Web-программирование
Лекция 5. python 3 (часть 2)

асист. каф. 308 Трутнева Надежда Владимировна

тел: **8-926-880-12-76**

почта: **ntrutn@gmail.com**

Модули

Модули выполняют три важных функции:

- Повторное использование кода: такой код может быть загружен много раз во многих местах.
- Управление адресным пространством: модуль — это высокоуровневая организация программ, это пакет имен, который избавляет вас от конфликтов. Каждый объект «проживает» свой цикл внутри своего модуля, поэтому модуль — это средство для группировки системных компонентов.
- Глобализация сервисов и данных: для реализации объекта, который используется во многих местах, достаточно написать один модуль, который будет импортирован.

Модули

Модуль — файл, в котором находятся классы, функции или данные, которые можно использовать в других программах.

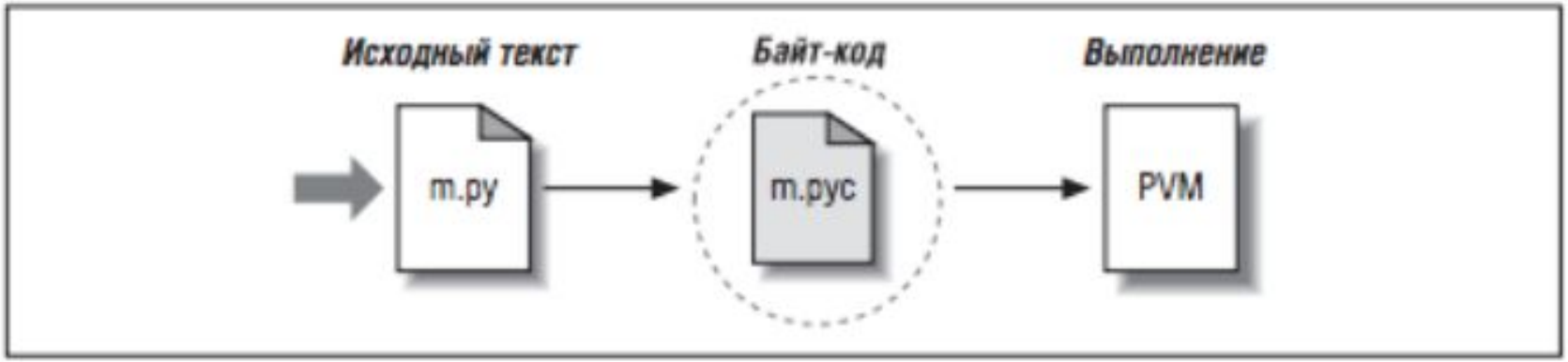
Объекты из модуля могут быть импортированы в другие модули.

Имя файла образуется путем добавления к имени модуля расширения .py.

При импорте модуля интерпретатор ищет файл с именем my_module.py:
1. в текущем каталоге,
2. в каталогах, указанных в переменной окружения PYTHONPATH
3. в зависящих от платформы путях по умолчанию, а также в специальных файлах с расширением '.pth', которые лежат в стандартных каталогах.

Программист может внести изменения в PYTHONPATH и в '.pth', добавив туда свой путь. Каталоги, в которых осуществляется поиск, можно посмотреть в переменной sys.path.

Модули



Для ускорения запуска программ, использующих большое количество модулей, если уже существует файл с именем my_module.pyc в том же каталоге, где найден my_module.py, считается, что он содержит байт-компилированный модуль my_module. Если такого файла нет, то он создается, и время последнего изменения my_module.py записывается в созданном my_module.pyc.

Содержимое байт-компилированных файлов является платформенно-независимым (но может быть разным для разных версий интерпретатора), так что каталог с модулями может совместно использоваться машинами с разными архитектурами.

Модули

Каждый модуль имеет собственное пространство имен, являющееся глобальной областью видимости для всех определенных в нем функций.

Если в модуле определена переменная __all__ (список атрибутов, которые могут быть подключены), то будут подключены только атрибуты из этого списка. Если переменная __all__ не определена, то будут подключены все атрибуты, не начинающиеся с нижнего подчёркивания.

Для того чтобы переменные этого модуля не попали в конфликт с другими глобальными именами или другими модулями, нужно использовать префикс: **_имя_модуля_._имя_переменной_** .

Модули могут импортировать другие модули. Обычно инструкцию import располагают в начале модуля или программы.

Модули (импорт стандартных модулей)

```
import os
os.getcwd()


>>> import time, random
>>> time.time()
1376047104.056417
>>> random.random()
0.9874550833306869

>>> import math     # название модуля становится
переменной
>>> math.e          # доступ к атрибутам модуля
2.718281828459045


>>> import math as m     # m - псевдоним
>>> m.e
2.718281828459045
```

Модули

```
>>> import notexist
Traceback (most recent call last):
  File "", line 1, in
    import notexist
ImportError: No module named 'notexist'
>>> import math
>>> math.Ë
Traceback (most recent call last):
  File "", line 1, in
    math.Ë
AttributeError: 'module' object has no attribute 'Ë'
```

Инструкция **from**

```
from <Название модуля> import <Атрибут 1> [ as
<Псевдоним 1> ], [<Атрибут 2> [ as <Псевдоним 2> ] …]
```

```
>>> from math import e, ceil as c
>>> e
2.718281828459045
>>> c(4.6)
5
```

```
>>> from math import (sin, cos,
...                   tan, atan)
```

Инструкция **from**

```
from <Название модуля> import *
```

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

```
>>> from sys import *
>>> version
'3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43) [MSC
v.1600 32 bit (Intel)]'

>>> version_info
sys.version_info(major=3, minor=3, micro=2,
releaselevel='final', serial=0)
```

Инструкция **from**

глобальная или локальная переменная

???

```
>>> from small import x, y
>>> x = 42
```

```
>>> import small
>>> small.x = 42
```

Модули

mymodule.py

```
def hello():
    print('Hello, world!')

def fib(n):
    a = b = 1
    for i in range(n - 2):
        a, b = b, a + b
    return b

# если модуль запущен как скрипт
if __name__ == "__main__":
    hello()
    for i in range(10):
        print(fib(i))
```

Пакеты

Пакеты — способ структурирования пространств имен модулей на основе файловой системы.

Так же, как применение модулей делает безопасным использование глобального пространства имен авторами различных модулей, применение пакетов делает безопасным использование имен модулей авторами многомодульных пакетов.

```
TCP/                    каталог, в котором лежит пакет
  _init_.py             по этому файлу интерпретатор распознает, что в каталоге лежит пакет
  main.py

  Server/
      _init_.py
      tcp.py
      server.py
      lib.py
  Client/
      _init_.py
      tcp.py
      client.py
      lib.py
```

Пакеты

Импорт индивидуальных модулей из пакета
>>> import TCP.Server.lib
>>> import TCP.Client.lib

Тогда вызов функции connect() выглядит так:
>>> import TCP.Server.lib.connect()

ИЛИ

>>> from TCP.Server import lib as server_lib

вместо lib может быть подставлен модуль, подпакет или имя, определенное в TCP.Server (функция, класс, переменная)

>>> from TCP.Client import lib as client_lib
>>> server_lib.connect()
>>> client_lib.connect()

Пакеты

```
>>> from TCP import *
```

__init__.py

```
__all__ = ["Server", "Client"]
```

Функции

Функция в python - объект, принимающий аргументы и возвращающий значение.
Обычно функция определяется с помощью инструкции def.

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

Вызов функции:

```
>>> add(1, 10)
11
>>> add('abc', 'def')
'abcdef'
```

Функции

Функция может возвращать любые объекты — списки, кортежи, функции.

```
def newfunc(n):
    print "n = ", n
    def myfunc(x):
        print "x = ", x
        return x+n
    return myfunc

new = newfunc(100)

a = new(200)
print a
```

Функции

Функция может и не заканчиваться инструкцией return, при этом функция вернет значение None:

```
>>> def func():
...     pass
...
>>> print(func())
None
```

Аргументы функции

```
>>> def func(a, b, c=2): # c - необязательный аргумент
...     return a + b + c
...
>>> func(1, 2)  # a = 1, b = 2, c = 2 (по умолчанию)
5
>>> func(1, 2, 3)  # a = 1, b = 2, c = 3
6
>>> func(a=1, b=3)  # a = 1, b = 3, c = 2
6
>>> func(a=3, c=6)  # a = 3, c = 6, b не определен
Traceback (most recent call last):
  File "", line 1, in
    func(a=3, c=6)
TypeError: func() takes at least 2 arguments (2 given)
```

Аргументы функции (переменное количество аргументов)

```
>>> def func(*args):
...     return args
...
>>> func(1, 2, 3, 'abc')
(1, 2, 3, 'abc')
>>> func()
()
>>> func(1)
(1,)
```

Аргументы функции (произвольное количество аргументов)

```
>>> def func(**kwargs):
...     return kwargs
...
>>> func(a=1, b=2, c=3)
{'a': 1, 'c': 3, 'b': 2}
>>> func()
{}
>>> func(a='python')
{'a': 'python'}
```

Анонимные функции

```
>>> func = lambda x, y: x + y
>>> func(1, 2)
3
>>> func('a', 'b')
'ab'
>>> (lambda x, y: x + y)(1, 2)
3
>>> (lambda x, y: x + y)('a', 'b')
'ab'
```

```
>>> func = lambda *args: args
>>> func(1, 2, 3, 4)
(1, 2, 3, 4)
```

Классы и объекты

Объектно-ориентированное программирование (ООП) — парадигма программирования, в которой основными концепциями являются понятия объектов и классов.

Класс — тип, описывающий устройство объектов. **Объект** — это экземпляр класса.

В Python всё является объектами.

Объектно-ориентированный подход в программировании подразумевает следующий алгоритм действий:

- Описывается проблема с помощью обычного языка с использованием понятий, действий, прилагательных.
- На основе понятий формулируются классы.
- На основе действий проектируются методы.
- Реализуются методы и атрибуты.

Классы и объекты

- **Полиморфизм:** в разных объектах одна и та же операция может выполнять различные функции. Простым примером полиморфизма может служить функция count(), выполняющая одинаковое действие для различных типов обьектов: 'abc'.count('a') и [1, 2, 'a'].count('a'). Оператор плюс полиморфичен при сложении чисел и при сложении строк.
- **Инкапсуляция:** можно скрыть ненужные внутренние подробности работы объекта от окружающего мира. Этот принцип основан на использовании атрибутов внутри класса. Атрибуты могут иметь различные состояния в промежутках между вызовами методов класса, вследствие чего сам объект данного класса также получает различные состояния.
- **Наследование:** можно создавать специализированные классы на основе базовых. Это позволяет нам избегать написания повторного кода.
- **Композиция:** объект может быть составным и включать в себя другие объекты.

Классы и объекты

Наиболее важные особенности классов в Python:

- множественное наследование;
- производный класс может переопределить любые методы базовых классов;
- в любом месте можно вызвать метод с тем же именем базового класса;
- все атрибуты класса в Python по умолчанию являются public, т.е. доступны отовсюду; все методы — виртуальные, т.е. перегружают базовые.

Классы и объекты

```
class имя_класса:
    инструкция 1
    .
    инструкция №
```

Каждая такая запись генерирует свой объект класса.
В Python описание класса - это создание объекта (в C++ объявление).

Есть также другой тип объекта Python — экземпляр класса, который генерируется при вызове:

```
экземпляр_класса = имя_класса()
```

Объект класса и экземпляр класса — это два разных объекта. Первый генерируется на этапе объявления класса, второй — при вызове имени класса. Объект класса может быть один, экземпляров класса может быть много.

Атрибуты класса

Атрибуты класса бывают двух видов:

- атрибуты данных;
- атрибуты-методы.

Память для атрибутов выделяется в момент их первого присваивания — либо снаружи, либо внутри метода. Методы начинаются со служебного слова **def**.

Доступ к атрибутам выполняется по схеме **obj.attrname**.

```
class Simple:
    u'Простой класс'
    var = 87
    def f(x):
        return 'Hello world'

>>> print Simple.var.__doc__     #вызов стандартных атрибутов
int(x[, base]) -> integer
...
```

Атрибуты класса

Создание экземпляра класса:

```
smpl = Simple()
```

Будет создан пустой объект smpl. Если мы хотим, чтобы при создании выполнялись какие-то действия, нужно определить конструктор, который будет вызываться автоматически:

```
class Simple:
    def __init__(self):
        self.list = []
```

Конструктору можно передать аргументы:

```
class Simple:
    def __init__(self, count, str):
        self.list = []
        self.count = count
        self.str = str
```

Атрибуты класса

Атрибут данных можно сделать приватным (private) — т.е. недоступным снаружи — для этого слева нужно поставить два символа подчеркивания:

```
class Simple:
    u'Простой класс с приватным атрибутом'
    __private_attr = 10
    def __init__(self, count, str):
        self.__private_attr = 20
        print self.__private_attr

s = Simple(1,'22')
print s.__private_attr
```

Последняя строка вызовет исключение — атрибут __private_attr годен только для внутреннего использования.

Классы

Методы необязательно определять внутри тела класса:

```
def method_for_simple(self, x, y):
    return x + y
class Simple:
    f = method_for_simple

>>> s = Simple()
>>> print s.f(1,2)
3
```

Пустой класс можно использовать в качестве заготовки для структуры данных:

```
class Customer:
    pass
custom = Customer()
custom.name = 'Вася'
```

Классы (self)

Обычно первый аргумент в имени метода — self.

```
class Simple:
    def __init__(self):
        self.list = []
    def f1(self):
        self.list.append(123)
    def f2(self):
        self.f1()

>>> s = Simple()
>>> s.f2()
>>> print s.list
[123]
```

Self — это аналог "this" в C++.

Наследование

Определение производного класса выглядит следующим образом:

class Derived(Base):

Если базовый класс определен не в текущем модуле:

class Derived(module_name.Base):

Разрешение имен атрибутов работает сверху вниз: если атрибут не найден в текущем классе, поиск продолжается в базовом классе, и так далее по рекурсии. Производные классы могут переопределить методы базовых классов — все методы являются в этом смысле виртуальными. Вызвать метод базового класса можно с префиксом:

Base.method()

Наследование

В Python существует ограниченная поддержка множественного наследования:

class Derived(Base1,Base2,Base3):

Поиск атрибута производится в следующем порядке:

- в Derived;
- в Base1, затем рекурсивно в базовых классах Base1;
- в Base2, затем рекурсивно в базовых классах Base2
- и т.д.

Пример

```
# -*- coding: utf-8 -*-
class Person:
	def __init__(self, name, job=None, pay=0):
		self.name = name
		self.job = job
		self.pay = pay
	def lastName(self):
		return self.name.split()[-1]
	def giveRaise(self, percent):
		self.pay = int(self.pay * (1 + percent))
	def __str__(self):
		return '[Person: %s, %s]' % (self.name, self.pay)

class Manager(Person):
	def __init__(self, name, pay):
		Person.__init__(self, name, 'mgr', pay)
	def giveRaise(self, percent, bonus=100):
		Person.giveRaise(self, percent + bonus)
```

Пример

```
>>> ivan = Person('Иван Petrov')
>>> john = Person('John Sidorov', job='dev', pay=100000)
```

Вызываем перегруженную функцию __str__;

```
>>> print(ivan)
>>> print(john)
```

Выводим фамилию:

```
>>> print(ivan.lastName(), john.lastName())
```

Начисляем премиальные:

```
>>> john.giveRaise(.10)
```

Создаем инстанс класса Manager:

```
>>> tom = Manager('Tom Jones', 50000)
```

Начисляем мегапремиальные:

```
>>> tom.giveRaise(.10)
```

Спасибо за внимание !

ВОПРОСЫ???