Занятие 3: Функции. Модули. Классы.

Практикум на ЭВМ 2017/2018

Попов Артём Сергеевич

МГУ имени М. В. Ломоносова, факультет ВМК, кафедра ММП

Организация кода

Функции, модули, классы устраняют необходимость вставлять в программу избыточные копии блоков одинакового кода.

Грамотная организация кода позволяет:

- Максимизировать многократное использование кода
- Минимизировать избыточность кода
- Упростить отладку кода
- Улучшить читаемость кода
- Улучшить логику кода

Синтаксис

Определение функции:

```
>>> def f(x):
...     """This function does nothing"""
...     return x
```

У аргументов функции могут быть значения по умолчанию:
(такие аргументы всегда идут в конце)

```
>>> def f(x, y, z=0):
...     return x + y * z
```

При вызове функции могут использоваться именованные аргументы (всегда идут за позиционными):

```
>>> f(1, 2)
>>> f(1, z=2, y=1)
```

Методы функции

Всё в Python — объект, функция тоже объект!

У функции есть некоторые атрибуты:

```
>>> f.__doc__
"This function does nothing"
>>> f.__dict__ # список атрибутов функции
{}
```

Функция со счётчиком запусков:

```
>>> def f(x):
>>>     """This function does nothing"""
...     if 'f_count' in f.__dict__:
...         f.f_count += 1
...     else:
...         f.f_count = 1
...     return x
```

Значения по умолчанию: проблемы

Не нужно использовать изменяемые объекты в качестве значений по умолчанию:

```
>>> def function(list_of_items, my_set=set()):
...     final_list = list()
...     for item in list_of_items:
...         if item not in my_set:
...             my_set.add(item)
...             final_list.append(item)
...     return final_list
...
>>> my_list = [1, 2, 3]
>>> function(my_list)
[1, 2, 3]
>>> function(my_list)
[]
```

Функции с переменным числом аргументов

У функции может быть произвольное число неименованных и именованных аргументов:

```
>>> def f(*args, **kwargs):
...     if 'x' in kwargs:
...         print(kwargs['x'])
...     print(args)
...     print(kwargs)
...
>>> f(3, 1, 7, x=1, y=8)
1
(3, 1, 7)
{'x': 1, 'y': 8}
```

Пример: функция минимума

Функция минимума из двух элементов:

```
>>> def min(x, y):
...     return x if x < y else y
```

Функция минимума для произвольного числа аргументов:

```
>>> def min(*args):
...     res = float('inf')
...     for arg in args:
...         if arg < res:
...             res = arg
...     return res
```

У такой реализации есть проблема:

```
>>> min()
inf
```

Ограничения на аргументы

Более правильная реализация с обязательным аргументом:

```
>>> def min(first, *args):
...     res = first
...     for arg in args:
...         if arg < res:
...             res = arg
...     return res
```

Аргументы могу быть строго именованными:

```
>>> def f(x, *args, age):
...     if age < 18:
...         return 0
...     return x + sum(args)
...
>>> f(1, 2, 20)
TypeError:
f() missing 1 required keyword-only argument: 'age'
```

Преобразование контейнеров в аргументы

`*` перед контейнером преобразует его в аргументы функции

```
>>> my_list = [1, 2, 3]
>>> # две эквивалентные записи
>>> f(my_list[0], my_list[1], my_list[2])
>>> f(*my_list)
```

`zip()` соединяет несколько контейнеров в контейнер кортежей:

```
>>> your_list = ['a', 'b', 'c']
>>> t = list(zip(my_list, your_list))
>>> t
[(1, 'a'), (2, 'b'), (3, 'c')]
```

Обратное преобразование с использованием `*`:

```
>>> list(zip(*t))
[(1, 2, 3), ('a', 'b', 'c')]
```

Области видимости

Область видимости — место, где определяются переменные и выполняется их поиск.

Правило LEGB — последовательный поиск имени в 4 областях:

- L — local (все имена, которым присваиваются значения внутри функции)
- E — enclosing (локальные области объемлющих функций)
- G — global (имена на верхнем уровне модуля)
- B — built-in (встроенные имена)

Явное изменение области видимости имени (L на E):

```
>>> def f():
...     nonlocal y      # global y (L на G)
...     y += 1
```

Пример работы LEGB

```
>>> x = 50
>>> y = 1
>>> def enc_func(x):
...     y = 30
...     def func(y):
...         nonlocal x
...         print(x + y)
...     func(11)
...
>>> enc_func(5)
16
```

Для присваиваний LEGB не работает

Операция присваивания по умолчанию создаёт локальную переменную:

```
>>> x = 17
>>> def f():
...     x += 5
UnboundLocalError: local variable 'x' referenced before
assignment
```

Синтаксис

Модуль — файл с расширением .py

Оператор `import` импортирует указанный модуль и создаёт на него ссылку в текущей области видимости:

```
>>> import my_module
>>> my_module.function_in_module()
```

С помощью оператора `as` можно изменить имя переменной, в которую будет записана ссылка на модуль:

```
>>> import my_module as mm
```

С помощью связки `from ... import` можно импортировать имя из другого модуля в текущую область видимости:

```
>>> from my_module import function_in_module
```

Ещё о модулях

Если в втором аргументе `from ... import` указать *, импортируются все глобальные имена модуля или все переменные, записанные в `__all__`

У модуля есть некоторые атрибуты:

```
>>> import my_module
>>> my_module.__name__
'my_module'
>>> my_module.__file__
'/home/user/Programs/my_module.py'
```

У модулей, переданных на выполнение интерпретатору, `__name__ == __main__`

Импортирование модуля

Что происходит при импортировании модуля:

1. Поиск файла модуля
   (по директориям, находящимся в переменной `sys.path`)
2. Компиляция в байт-код
   (если нет актуальной скомпилированной версии модуля)
3. Запуск программного кода модуля

Если в модуле что-то изменилось, его необходимо перегрузить:

```
>>> import imp
>>> imp.reload(my_module)
```

Когда __name__ == __main__

Модуль может содержать не только функции или классы, но и исполняющие их инструкции.
Чтобы эта часть кода не выполнялась при импорте, её заключают в специальный блок:

```
def sin(x):
    return x - x ** 3 / 6

if __name__ == '__main__':
    print('test my method')
    print(sin(3.14))
    print(sin(0))
    print('end of test')
```

Синтаксис

Объявление класса:

```
>>> class MyClass:
...     def __init__(self, value):     # конструктор класса
...         self.value = value
...
...     def get_value(self):
...         return self.value
```

При вызове класса порождается его экземпляр:

```
>>> x = MyClass()
```

Каждый объект экземпляра наследует атрибуты класса и приобретает свое собственное пространство имен

Атрибуты экземпляров класса

Атрибут класса $\neq$ атрибут экземпляра

Атрибуты экземпляра добавляются к экземпляру с помощью использования self или прямым присваиванием к экземпляру:

```
>>> class MyClass:
...     def __init__(self, value):
...         self.value = value
...
>>> new_obj = MyClass(2)
>>> new_obj.obj_attr = -2
```

value и obj_attr — атрибуты экземпляра класса

Атрибуты класса

Атрибут класса $\neq$ атрибут экземпляра

Атрибуты класса объявляются в теле класса или прямым присваиванием к классу. Изменение атрибута класса отразится на всех его экземплярах:

```
>>> class MyClass:
...     class_attr = 0
...     def __init__(self, value):
...         self.value = value
...
>>> new_obj = MyClass()
>>> MyClass.second_class_attr = '127'
>>> new_obj.second_class_attr
'127'
```

`class_attr` и `second_class_attr` — атрибуты класса

Связанные и несвязанные методы

У связанного метода первый аргумент — соответствующий экземпляр класса:

```
>>> class MyClass:
...     def my_method(self):
...         print('Do nothing')
...
>>> x = MyClass()
>>> x.my_mehtod() # связанный метод
'Do nothing'
```

Несвязанному методу необходимо явно передать первым аргументом экземпляр класса:

```
>>> the_same_method = MyClass.my_method # несвязанный метод
>>> the_same_method()
TypeError: my_method() missing 1 required positional
argument: 'self'
>>> the_same_method(x)
'Do nothing'
```

Внутренние атрибуты классов и экземпляров

```
>>> class MyClass:
...     """Documentaion"""
...     def __init__(self, value):
...         self.value = value
...
...     def one_method():
...         print('Do nothing')
...
>>> x = MyClass(17)
>>> MyClass.__name__ # имя
'MyClass'
>>> MyClass.__doc__ # документация
'Documentaion'
>>> x.__dict__ # атрибуты экземпляра
{'value': 17}
```

ООП в Python

Классы — основной инструмент ООП в Python

Основные принципы ООП (wiki):

- Полиморфизм — свойство системы, позволяющее использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта
- Инкапсуляция — свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и ограничить к ним доступ.
- Наследование — свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью

Очевидно, что полиморфизм в Python есть :)

Наследование

Класс может наследовать другие классы:

```
>>> class Soldier:
...     def run(self):
...         print('run run')
...
...     def shoot(self):
...         print('shoot shoot')
...
>>> class Captain(Soldier):
...     def command(self, other_soldier):
...         other_soldier.run()
...
>>> s1 = Soldier()
>>> s2 = Captain()
>>> s2.command(s1)
'run run'
>>> s2.shoot()
'shoot shoot'
```

Разрешение конфликтов имён

При обращении к атрибуту/методу экземпляра поиск ведётся:

1. В самом экземпляре
2. В классе экземпляра
3. По иерархии наследования

При множественном наследовании Python использует алогритм линеаризации C3 для определения вызываемого метода.
Получить порядок:

```
>>> C.__mro__
__main__.C, __main__.A, __main__.B, object
```

Пример конфликтной ситуации

```
>>> class A:
...     def get_attr(self):
...         return 'a'
...
>>> class B:
...     def get_attr(self):
...         return 'b'
...
>>> class C(A, B):
...     pass
...
>>> obj = C()
>>> obj.get_attr()
'a'
```

Перегрузка конструктора наследника

Если у класса-наследника определён метод `__init__`, конструкторы родителей необходимо вызывать явно:

```
class Soldier:
    def __init__(self, speed=0):
        self.speed = speed

    def run(self):
        print('run ' * self.speed)

class Captain(Soldier):
    def __init__(self, speed=0, wisdom=0):
        Soldier.__init__(self, speed)
        self.wisdom = wisdom

    def command(self, other_soldier):
        for i in range(self.wisdom):
            other_soldier.run()
```

Перегрузка конструктора наследника

Другая реализация через `super()` (позволяет вызывать методы предков):

```
class Soldier:
    def __init__(self, speed=0):
        self.speed = speed

    def run(self):
        print('run ' * self.speed)

class Captain(Soldier):
    def __init__(self, speed=0, wisdom=0):
        super().__init__(speed)
        self.wisdom = wisdom

    def command(self, other_soldier):
        for i in range(self.wisdom):
            other_soldier.run()
```

Классы-примеси (mixin)

Если известно, что класс реализует некоторый интерфейс, можно использовать классы-примеси для его модификации:

```
>>> def get_speed(self):
...     return self.speed
>>> Soldier.get_speed = get_speed
...
>>> class Speed_calculations_mixin:
...     def get_distance(self, time_sec):
...         return super().get_speed() * time_sec
...
...     def get_time(self, distance):
...         return distance / super().get_speed()
...
>>> class Major(Speed_calculations_mixin, Captain):
...     pass
```

Перегрузка операторов

Перегрузка операторов реализуется за счёт написания *специальных методов* (`__method__`)

Некоторые из них:

- `__init__` — вызывается при создании нового экземпляра
- `__add__` — вызывается при сложении экземпляров
- `__repr__` — вызывается при выводе объекта
- `__getattr__` — вызывается при попытке прочитать значение несуществующего атрибута
- `__getitem__` — взятие элемента по индексу

Полный список:
https://docs.python.org/3/reference/datamodel.html#special-method-names

Пример перегрузки операторов

```
>>> class vector(list):
...     def __init__(self, some_list):
...         self.values = some_list
...
...     def __add__(self, other_vector):
...         if len(self.values) == len(other_vector.values):
...             new_vector = vector([])
...             for elem in zip(self.values,
...                             other_vector.values):
...                 new_vector.values.append(elem[0] + elem[1])
...             return new_vector
...         else:
...             raise TypeError('Wrong dimensions')
...
...     def __getitem__(self, i):
...         if 0 <= i <= len(self.values):
...             return self.values[i]
```

Сокрытие атрибутов

В Python нет модификторов доступа к атрибутам и методам

К внутренним атрибутам принято добавлять в начале символ подчёркивания:

```
>>> class MyClass:
...     _important_attr = 32
```

При добавлении двух подчёркиваний атрибут будет автоматически получать более сложное название:

```
>>> class MyClass:
...     __very_important_attr = 32
...
>>> x = MyClass()
>>> x. __very_important_attr
AttributeError: 'MyClass' object has no attribute ...
>>> x._MyClass__very_important_attr
32
```

Сокрытие атрибутов — перегрузка операторов

Для ограничения доступа к атрибутам можно перегрузить операторы `__setattr__` и `__getattr__` :

```
>>> class A:
...     def __init__(self, value):
...         self.value = value
...
...     def __setattr__(self, name, value):
...         print('set atrribute', value)
...         self.__dict__[name] = value # словарь атрибутов
...
...     def __getattr__(self, name):
...         print('wrong attr')
...
>>> a = A(123)
'set attribute 123'
>>> a.some_attr
'wrong attribute'
```

Свойства

Протокол свойств позволяет направлять операции чтения, записи и удаления для отдельных атрибутов отдельным функциям и методам

```
attribute = property(fget, fset, fdel, doc)
```

- `fget` — функция, которая будет вызываться при попытке прочитать значение атрибута
- `fset` — функция, которая будет вызываться при попытке выполнить операцию присваивания
- `fdel` — функция, которая будет вызываться при попытке удалить атрибут.
- `doc` — передается строка документирования с описанием атрибута

Сокрытие атрибутов — свойства

```
>>> class A:
...     def __init__(self, value):
...         self._value = value
...
...     def get_val(self):
...         return self._value
...
...     def set_val(self, value):
...         print('set atribute', value)
...         self._value = value
...
...     value = property(get_val, set_val, None, "no doc")
...
>>> a = A(123)
>>> a.value = -1
'set atribute -1'
```

Атрибут __slots__

С помощью атрибута __slots__ можно зафиксировать множество возможных атрибутов:

```
>>> class Point:
...     __slots__ = ['x', 'y']
...
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...
...     def get_r(self):
...         return (self.x ** 2 + self.y ** 2) ** 0.5
...
>>> m = Point(3, 4)
>>> m.z = 5
AttributeError: 'Point' object has no attribute 'z'
```

Экземпляры с __slots__ требуют меньше памяти (если в __slots__ нет __dict__)

Классы в scikit-learn

Основные сущности, с которыми мы будем работать:

- Конкретное семейство алгоритмов — класс
- Алгоритм из семейства с заданными параметрами — экземпляр класса
- Алгоритм обучения — метод класса (`.fit`)
- Получение предсказаний — метод класса (`.predict`)

Какие преимущества у класса перед набором функций для реализации алгоритма?

Заключение

Несколько важных замечаний:

- Функции, модули и классы являются объектами в Python
- У функции может быть несколько аргументов
- Модули выполняются при импортировании!
- Класс и объект класса не одно и то же!
- Скрывать атрибуты класса возможно