

Программирование на Python

1. Типы данных в Python

Настало время познакомиться с типами данных в Python. Числовые типы `int` и `float` уже были упомянуты ранее:

```
x = 3.5
print type(x) % OUT: <type 'float'>

y = 4
print type(y) % OUT: <type 'int'>
```

С помощью функции `type` можно узнать код произвольного объекта. Строки, например, могут быть двух типов:

```
print type("abs") % OUT: <type 'str'>
print type(u"abs") % OUT: <type 'unicode'>
```

Отличия между ними будут подробнее обсуждаться позднее.

Упорядоченный набор значений в Python можно представить с помощью списка:

```
w = [1, 2, 3]
print type(w) % OUT: <type 'list'>
```

Списки в Python очень похожи на знакомые по другим языкам программирования массивы, но, в отличие от массивов, могут содержать элементы разных типов.

Python является языком с утиной типизацией. В языках с динамической типизацией, частным случаем которой является утиная типизация, тип переменной определяется не в момент её объявления, а в момент присваивания значения. Следовательно, в разных участках кода переменная может принимать значения разных типов. В случае утиной типизации границы применимости объекта определяются не столько конкретным типом и иерархией наследования, сколько наличием и отсутствием у него определенных методов и свойств. Термин происходит от шуточного «утиного теста»: «Если нечто выглядит, плавает и крикает как утка, то это, вероятно, и есть утка».

Другой способ представить упорядоченный набор значений состоит в определении кортежа (`tuple`). В Python кортежи записываются в круглых скобках:

```
print type((1, 2, 3)) % OUT: <type 'tuple'>
```

Кортеж относится к так называемым неизменяемым типам данных, чем существенно отличается от списка. Попытка изменить кортеж или добавить в него еще один элемент приведет к возникновению ошибки. Эта особенность позволяет использовать кортежи в качестве ключей словаря.

Неупорядоченный набор значений представим в виде множеств, которые записываются в фигурных скобках:

```
print type({1, 2, 3}) % OUT: <type 'set'>
```

Еще один тип данных, словарь, позволяет хранить в себе пары вида ключ-значение. Создать словарь можно с помощью конструктора `dict()`:

```
e = dict() % Создан пустой словарь
e['abc'] = 3.5 % Строке-ключу 'abc' поставлено в
               % соответствие значение 3.5 типа float
```

Ключами в словаре могут быть не только строки. В качестве ключа допустимо использовать, например, числовые значения и кортежи:

```
e[3.5] = 'abc'
e[(1, 2, 3)] = 4.5
```

Но использование списка в качестве **ключа недопустимо**, так как список является **изменяемым**.

Синтаксис для чтения данных из словаря по ключу аналогичен синтаксису для получения данных по **индексу списка**. Грубо говоря, ключ есть обобщение понятия индекса списка: ключ может принимать значение произвольного хэшируемого типа. К слову, список не является хэшируемым, и именно поэтому не может быть выбран в качестве ключа.

В Python словарь использует хэш-таблицы. Идея заключается в том, чтобы заменить сравнение ключей более быстрым **сравнением их хэш-значений**. В таком случае важно, чтобы все ключи были хэшируемы, то есть хэш-значение не менялось во время исполнения программы, а равные ключи-объекты имели одинаковое хэш-значение.

Ранее было сказано, что множества являются изменяемыми объектами, а значит не могут выступать в роли ключей. Но в качестве ключей можно использовать неизменяемые множества, которые можно получить используя функцию `frozenset()`:

```
s = frozenset(1,2,3)
e[s] = 3.76
```

Таким образом, к данному моменту были изучены основные типы данных в Python и подробно было рассказано о существенных отличиях изменяемых и неизменяемых типов. На следующем занятии будут рассмотрены основные управляющие конструкции языка Python.

2. Синтаксис языка Python

В Python оператор условного перехода выглядит привычным образом. Например, пусть дан следующий код:

```
if x:
    print "OK"
else:
    print "NOT OK"
```

Если переменная `x` была равна **True**, то исполнится код в теле инструкции `if` и на экран будет выведено `OK`. В противном случае, например если `x` определена как **False**, исполнится код в теле инструкции `else`. В Python для выделения блоков кода используются **отступы**: отступы инструкций в пределах одного блока должны **совпадать по величине**.

Другой пример. Следующий код выводит числа от 0 до 9:

```
for i in range(10):
    print i
```

При этом в `range()` можно указать начальное значение `i`. Например, вот этот код выводит числа от 1 до 9.

```
for i in range(1,10):
    print i
```

В данных примерах каждое число выводится на отдельной строке, поскольку `print` по умолчанию **начинает новую строку** после вывода требуемого значения. Чтобы все числа выводились **в одной строке**, необходимо добавить **запятую** в конец строки с `print` следующим образом:

```
for i in range(1,10):
    print i,
```

Теперь числа от 1 до 9 выводятся в одной строке через пробел.

На самом деле `range()` это функция, которая возвращает список натуральных чисел в определяемом её аргументами диапазоне:

```
print range(2,5)
OUT: [2,3,4]
```

Использовать `range()` совершенно не обязательно. Циклы в Python перебирают по очереди элементы списка, стоящего после `in`. Например, можно написать так:

```
for i in [2,3,4]:
    print i,
```

В Python 2 кроме функции `range()` существует и функция `xrange()`, которую также можно использовать в цикле:

```
for i in xrange(2,5):
    print i,
OUT: 2, 3, 4
```

Результат выполнения не отличается от такового при использовании `range()`. Но можно убедиться, что `range` и `xrange` возвращают объекты разных типов:

```
print type(range(2,5))
OUT: list
print type(xrange(2,5))
OUT: iter
```

Функция `xrange` возвращает не список, а так называемый генератор. В то время, как при использовании `range()` сначала создается список, а только потом происходит итерирование по этому списку, при использовании генераторов список никогда не создается и не занимает память. Это особенно актуально, когда необходимо итерировать в очень большом диапазоне значений.

```
print xrange(2,500000000)
OUT: xrange(2,500000000)
```

В Python существует способ удобно и компактно задавать некоторые списки с помощью так называемого конструктора списка (англ. list comprehension.):

```
w = [ x ** 2 for x in range(1,11) ]
print w
OUT: [ 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 ]
print type(w)
OUT: <type 'list'>
```

В этом примере создан список квадратов чисел от 1 до 10. Если необходимо построить список, например, квадратов только четных чисел из этого диапазона, конструктор допускает использование условия:

```
w = [ x ** 2 for x in range(1,11) if x % 2 == 0 ]
OUT: [ 4, 16, 36, 64, 100 ]
```

Аналогично выглядит конструктор генераторов:

```
w = ( x ** 2 for x in range(1,11) if x % 2 == 0 )
OUT: ( 4, 16, 36, 64, 100 )
print type(w)
OUT: <type 'generator'>
```

В этом случае весь список не будет храниться в памяти во время работы цикла.

Также в Python доступен цикл `while`, который продолжает выполнение, пока выполнено указанное условие, а также операторы `break`, который досрочно прерывает цикл, и `continue`, который начинает следующий проход цикла, минуя оставшееся тело цикла. Например:

```
s = 0
while True: % Это условие всегда выполняется
    s += 1
    if s % 2 == 0: % Через раз
        print "Continue" % вместо s будет выведено "continue"
        continue % и код ниже в таком случае не будет исполнен в этой итерации
    print s
    if s > 10: % С помощью этой конструкции
        break % <<бесконечный цикл досрочно прерывается>>
```

Важно уметь определять свои функции. Например, следующий код представляет собой альтернативную реализацию уже известной функции `range`:

```
def myrange(a,b):
    res = [ ]
    s = a
    while s!=b:
        res.append(s)
        s+=1
    return res
```

Этот пример демонстрирует синтаксис для создания функций. На самом деле при создании функции необходимо обработать особые случаи, но это не является текущей целью.

Функции в Python являются объектами **первого класса**: их можно передавать в качестве аргументов, присваивать их переменным и так далее. Например, функция `map()` позволяет обрабатывать одну или несколько последовательностей с помощью переданной в качестве аргумента функции:

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

print map(sq, range(10))
OUT: [ 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Функция в Python может быть определена как с помощью оператора `def`, так и с помощью лямбда-выражения. Следующие операторы эквивалентны:

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

sq = lambda x: x**2
```

Тогда предыдущий код можно записать более компактно:

```
print map(lambda x: x**2, range(10))
OUT: [ 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

3. Чтение данных из файла

Часто для анализа необходимо загрузить данные, находящиеся во внешнем файле. Этот файл может быть простым текстовым документом, XML-документом или иметь любой другой пригодный для хранения данных формат. Для достижения этой цели Python предоставляет целый ряд разнообразных инструментов. Работа будет производиться в новой тетради IPython.

Самый простой способ прочитать данные из файла — использовать функцию `open` из стандартной библиотеки. Эта функция позволяет считать содержимое простого текстового файла. Информацию о том, какие аргументы принимает функция и какое значение возвращает можно найти в `docstring` этой функции. Чтобы прочитать `docstring`, то есть строку документации по этой функции, необходимо набрать вопросительный знак и затем имя функции:

```
?open
```

В результате внизу появляется требуемое описание функции:

```
open(name[, mode[, buffering]]) -> file object

Open a file using the file() type, returns a file object. This is the
preferred way to open a file. See file.__doc__ for further information.
```

По данному тексту можно понять, что первый аргумент является обязательным и имеет смысл пути к файлу с данными (если файл лежит в рабочей директории, можно просто написать имя файла). Второй аргумент определяет режим, в котором требуется открыть файл. Существуют несколько основных режимов:

```
r    --- режим чтения      ( режим по умолчанию )
w    --- режим записи      ( данные из старого файла стираются )
a    --- режим дозаписи    ( новые данные будут добавлены в конец файла )
r+   --- режим чтения и записи
```

Последний аргумент сейчас не представляет интереса. Функция возвращает объект-файл, с помощью которого будет происходить взаимодействие с файлом:

```
file_obj = open('example_utf8.txt', 'r')
type(file_obj) % OUT file
```

Таким образом файл example_utf8.txt, находящийся в рабочем каталоге, будет открыт только для чтения. С помощью метода `read()` можно вывести на экран все содержимое файла:

```
print file_obj.read()
% OUTPUT:
% Привет, мир!
% тестовый файл
% урок чтения данных в python
..
```

Часто бывает удобно считывать файл не целиком. С помощью метода `readline` можно не считывать весь файл, а считать только несколько первых строк. При первом вызове метода будет возвращена первая строка, а при повторном — вторая. Также сначала нужно открыть файл для чтения заново.

```
file_obj = open('example_utf8.txt', 'r')
print file_obj.readline()
% OUTPUT:
% Привет, мир!
print file_obj.readline()
% OUTPUT:
% тестовый файл
```

На самом деле пользоваться такими функциями совершенно не обязательно. С файлом можно работать как с обычным генератором. В следующем примере строки файла выводятся с помощью цикла `for`:

```
file_obj = open('example_utf8.txt')
for line in file_obj:
    print line.strip()
```

Функция `strip()` удаляет все пробелы в начале и конце строки, включая символы табуляции, новой строки и так далее.

Бывает необходимым сохранить строки файла в виде списка. Это можно сделать с помощью конструктора списка `list()`, в качестве аргумента которого будет передан объект-файл. Поскольку объект-файл можно использовать как генератор, получившийся список будет содержать все строки файла:

```
file_obj = open('example_utf8.txt')
data_list = list(file_obj)
for line in data_list: print line.strip() % Проверка, что файл считан
```

Другой способ состоит в использовании метода `readlines()` файл-объекта. Этот метод возвращает список из всех содержащихся в файле строк:

```
file_obj = open('example_utf8.txt')
data_list = file_obj.readlines()
for line in data_list: print line.strip() % Проверка, что файл считан
```

Оба способа дают одинаковый результат и можно выбрать любой из них.

После того, как работа с файлом закончена, файл нужно закрыть с помощью команды `close()`. Считается хорошим тоном закрывать файлы, работа с которыми закончена. Во-первых, так будут освобождены системные ресурсы, которые задействованы в работе с файлом. Во-вторых, после закрытия файла он не может быть по ошибке испорчен и, если он редактировался, изменения точно будут сохранены на диске. Если метод `close()` был вызван для открытого файла, попытка продолжить работу с ним приведет к возникновению ошибки.

```
file_obj = open('example_utf8.txt')
file_obj.close()
file_obj.read()
% ValueError: I/O operation on closed file
```

Файл, с которым производилась работа до этого момента, был в кодировке utf8. Однако часто бывает необходимым прочитать файл в другой кодировке, например KOI8-R. Эта кодировка соответствует русскоязычной кириллице. Если непосредственно использовать функцию `open()`, как это делалось раньше:

```
file_obj = open('example_koi_8.txt')
print file_obj.read()
%  %D0%9F%D1%80%D0%B8%D0%B2%D0%B5%D1%82%2C%20%D0%BC%D0%B8%D1%80!
%  %0A%D1%82%D0%B5%D1%81%D1%82%D0%BE%D0%B2%D1%8B%D0%B9%20%D1%84%D0%B0%D0%B9%D0%BB
%  0A%D1%83%D1%80%D0%BE%D0%BA%20%5C%22%D1%87%D1%82%D0%B5%D0%BD%D0%B8%D0%B5%20%D0%B4%D0%
    B0%D0%BD%D0%BD%D1%8B%D1%85%20%D0%B2%20python%5C%22
```

Для того, чтобы корректно отобразить прочитанный файл, можно использовать библиотеку `codecs`. Импортирование библиотеки происходит с помощью ключевого слова `import`. После этого файл нужно прочитать функцией `open()`, которая определена в библиотеке `codecs`. Эта функция отличается тем, что в ней можно указать такие дополнительные параметры как кодировка файла.

```
import codecs
file_obj = codecs.open('example_koi_8.txt', 'r', encoding='koi8-r')
print file_obj.read()
%  Привет, мир!
%  тестовый файл
%  урок `чтение данных в python
..
```

Теперь, поскольку файл был открыт с указанием правильной кодировки, он отображается корректно. В следующем видео будет показано, как сохранять данные в файлы.

4. Запись данных в файл

В этом видео подробно разбирается, как записать данные в файл средствами Python. Задача следующая — создать несколько текстовых строк и записать их в определенный файл.

С помощью стандартной функции `open()` необходимо открыть файл для записи:

```
file_obj = open('file_to_write_in.txt', 'w')
```

Таким образом файл `file_to_write_in.txt` в рабочем каталоге создается для записи. Если файл с таким именем существовал до этого, его содержимое теряется.

Теперь создадим переменную для строки, которую требуется записать в файл:

```
string = 'строка для записи в файл\n'
```

В конце строки набран символ перевода строки. Это делается для того, чтобы при записи нескольких строк в файл они располагались на разных строчках, а не склеивались в одну. Запись производится с помощью метода `write`, который в качестве аргумента принимает строку для записи в файл. После того, как нужные данные записаны, файл нужно обязательно **закрыть** с помощью метода `close`:

```
file_obj.write(string)
file_obj.close()
```

Если не закрыть файл, то очень легко его испортить. Хорошим тоном считается закрывать файлы **сразу же**, как только работа с ними прекращена.

Проверить, что запись файла была успешна, можно с помощью вызова утилиты `cat` средствами командной строки (для Unix-подобных операционных систем):

```
!cat file_to_write_in.txt
% OUT: строка для записи в файл
```

Пусть теперь необходимо добавить к файлу еще одну строчку. Если открыть файл и записать вторую строку так, как это делалось раньше:

```
file_obj = open('file_to_write_in.txt', 'w')
second_string = 'вторая строка для записи в файл\n'
file_obj.write(second_string)
file_obj.close()
```

то окажется, что перед записью новой строки все содержимое файла было стерто:

```
!cat file_to_write_in.txt
% OUT: вторая строка для записи в файл
```

В режиме "w", если файл уже существует, его содержимое перезаписывается новым. Чтобы добавить строчку к существующему файлу необходимо использовать режим "a".

```
# Запись 3 строки
file_obj = open('file_to_write_in.txt', 'a')
third_string = 'третья строка для записи в файл\n'
file_obj.write(third_string)
file_obj.close()
```

После выполнения этих команд в конец файла будет добавлена новая строчка:

```
!cat file_to_write_in.txt
% OUT: вторая строка для записи в файл
      третья строка для записи в файл
```

Часто необходимо записать сразу целый список строк в файл. Можно использовать цикл по списку из строк и записывать каждую с помощью функции `write`. Метод `writelines` позволяет сделать то же самое быстрее: достаточно просто передать в качестве первого аргумента список строк и он будет записан в файл.

В следующем примере файл будет открыт немного по-другому. С помощью конструкции `with ... as ...` : удастся избежать проблем с необходимостью помнить о своевременном закрытии файла. Файл автоматически будет закрыт, как только весь код блока `with` будет выполнен.

```
digits = range(1,11)
with open('second_file_to_write_in.txt', 'w') as file_obj:
    file_obj.writelines(digit + '\n' for digit in map(str, digits))
```

Здесь с помощью функции `map` список чисел стал списком строк, а с помощью конструктора — добавлены символы новой строки. Файл, таким образом, будет иметь следующий вид:

```
!cat second_file_to_write_in.txt
% OUT: 1
      2
      3
      4
      5
      6
      7
      8
      9
      10
```

На этом занятия, посвященные работе с файлами, завершены.