OOPython

Лекция 3. Введение в синтаксис языка, часть Б

1. **Функции:**
   a. Одной переменной

```
def Function(x):
        <do_smth>
        return <do_smth2(x)>
```

Отличие от C++: тип возвращаемого значения явно не указывается.
Совет по стилю: имена функций должны начинаться с заглавной буквы.

   b. Области видимости

```
print sum # sum is a built-in Python function
sum = 500 # rebind the name sum to an int
print sum # sum is a global variable
def myfunc(n):          # заголовок функции
        sum = n + 1                         }
        print sum # sum is a local variable } - тело функции

# возврат к глобальной области видимости
return sum
sum = myfunc(2) + 1 # new value in global variable sum
print sum
```

Табуляция разделяет области видимости (аналог “{“ и “}” в C++). Если одним и тем же именем называется несколько объектов из различных областей видимости, то интерпретатор сначала ищет объекты в локальной, затем в глобальной и только затем во встроенных функциях (функция sum).

Для открытия доступа к глобальным переменным внутри локальной области видимости требуется снабдить переменную спецификатором global:

```
a = 20; b = -2.5 # global variables
def f1(x):
        a = 21 # this is a new local variable
```

```
        return a*x + b # 21*x - 2.5

print a

def f2(x):
        global a
        a = 21 # the global a is changed
        return a*x + b # 21*x - 2.5
f1(3); print a # 20 is printed
f2(3); print a # 21 is printed
```

Функции в Python способны возвращать несколько значений. Чтобы сделать это, возвращаемые значения разграничиваются запятой. Формально при этом возвращается **кортеж** - неизменяемый список.

```
def yfunc(t, v0):
        acceleration_of_gravity = 9.81
        vertical_coordinate = v0*time - 0.5*acceleration_of_gravity*(time**2)
        velocity = v0 - acceleration_of_gravity*time
        return vertical_coordinate, velocity
```

position, velocity = yfunc(0.6, 3) # при вызове функции должно использоваться множественное присваивание

Совет по стилю: в языке существует соглашение снабжать функции docstring, сожержащую краткое описание работы функции, аргументы и возвращаемые значения. Опис

```
def Function(x0, y0, x1, y1):
        """
        <doc_string>
        """
        a = (y1 - y0)/float(x1 - x0)
        b = y0 - a*x0
        return a, b
```

Функции в качестве аргументов функций:

Да, в Python подобное возможно воплотить с легкостью. Пример - численное дифференцирование:

```
def SecondDerivativeNumerical(function, x, step=1E-6):
        value = (function(x - step) - 2*function(x) + function(x + step)) /
float(step**2)
```

return value

Лямбда-функции представляют собой неименованные функции, которыми пользуются для экономии места в тексте программы, ими пользуются “на лету”. Определение лямбда-функции: f = **lambda** x: x**2 + 4

Типы передачи аргументов:

- по значению (как в C++). Присуща неизменяемым объектам: числовые типы (int, float, complex), кортежи, строки, ...
- по ссылке (аналог передачи по указателю в C++). Присуща изменяемым объектам: списки, множества, словари.

Пример:

```
>>> def f(a):   # Создается локальная переменная “a”, которая ссылается на переданный объект
...     a = 99  # Создается новый объект “99”, на который теперь ссылается локальная переменная “a”
…
>>> b = 88
>>> f(b)# Первоначально имена a и b ссылаются на одно и то же число 88; в теле функции переменная “a” ссылается на новый объект “99”
>>> print(b)    # Переменная b не изменилась
>>> 88
```

Это является передачей аргумента по значению. Теперь, рассмотрим пример на 2-й способ:

```
def function2arguments(arg1, arg2):
...     arg1 = 5
...     arg2[0] = 'newValue'
...
>>> _arg1 = 6.
>>> _list = ['oldValue', 'secondElement']
>>> function2arguments(_arg1, _list)
>>> print _list
['newValue', 'secondElement']
```

Значение глобальной переменной _list[0] изменилось. Попытка изменения тем же образом кортежа будет вызовет ошибку (исключение):

```
_tuple = tuple(_list)
>>> function2arguments(_arg1, _tuple)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in function2arguments
```

TypeError: 'tuple' object does not support item assignment

2. Условные операторы.

```
if <condition>:
        <do_smth>
elif <condition>:
        <do_smth_else>
else:
        <do_smth_else2>
```

Логическое “и” = “and”
Логическое “или” = “or”

3. **Модули** (синоним библиотек) – программы-скрипты на python (текстовые файлы с расширением **.py**), которые можно включать в код текущей программы. Любая программа на Питоне может быть модулем. Примерно то же самое, что и #include в C++, но включения текста модуля в код не происходит, вместо этого имена объектов (переменные, функции, классы, …) становятся **доступными** в текущей программе.

Связанные модули принято объединять в **пакеты**. **Пакет** представляет собой каталог с файлами-модулями.

Способы включения другой программы **module.py** в качестве модуля:

- import module # доступ к именам как module.object
- from module import object # object
- import module as **mod - псевдоним. # mod.object**

4. NumPy — это расширение языка Python, добавляющее поддержку больших многомерных массивов и матриц, вместе с большой библиотекой высокоуровневых математических функций для операций с этими массивами. Библиотека NumPy предоставляет реализации вычислительных алгоритмов (в виде функций и операторов), оптимизированные для работы с многомерными массивами.

Массив в numpy - вид списка, в котором:

- все элементы должны быть одного типа (int, float, complex, …)
- размер массива должен быть известен изначально (массив статический). Изменение числа элементов запрещено.

Основным объектом NumPy является однородный многомерный массив (в numpy называется numpy.ndarray). Это многомерный массив элементов (обычно чисел), одного типа.

Наиболее важные атрибуты объектов ndarray:

ndarray.ndim - число измерений (чаще их называют "оси") массива.

ndarray.shape - размеры массива, его форма. Это кортеж натуральных чисел, показывающий длину массива по каждой оси. Для матрицы из n строк и m столбов, shape будет (n,m). Число элементов кортежа shape равно ndim.

ndarray.size - количество элементов массива. Очевидно, равно произведению всех элементов атрибута shape.

ndarray.dtype - объект, описывающий тип элементов массива. Можно определить dtype, используя стандартные типы данных Python. NumPy здесь предоставляет целый букет возможностей, как встроенных, например: bool_, character, int8, int16, int32, int64, float8, float16, float32, float64, complex64, object_, так и возможность определить собственные типы данных, в том числе и составные.

ndarray.itemsize - размер каждого элемента массива в байтах.

ndarray.data - буфер, содержащий фактические элементы массива. Обычно не нужно использовать этот атрибут, так как обращаться к элементам массива проще всего с помощью индексов.

Замечание: всё в Питоне - объект, доступ к методам объекта осуществляется через символ **“.”**.

Создание массивов

В NumPy существует много способов создать массив. Один из наиболее простых - создать массив из обычных списков или кортежей Python, используя функцию numpy.array() (запомните: array - функция, создающая объект типа ndarray):

```
>>>
```

```
>>> import numpy as np
>>> a = np.array([1, 2, 3])
>>> a
array([1, 2, 3])
```

```
>>> type(a)
<class 'numpy.ndarray'>
```

Функция array() трансформирует вложенные последовательности в многомерные массивы. Тип элементов массива зависит от типа элементов исходной последовательности (но можно и переопределить его в момент создания).

```
>>>

>>> b = np.array([[1.5, 2, 3], [4, 5, 6]])
>>> b
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

Можно также переопределить тип в момент создания:

```
>>>

>>> b = np.array([[1.5, 2, 3], [4, 5, 6]], dtype=np.complex)
>>> b
array([[ 1.5+0.j,  2.0+0.j,  3.0+0.j],
       [ 4.0+0.j,  5.0+0.j,  6.0+0.j]])
```

Функция array() не единственная функция для создания массивов. Обычно элементы массива вначале неизвестны, а массив, в котором они будут храниться, уже нужен. Поэтому имеется несколько функций для того, чтобы создавать массивы с каким-то исходным содержимым (по умолчанию тип создаваемого массива — float64).

Функция zeros() создает массив из нулей, а функция ones() — массив из единиц. Обе функции принимают кортеж с размерами, и аргумент dtype

```
np.linspace(start, stop, numberOfElements, dtype = 'float16/32/64') #dtype явно задает тип данных в массиве

np.arange(start, stop, step)
np.zeros(number) # создает 1D массив из нулей
np.ones(number) # 1D массив из единиц
A = numpy.array([[1,2,3,4],[5,6,7,8]]) # Creates a 2D array with initialized values
```

```
>>> print A
```

B = numpy.ndarray((2,4)) # Creates a 2D array with uninitialized values; **ndarray = n-dimensional array**

```
>>> B.fill(7) # Fill it in with a constant value 7
>>> print B
[[ 7. 7. 7. 7.]
 [ 7. 7. 7. 7.]]

>>> np.zeros((2, 1))
array([[ 0.], [ 0.]])
```

NumPy **перегружен** для работы с функциями. **Перегрузка функции** (один из видов **полиморфизма**) - определение нескольких функций с одинаковым именем, но различной функциональностью (типами и количеством аргументов, телом функции, возвращаемым значением). То есть в функцию np.nameOfFunction можно передать массивы Numpy в качестве аргументов (посмотреть, что получится):

```
N = 10
listOfArrays = []
listOfArrays.append( np.array([i**2 for i in range(N)]) )
listOfArrays.append( np.array([i**3 for i in range(N)]) )

def Function(x, y):
        “”” return a number using 2 variables “”””
        return np.sin(x) * np.cos(y)

print Function(listOfArrays[0], listOfArrays[1])
```

Свойство выполнять операции над массивами называется **векторизацией** (мы еще коснемся её в конце курса). Без векторизации - никуда в области высокопроизводительных вычислений.

Полную информацию о библиотеке NumPy и ее функциях можно найти на **www.numpy.org.**