Функции

Дмитрий Яковлев

Санкт-Петербургский государственный университет

17 марта 2022 г.

План лекции

Функции

Синтаксис

Основные инструкции:

- `def` - создаёт объект и присваивает ему имя
- `lambda` - создаёт объект и возвращает его в виде результата
- `return` - возвращает результат (при отсутствии возвращает по умолчанию - `None`)

Формат:

```
def name(arg1, ...):
    ...
    return value
```

```
def adder(c):
    f = lambda x: c + x
    return f
```

```
>>> f = adder("GCTTA_")
>>> f("GAATCCAGCACGGAC")
'GCTTA_GAATCCAGCACGGAC'
```

Байткод функции

```
def hello():
    print("Hello world!")
hello()
```

```
python3 -m dis .\hello.py
  1           0 LOAD_CONST               0 (<code ...>)
              2 LOAD_CONST               1 ('hello')
              4 MAKE_FUNCTION            0
              6 STORE_NAME               0 (hello)

  3           8 LOAD_NAME                0 (hello)
             10 CALL_FUNCTION            0
             12 POP_TOP
             14 LOAD_CONST               2 (None)
             16 RETURN_VALUE
```

Замечание про функции

Функции - самые обычные объекты:

```
>>> add = adder
>>> f = add(3)
>>> f(5)
8
```

Стоит помнить, что в Python **нет понятия компиляции**. Функции записываются в память во время исполнения.

```
>>> dir()
['__builtins__', ..., '__spec__']
>>> def f():
...     return 5
>>> dir()
['__builtins__', ..., '__spec__', 'f']
```

Документация функций

Через атрибут **__doc__** можно получить документацию к функции:

```
>>> def f(a, b):
...     """Do nothing"""
...     pass
>>> f.__doc__
'Do nothing'
>>> f.__name__
'f'
```

При отсутствии `return`, функции возвращает `None`

```
>>> f(24, 21)
None
```

Документация функции

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout,
     flush=False)

    Prints the values to a stream, or to sys.stdout
    by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to
     current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value,
    default a newline.
    flush: whether to forcibly flush the stream.
```

Передача аргументов (1)

Аргументы функции передаются по **ссылкам**, но:

- **Атомарные** объекты (числа и строки) передаются **«по значению»**

```
>>> x = 5
>>> def f(x):
...     x = 6
>>> f(x)
>>> x
5
```

- **Изменяемые** объекты передаются **«по указателю»**

```
>>> seq = ['A', 'G', 'C']
>>> def f(seq):
...     seq[0] = 'G'
>>> f(seq)
>>> seq
['G', 'G', 'C']
```

Передача аргументов (2)

Что будет выведено на экран?

```
>>> x = [3, 4]
>>> def f(x):
...         x[0] = 1
...         x = [3, 4]
>>> f(x)
>>> y = x
>>> y.append(5)
>>> y = 6
>>> x
???
```

Передача аргументов (2)

Что будет выведено на экран?

```
>>> x = [3, 4] # создали ссылку на новый объект - список
>>> def f(x):
...         x[0] = 1 # изменяем объект, лежащий по ссылке
...         x = [3, 4] # локальный x - ссылка на новый объект
>>> f(x) # x = [1, 4]
>>> y = x # создаём вторую ссылку на объект x
>>> y.append(5) # x = [1, 4, 5]
>>> y = 6 # создали новый объект 6 и y ссылается на него
>>> x
[1, 4, 5]
```

В Python нет **const**. Как быть?

const?

Способы избежать воздействия на изменяемые объекты:

- Передавать **копию**

```
>>> x = [3, 4]
>>> f(x[:])
```

- Создавать копию объекта внутри функции

```
>>> def f(x):
...         x = x[:]
```

- С помощью **декораторов** - об этом позже
- Передавать **кортеж** - при изменении нужно будет ловить исключения

```
>>> f(tuple(x))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
TypeError: 'tuple' object does not support ...
```

Сопоставление аргументов

Режимы сопоставлений:

- По позиции - `def f(x, y, z)`
- По именам (значение по умолчанию) - `def f(x=0)`
- Произвольное количество аргументов - `def f(*args)`
- Только именованные аргументы - `def f(*, x=0)`
- Только по позиции (нельзя по имени) - `def f(x, y, /)`

Сопоставление аргументов: по позиции

Всё просто - значения и имена аргументов ставятся в соответствие в порядке их следования **слева направо**.

```
>>> def sum(x, y, z):
...     return x + y + z
...
>>> sum(3, 4, 5)
12
>>> sum(3, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sum() missing 1 required positional argument: z
```

Сопоставление аргументов: по именам

Соответствие определяется за счёт **имён аргументов**. Если аргумент не передаётся, то значение **по умолчанию**.

```
>>> def sum(x, y, z=0):
...     return x + y + z
>>> sum(3, 4)
7
>>> sum(3, 4, 5)
12
>>> sum(3, 4, z=5)
12
>>> sum(3, y=4)
7
```

После ключевых аргументов не могут следовать не ключевые.

```
>>> def sum(x=0, y, z):
...     return x + y + z
  File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
```

Параметры по умолчанию (1)

Что будет выведено на экран?

```
>>> def f(x, l=[]):
...     l.append(x)
...     return l
...
>>> f(1)
???
>>> f(2)
???
```

Параметры по умолчанию (2)

```
def f(x, l=[]):
    l.append(x)
    return l
f(5)
```

```
1           0 BUILD_LIST               0
            2 BUILD_TUPLE              1
            4 LOAD_CONST               0 (<code object f..>)
            6 LOAD_CONST               1 ('f')
            8 MAKE_FUNCTION            1 (defaults)
           10 STORE_NAME               0 (f)

4          12 LOAD_NAME                0 (f)
           14 LOAD_CONST               2 (5)
           16 CALL_FUNCTION            1
           18 POP_TOP
           20 LOAD_CONST               3 (None)
           22 RETURN_VALUE
```

Параметры по умолчанию (3)

```
>>> def f(x, l=[]):
...     l.append(x)
...     return l
...
>>> f.__defaults__
([],)
>>> f(1)
[1]
>>> f(2)
[1, 2]
>>> f.__defaults__
([1, 2],)
```

Как решить данную проблему?

Параметры по умолчанию (4)

Значения по умолчанию инициализируется лишь **однажды** - во время интерпретации в байт код.

```
>>> def f(x, l=[]):
...     l.append(x)
...     return l
...
>>> f(1)
[1]
>>> f(2)
[1, 2]
```

```
>>> def f(x, l=None):
...     l = list(l or [])
...     l.append(x)
...     return l
>>> f(1)
[1]
>>> f(2)
[2]
```

`or` - если первое `False`, то возвращает второй аргумент

Упаковка

Наличие символа * перед именем аргумента объединяет произвольное количество аргументов в кортеж.

```
>>> def sum(*args):
...     res = 0
...     for x in args:
...         res += x
...     return res
>>> sum(0, 1, 2, 3, 4, 5)
15
>>> sum()
0
```

Пусть у нас имеется коллекция (например список). Как его передать в функцию?

Распаковка (1)

Для распаковки коллекций существует аналогичный синтаксис с символом *.

```
>>> sum(*[1, 2, 3])
6
>>> sum(*{1, 2, 3})
6
>>> sum(*(1, 2, 3))
6
```

Что нужно сделать, чтобы работал для всех объектов, которые можно складывать?

```
>>> sum(*{"A", "G", "C", "T"})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in sum
TypeError: unsupported type(s) for +=: 'int' and 'str'
```

Распаковка (2)

Расширим нашу функцию для всех объектов, которые можно складывать:

```
>>> def sum(first, *args):
...     res = first
...     for x in args:
...         res += x
...     return res
...
>>> sum(*{"A", "G", "C", "T"})
'CATG'
>>> sum(*[1, 2, 3, 4])
10
>>> sum([1, 2, 3, 4])
[1, 2, 3, 4]
```

Распаковка (3)

Примеры использования распаковки:

```
>>> x, y, z = [1, 2, 3]
>>> first, *middle, last = range(100)
>>> first
0
>>> last
99
>>> middle
[1, ..., 98]
```

Аналогично можно использовать в циклах:

```
>>> for _, *y in [range(3), range(4), range(5)]:
...     print(y)
[1, 2]
[1, 2, 3]
[1, 2, 3, 4]
```

Распаковка (4)

Примеры использования распаковки:

Данные:

```
# comp_id, compound, score
# 5 C1CCCCC1 5.23
# 7 CN=C=O 1.75
```

Плохо

```
for line in data:
    item = line.split()
    save(item[0], float(item[2]))
```

Хорошо

```
for line in data:
    comp_id, _, score = line.split()
    save(comp_id, float(score))
```

Сопоставление аргументов: только ключевые аргументы

Хотим, чтобы в функцию всегда передавали аргументы **только** по имени:

```
>>> def f(x, y, *, degree=1):
...     return x**degree + y**degree
...
>>> f(2, 3)
5
>>> f(2, 3, degree=2)
13
>>> f(2, 3, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes 2 positional arguments but 3 given
```

Упаковка ключевых аргументов

Аналогично ключевые аргументы можно упаковывать и распаковывать:

```
>>> def f(**kwargs):
...     for key, value in kwargs.items():
...         print("{} - {}".format(key, value))
...
>>> f(**{"AGGCAC" : 33, "GGGTTA" : 44})
AGGCAC - 33
GGGTTA - 44
```

Только по позиции

Когда хотим запретить, чтобы передавали по имени:

```
>>> def f(x, y, /):
...     print(x + y)
...
>>> f(3, y=6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() got some positional-only arguments
passed as keyword arguments: 'y'
```

Сумма чисел из промежутка

1. Реализуем функцию sum, которая будет суммировать числа, если они из некоторого промежутка. Если промежуток не указан, то сумма всех.
2. Реализуйте функцию которая принимает промежуток и создает функцию из пункта 1

```
def bounded_sum(first, *args, lo=float("-inf"), \
                hi=float("inf")):


def make_min(*, lo, hi):
  def inner(first, *args):
    ...
  return inner
```

Области видимости

Области видимости

Виды:

- Встроенная область видимости - `min`, `list`, `str`
- Глобальная область видимости - на уровне модуля
- Локальные области видимости объемлющие функцию
- Локальная область видимости (функция)

Правило LEGB: Поиск имени переменной происходит в следующем порядке - в локальной области видимости, в областях видимости объемлющих функцию, в глобальной области и во встроенной области видимости.

Области видимости: локальная

Что будет выведено на экран в результате данного кода?

```
>>> a_var = 'global value'
>>> def outer():
...     a_var = 'enclosed value'
...     def inner():
...         a_var = 'local value'
...         print(a_var)
...     inner()
>>> outer()
???
```

Области видимости: локальная

Что будет выведено на экран в результате данного кода?

```
>>> a_var = 'global value'
>>> def outer():
...     a_var = 'enclosed value'
...     def inner():
...         a_var = 'local value'
...         print(a_var)
...     inner()
>>> outer()
local value
```

Области видимости: объемлющая

Что будет выведено на экран в результате данного кода?

```
>>> a_var = 'global value'
>>> def outer():
...     a_var = 'enclosed value'
...     def inner():
...         print(a_var)
...     inner()
>>> outer()
???
```

Области видимости: объемлющая

Что будет выведено на экран в результате данного кода?

```
>>> a_var = 'global value'
>>> def outer():
...     a_var = 'enclosed value'
...     def inner():
...         print(a_var)
...     inner()
>>> outer()
enclosed value
```

Области видимости: подводный камень

```
>>> counter = 0
>>> def f():
...     counter += 1
...
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
UnboundLocalError: local variable 'counter' referenced ...
```

Как быть?

Области видимости: подводный камень (2)

```
>>> counter = 0
>>> def f():
...     print(counter)
...     counter += 1
...
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
UnboundLocalError: local variable 'counter' referenced ...
```

Локальные переменные

Статическое разрешение локальных перемен

```
def foo():
    def bar():
        return x
    print(bar.__closure__)
    print(locals())
    x = 1
    print(bar.__closure__)
    print(locals())
```

```
(<cell at 0x10ad0f910: empty>,)
{'bar': <function ... at 0x10acaef70>}

(<cell at 0x10ad0f910: int object at 0x10ab984d0>,)
{'bar': <function ... at 0x10acaef70>, 'x': 92}
```

Оператор global

При указании оператора global интерпретатор начинает поиск имён вне функции:

```
>>> counter = 0
>>> def f():
...     global counter
...     counter += 1
...     return counter
>>> f()
1
>>> f()
2
```

Оператор nonlocal (1)

При указании оператора nonlocal интерпретатор ищет вне функции, но не в глобальной и не встроенной области.

```
>>> def f():
...     counter = 0
...     def inner():
...         nonlocal counter
...         counter += 1
...         return counter
...     return inner
...
>>> inner = f()
>>> inner()
1
>>> inner()
2
```

Оператор nonlocal (2)

```
>>> outer()
>>> def outer():
...     x = "local"
...     def inner():
...         nonlocal x
...         x = "nonlocal"
...         print("inner:", x)
...     inner()
...     print("outer:", x)
...
>>>
>>> outer()
inner: nonlocal
outer: nonlocal
```

Оператор nonlocal (3)

```
>>> def outer():
...     x = "local"
...     def inner():
...         x = "nonlocal"
...         print("inner:", x)
...     inner()
...     print("outer:", x)
...
>>>
>>> outer()
inner: nonlocal
outer: local
```

Оператор nonlocal (4)

Изменяемая ячейка памяти (пользователь никак не может получить):

```
>>> def cell(value=None):
...     def get():
...         return value
...     def set(update):
...         nonlocal value
...         value = update
...     return get, set
...
>>> get, set = cell()
>>> set(42)
>>> get()
42
```

Функции высших порядков

Функции высших порядков: filter

Функция filter фильтровать значения с помощью переданного предиката.

```
>>> filter(lambda x: x > 2, range(-5, 5))
<filter object at 0x10a8d5c18>
>>> set(filter(lambda x: x > 2, range(-5, 5)))
{3, 4}
>>> elements = [[], 0, "", {}, set, None]
>>> list(filter(None, xs))
[]
```

Функции высших порядков: map

Функция map позволяет обрабатывать одну или несколько последовательностей с помощью заданной функции

```
>>> map(lambda x: x ** 2, range(8))
<map object at 0x10a8d5c18>
>>> list(map(lambda x: x ** 2, range(8)))
[0, 1, 4, 9, 16, 25, 36, 49]
>>> list(map(lambda x: x ** 2 if x % 2 else x**3, range(8)))
[0, 1, 8, 9, 64, 25, 216, 49]
>>> list(map(lambda x, n: x ** n, range(4), range(5)))
[1, 1, 4, 27]
```

Как сделать min-max normalization?

$$z = \frac{x - min(x)}{max(x) - min(x)}$$

Функции высших порядков: zip

Функция zip принимает два или более итератора и возвращает список кортежей, составленных и соответствующих элементов

```
>>> list(zip(range(5), "ABCDE"))
[(0, 'A'), (1, 'B'), (2, 'C'), (3, 'D'), (4, 'E')]

>>> triples = zip(range(2), "AB", range(5, 7))
>>> list(triples)
[(0, 'A', 5), (1, 'B', 6)]

>>> res = 0
>>> for x, y in zip(range(5),range(5,10)):
...     res += (x - y)**2
>>> res
125
```

Как построить zip через map ?

```
map(lambda *args: args, ...)
```

Генераторы списков

Синтаксические конструкции, позволяющие создавать заполненные списки по определённым правилам.

```
>>> [x**2 for x in range(5)]
[0, 1, 4, 9, 16]
>>> [x**2 for x in range(10) if x % 2]
[1, 9, 25, 49, 81]
```

Аналогично генераторы множеств и словарей:

```
>>> {x**2 for x in range(-10, 10) if x % 2}
{81, 49, 25, 9, 1}
>>> {k : v for k, v in zip(range(5),"ABCDE")}
{0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E'}
```

Резюме

- Функции - это обычные объекты
- Передача аргментов по позиции или по именам
- Упаковка/Распаковка
- Область видимости - LEGB
- Функциональное программирование - `filter`, `map`, `zip`, ...

Спасибо за внимание!