительных к тем, что имеют неизменчивые последовательности. Все они связаны с изменением списка.

3.4.3 Операции над последовательностями

Ниже обобщены основные методы последовательностей. Следует напомнить, что последовательности бывают неизменчивыми и изменчивыми. У последних методов чуть больше.

Синтаксис	**Семантика**
len(s)	Длина последовательности s
x in s	Проверка принадлежности элемента последовательности. В новых версиях Python можно проверять принадлежность подстроки строке. Возвращает True или False
x not in s	Тоже самое, что и not x in s
s + s1	Конкатенация последовательностей
s*n или n*s	Последовательность из n раз повторенной s. Если n $<$ 0, возвращается пустая n*s последовательность.
s[i]	Возвращает i-й элемент s или len(s)+i-й, если i $<$ 0
s[i:j:d]	Срез из последовательности s от i до j с шагом d
min(s)	Наименьший элемент s
max(s)	Наибольший элемент s

Дополнительные конструкции для изменчивых последовательностей:

s[i] = x	i-й элемент списка s заменяется на x
s[i:j:d] = t	Срез от i до j (с шагом d) заменяется на (список) t
del s[i:j:d]	Удаление элементов среза из последовательности

В таблице ниже приведен ряд методов изменчивых последовательностей (например, списков).

Метод	**Описание**
append(x)	Добавляет элемент в конец последовательности
count(x)	Считает количество элементов, равных x
extend(s)	Добавляет к концу последовательности последовательность s
index(x)	Возвращает наименьшее i, такое, что s[i] == x. Возбуждает исключение ValueError, если x не найден в s
insert(i, x)	Вставляет элемент x в i-й промежуток
pop([i])	Возвращает i-й элемент, удаляя его из последовательности
reverse()	Меняет порядок элементов на обратный
sort([cmpfunc])	Сортирует элементы последовательности. Может быть указана своя функция сравнения cmpfunc

Для получения отдельного элемента последовательности используются квадратные скобки, в которых стоит выражение, дающее индекс. Индексы последовательностей в Python начинаются с нуля. Отрицательные индексы служат для отсчета элементов с конца последовательности (-1 — последний элемент). Пример проясняет дело:

```
>>> s = [0, 1, 2, 3, 4]
>>> print s[0], s[-1], s[3]
0 4 3
>>> s[2] = -2
>>> print s
[0, 1, -2, 3, 4]
>>> del s[2]
>>> print s
[0, 1, 3, 4]
```

Удалять элементы можно только из изменчивых последовательностей и желательно не делать этого внутри цикла по последовательности.

При взятии среза последовательности принято нумеровать не элементы, а промежутки между ними, так как это удобно для указания произвольных срезов. Перед нулевым (по индексу) элементом последовательности промежуток имеет номер 0, после него — 1 и т.д. Отрицательные значения отсчитывают промежутки с конца строки. Для записи срезов используется следующий синтаксис:

```
последовательность[нач:кон:шаг]
```

где нач - промежуток начала среза, кон - конца среза, шаг - шаг. По умолчанию нач=0, кон=len(последовательность), шаг=1. Если шаг не указан, второе двоеточие можно опустить. Пример работы со срезами:

```
>>> s = range(10)
>>> s
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> s[0:3]
[0, 1, 2]
>>> s[-1:]
[9]
>>> s[::3]
[0, 3, 6, 9]
>>> s[0:0] = [-1, -1, -1]
>>> s
[-1, -1, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> del s[:3]
>>> s
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Как видно из этого примера, с помощью срезов удобно задавать любую подстроку, даже если она нулевой длины, как для удаления элементов, так и для вставки в строго определенное место.

3.4.4 Словарь

Словарь (хэш, ассоциативный массив) — это изменчивая структура данных для хранения пар ключ–значение, где значение однозначно определяется ключом. В качестве ключа может выступать неизменчивый тип данных (число, строка, кортеж и т.п.). Порядок пар ключ–значение произволен. Ниже приведен литерал для словаря и пример работы со словарем:

```
d = {1: 'one', 2: 'two', 3: 'three', 4: 'four'}
d0 = {0: 'zero'}
print d[1] # берется значение по ключу
d[0] = 0 # присваивается значение по ключу
del d[0] # удаляется пара ключ-значение с данным ключом
print d
for key, val in d.items(): # цикл по всему словарю
    print key, val
for key in d.keys(): # цикл по ключам словаря
    print key, d[key]
for val in d.values():  # цикл по значениям словаря
    print val
```

```
d.update(d0) # пополняется словарь из другого
print len(d) # количество пар в словаре
```

3.4.5 Тип file

Объекты этого типа предназначены для работы с внешними данными. В простом случае — это файл на диске. Файловые объекты должны поддерживать основные методы: **read(), write(), readline(), readlines(), seek(), tell(), close()** и т.п.

Следующий пример показывает, как можно организовать копирование файла:

```
f1 = open('file1.txt', 'r')
f2 = open('file2.txt', 'w')
for line in f1.readlines():
    f2.write(line)
f2.close()
f1.close()
```

Стоит заметить, что кроме собственно файлов в Python используются и файлоподобные объекты. В очень многих функциях просто неважно, передан ли ей объект типа **file** или другого типа, если он имеет все те же методы (и в том же смысле). Например, копирование содержимого по ссылке (URL) в файл file2.txt можно достигнуть, если заменить первую строку на

```
import urllib
f1 = urllib.urlopen('www.google.com')
```

4 Выражения и переменные

4.1 Приоритет операций в выражениях

Приоритет операций показан в нижеследующей таблице (в порядке уменьшения). Для унарных операций x обозначает операнд. Ассоциативность операций в Python — слева–направо, за исключением операции возведения в степень (**), которая ассоциативна справа налево.

Операция	**Название**
lambda	лямбда–выражение
or	логическое ИЛИ
and	логическое И
not x	логическое НЕ
in, not in	проверка принадлежности
is, is not	проверка идентичности
<,<=,>,>=, !=, ==	сравнения
tmp	побитовое ИЛИ
hat	побитовое исключающее ИЛИ
amper	побитовое И
«, »	побитовые сдвиги
+, -	сложение и вычитание
*, /, %	умножение, деление, остаток
+x, -x	унарный плюс и смена знака
x	побитовое НЕ
**	возведение в степень
x.атрибут	ссылка на атрибут
x[индекс]	взятие элемента по индексу
x[от:до]	выделение среза (от и до)
f(аргумент,...)	вызов функции
(...)	скобки или кортеж
[...]	список или списковое включение
{кл:зн, ...}	словарь пар ключ–значение
'выражения'	преобразование к строке (repr)

Таким образом, порядок вычислений операндов определяется такими правилами:

1. Операнд слева вычисляется раньше операнда справа во всех бинарных операциях, кроме возведения в степень.

2. Цепочка сравнений вида a < b < c ... y < z фактически равносильна: (a < b) and (b < c) and ... and (y < z).

3. Перед фактическим выполнением операции вычисляются нужные для нее операнды. В большинстве бинарных операций предварительно вычисляются оба операнда (сначала левый), но операции or и and , а также цепочки сравнений вычисляют такое количество операндов, которое достаточно для получения результата. В невычисленной части выражения в таком случае могут даже быть неопределенные имена. Это важно учитывать, если используются функции с побочными эффектами.

4. Аргументы функций, выражения для списков, кортежей, словарей и т.п. вычисляются слева-направо, в порядке следования в выражении.

В случае неясности приоритетов желательно применять скобки. Несмотря на то, что одни и те же символы могут использоваться для разных операций, приоритеты операций не меняются. Так, % имеет тот же приоритет, что и *, а потому в следующем примере скобки просто необходимы, чтобы операция умножения произошла перед операцией форматирования:

```
print '%i' % (i*j)
```

Выражения могут фигурировать во многих операторах Python и даже как самостоятельный оператор. У выражения всегда есть результат, хотя в некоторых случаях (когда выражение вычисляется ради побочных эффектов) этот результат может быть «ничем» — **None**. Очень часто выражения стоят в правой части оператора присваивания или расширенного присваивания. В Python (в отличие, скажем, от C) нет операции присваивания, поэтому синтаксически перед знаком = могут стоять только идентификатор, индекс, срез, доступ к атрибуту или кортеж (список) из перечисленного.

4.2 Имена переменных

Имя может начинаться с латинской буквы (любого регистра) или подчеркивания, а дальше допустимо использование цифр. В качестве идентификаторов нельзя применять ключевые слова языка и нежелательно переопределять встроенные имена. Список ключевых слов можно узнать так:

```
>>> import keyword
>>> keyword.kwlist
['and', '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', 'yield']
```

Имена, начинающиеся с подчеркивания или двух подчеркиваний, имеют особый смысл. Одиночное подчеркивание говорит о том, что имя имеет местное применение, и не должно использоваться за пределами модуля. Двойным подчеркиванием в начале и в конце обычно наделяются специальные имена атрибутов классов (об этом позже).

4.3 Пространства имен и область видимости

Пространство имен — отображение из имен в объекты. В каждой точке программы интерпретатор видит три пространства имен: локальное, глобальное и встроенное.

Для понимания того, как Python находит значение некоторой переменной, необходимо ввести понятие **блока кода**. В Python блоком кода является то, что исполняется как единое целое, например, тело определения функции, класса или модуля.

Локальные имена — имена, которым присвоено значение в данном блоке кода. Глобальные имена — имена, определяемые на уровне блока кода определения модуля или те, которые явно заданы в операторе **global**. Встроенные имена — имена из специального словаря _builtins_.

Области видимости имен могут быть вложенными друг в друга, например, внутри вызванной функции видны имена, определенные в вызывающем коде. Переменные, которые используются в блоке кода, но связаны со значением вне кода, называются **свободными переменными**.

Так как переменную можно связать с объектом в любом месте блока, важно, чтобы это произошло до ее использования, иначе будет возбуждено исключение *NameError*. Связывание имен со значениями происходит в операторах присваивания, **for**, **import**, в формальных аргументах функций, при определении функции или класса, во втором параметре части **except** оператора **try-except**.

С областями видимости и связыванием имен есть много нюансов, которые хорошо описаны в документации. Желательно, чтобы программы не зависели от таких нюансов, а для этого достаточно придерживаться следующих правил:

1. Всегда следует связывать переменную со значением (текстуально) до ее использования.

2. Необходимо избегать глобальных переменных и передавать все в качестве параметров. Глобальными на уровне модуля должны остаться только имена-константы, имена классов и функций.

3. Никогда не следует использовать конструкцию

   ```
   from модуль import *
   ```

   Это может привести к затенению имен из других модулей, а внутри определения функции просто запрещено.

Предпочтительнее переделать код, нежели использовать глобальную переменную. Конечно, для программ, состоящих из одного модуля, это не так важно: все определенные на уровне модуля переменные глобальны.

Убрать связь имени с объектом можно с помощью оператора **del**. В этом случае, если объект не имеет других ссылок на него, он будет удален. Для управления памятью в Python используется **подсчет ссылок** (reference counting), для удаления наборов объемов с зацикленными ссылками — **сборка мусора** (garbage collection).

5 Стиль программирования и стандарты

5.1 Официальный style guide

Стиль программирования — дополнительные ограничения, накладываемые на структуру и вид программного кода группой совместно работающих программистов с целью получения удобных для применения, легко читаемых и эффективных программ. Основные ограничения на вид программы дает синтаксис языка программирования, и его нарушения вызывают синтаксические ошибки. Нарушение стиля не приводит к синтаксическим ошибкам, однако как отдельные программисты, так и целые коллективы сознательно ограничивают себя в средствах выражения ради упрощения совместной разработки, отладки и сопровождения программного продукта.

Стиль программирования затрагивает практически все аспекты написания исходного кода:

- именование объектов в зависимости от типа, назначения, области видимости;
- оформление функций, методов, классов, модулей и их документирование в коде программы;
- декомпозиция программы на модули с определенными характеристиками;
- способ включения отладочной информации;
- применение тех или иных функций (методов) в зависимости от предполагаемого уровня совместимости разрабатываемой программы с различными компьютерными платформами;

- ограничение используемых функций из соображений безопасности.

Для языка Python Гвидо ван Россум разработал официальный стиль. С оригинальным текстом «Python Style Guide» можно ознакомиться по адресу

http://www.python.org/dev/peps/pep-0008/

Его перевод (неофициальный) на русский язык можно найти в прилагаемом файле. Кроме того, существует так называемый «дзен» Python, оригинальный текст которого можно найти по адресу

http://www.python.org/dev/peps/pep-0020/ .

Хотя слишком серьезно воспринимать его не следует, знать о его существовании необходимо:

- Красивое лучше, чем уродливое.
- Явное лучше, чем неявное.
- Простое лучше, чем сложное.
- Сложное лучше, чем запутанное.
- Плоское лучше, чем вложенное.
- Разреженное лучше, чем плотное.
- Читаемость имеет значение.
- Особые случаи не настолько особые, чтобы нарушать правила.
- При этом практичность важнее безупречности.
- Ошибки никогда не должны замалчиваться.
- Если не замалчиваются явно.
- Встретив двусмысленность, отбрось искушение угадать.
- Должен существовать один — и, желательно, только один — очевидный способ сделать это.
- Хотя он поначалу может быть и не очевиден, если вы не голландец.
- Сейчас лучше, чем никогда.
- Хотя никогда зачастую лучше, чем прямо сейчас.
- Если реализацию сложно объяснить — идея плоха.
- Если реализацию легко объяснить — идея, возможно, хороша.
- Пространства имён — отличная штука! Будем делать их побольше!

5.2 Сборник PEP

Помимо обширной документации по синтаксису языка и модулям стандартной библиотеки, на официальном сайте организации, которая занимается развитием языка Python, можно найти и так называемый PEP — Python Enhancement Proposal (Предложения по улучшению Python):

http://www.python.org/dev/peps/

Этот сборник документов регулирует как введение новых возможностей в язык, так и стандартное использование существующих возможностей.