Функции Python.

Лекция 2

Python

Объявление функции тоже выражение. См. Make_function. (0-6). 8-16 Вызов функции.

```
~/python-2018
λ cat def.py
def foo():
    """I do nothing and return 92."""
    return 92
foo()

~/python-2018
λ python3 -m dis def.py
λ python3 -m dis def.py
  1           0 LOAD_CONST               0 (<code object ...>)
              2 LOAD_CONST               1 ('foo')
              4 MAKE_FUNCTION            0
              6 STORE_NAME               0 (foo)

  4           8 LOAD_NAME                0 (foo)
             10 CALL_FUNCTION            0
             12 POP_TOP
             14 LOAD_CONST               2 (None)
             16 RETURN_VALUE
```

```
>>> def foo():
...     """I do nothing and return 92."""
...     return 92
...
>>> foo.__name__
'foo'
>>> foo.__doc__
'I do nothing and return 92.'
>>> help(foo)
```

Зарезервированное слово def предваряет определение функции. За ним должны следовать имя функции и заключённый в скобки список формальных параметров. Выражения, формирующие тело функции, начинаются со следующей строки и должны иметь отступ. Первым выражением в теле функции может быть строковой литерал — этот литерал является строкой документации функции, или док-строкой (docstring).

Пример документации на функцию

```
>>> def my_function():
...     """Не делаем ничего, но документируем.
...
...     Нет, правда, эта функция ничего не делает.
...     """
...     pass
```

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

```
min(1, 2)
min(1, y=2)
min(x=1, y=2)
min(y=2, x=1)
```

```
def min(*args):
    # type(args) => <class 'tuple'>

    res = float('inf')
    for x in args:
        res = x if x < res else res
    return res
```

```
min(92, 10, 62)
min()
xs = [1, 2, 3]
min(*xs)
```

Полугруппа
Может и не быть e

Моноид это бинарная операция удовлетворяющая следующим законам
Законы моноида
Ассоциативны - (x+y)+z = x+(y+z)
Являются бинарными операциями – x+y+z – все они и результат одного типа
имеют нейтральный элемент (e) ? e op x = x
1*x = x, x+0 =x

Несколько неименованных аргументов для поиска минимума.
Разименовывание списка, на вход функции поступает три аргумента.
Проблема – возвращаем значение даже при отсутствии элементов, может быть не очень хорошо.

- Функтор – функция применяется к элементам находящимся в некотором контексте, в функторе (список - функтор), объект к которому применима fmap.
- X = [1,2,3]
- >>> mp = map(lambda x:x*2,x)
- >>> list(mp)
- [2, 4, 6]
- Аппликативный функтор – извлеченная упакованная функция (сама функция возвращает неупакованное значение) применяется к извлеченному упакованному значению и получает упакованные значения
- монада: вы применяете функцию, возвращающую упакованное значение, к упакованному значению (PyMonad)

Примеры монад. Монада - тип, который позволяет строить цепочки вычислений. Между этими вычислениями передаются только монады, что и делает эти цепочки в некотором смысле универсальными. PyMonad.

```
def positive_and_negative(x):
.... return List(x, -x)
List(9) >> positive_and_negative
# Результатом станет монада List(9, -9).
@curry
def add_and_sub(x, y):
.... return List(y + x, y - x)
List(2) >> add_and_sub(3)
# вернёт список List(5,-1)

List(2) >> positive_and_negative >>
add_and_sub(3)
# Результатом станет List(5, -1, 1, -5)
```

```
def min(first, *rest):
    res = first
    for x in rest:
        res = x if x < res else res
    return res
```

```
>> min("hello", ",", " ", "world")
' '
```

```
>>> min()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: min() missing 1 required
    positional argument: 'first'
```

Проблема, краш в случае отсутствия, но зато можно отследить ошибку если вдруг, уже нельзя вызывать с пустым аргументом функцию, нужен хотя бы один
Вызовем x= []
min(*x)

```
def min(*args, default=None):
    if not args:
        return default

    res, *rest = args
    for x in rest:
        res = x if x < res else res
    return res


min(xs, default=0) # must use a name!
```

Иногда нужен минимум по пустому значению.
Указываем дефолтное значение в случае пустого списка аргументов.
Нельзя передать default без указания его имени.

```
def unique(xs, seen=None):
    seen = seen or set()
    res = []
    for x in xs:
        if x in seen:
            continue
        seen.add(x)
        res.append(x)
    return res

>>> unique([1, 2, 3, 2])
[1, 2, 3]
>>> unique([1, 2, 3, 2])
[1, 2, 3]
```

```
def flatten(iterable, *, depth=None):
    """I flatten a given iterable up to a fixed depth."""
    ...
```

Как заставить использовать только именованные ключевые аргументы где нам нужно.

Поставить *.

Код становится более понятным. Так как стоит *, то интерпретатор ожидает именованных переменных.

```
def min(a,*,b=6):
  if(a<b):
    return a
  else:
    return b
print(min(4))
print(min(2,b=3))
print(min(2,3))
```

```
4
2
Traceback (most recent call last):
  File "main.py", line 8, in <module>
    print(min(2,3))
TypeError: min() takes 1 positional
argument but 2 were given
```

```
>>> def call_me(*args, **kwargs):
...     return (args, kwargs)
...
>>> call_me({"a": 92})
(({'a': 92},), {})
```

```
>>> def call_me(*args, **kwargs):
...     return (args, kwargs)
...
>>> call_me(**{"a": 92})
((), {'a': 92})
```

- *- преобразует в аргументы список без имен.
- ** - преобразует словарь в именованные аргументы.

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

mas = [1,2]
print(add(*mas)) # 3
di = {'x':1,'y':2}
print(add(**di)) # 3
```

```
>>> x, *xs = [1, 2, 3]
>>> x, xs
(1, [2, 3])
```

Звездочку использовать как шаблон.
Пример получения головы и хвоста списка.

```
>>> x, *xs = "123"
>>> x, xs
('1', ['2', '3'])
```

Может быть любой итерабельный тип,
Но при применении звездочки
Получается список.

```
>>> first, *rest, last = [1, 2, 3]
>>> first, rest, last
(1, [2], 3)
```

```
>>> rectangle = ((0, 0), (2, 3))
>>> (x1, y1), (x2, y2) = rectangle
>>> y2
3
```

Зачем нужно присвоение такого рода.
Извлекаем из кортежа координаты в отдельные переменные.

```
"""
name,price,quantity
spam,8,92
"""

d = {}
for line in text.splitlines():
    cells = line.split(',')
    d[cells[0]] = cells[1]
```

Иногда может помочь обнаружить ошибку. Здесь могут склеиться две строки и мы получим не тот результат. Например 1 и 2, 2и 3 станут одной строкой, но мы этого не увидим.

```
"""
name,price,quantity
spam,8,92
"""

d = {}
for line in text.splitlines():
    name, price, _ = line.split(',') # checks format!
    d[name] = price
```

Здесь должно быть обязательно три строки разделенные запятой.
Если это не так, то краш.

```
"""
name,price,quantity
spam,8,92
"""

d = {}
for line in text.splitlines():
    name, price, *_ = line.split(',') # ignore explicitly!
    d[name] = price
```

Если нам явно неважно количество строк после двух, то можно Экранировать их звездочкой. Тогда 0 или любое количество.

```
>>> print(*[1], *[2], 3)
1 2 3
>>> dict(**{'x': 1}, y=2, **{'z': 3})
{'x': 1, 'y': 2, 'z': 3}
```

```
>>> *range(4), 4
(0, 1, 2, 3, 4)
>>> [*range(4), 4]
[0, 1, 2, 3, 4]
>>> {*range(4), 4}
{0, 1, 2, 3, 4}
>>> {'x': 1, **{'y': 2}}
{'x': 1, 'y': 2}
```

```
>>> {'x': 1, **{'x': 2}}
{'x': 2}
>>> {**{'x': 2}, 'x': 1}
{'x': 1}
```

При совпадении ключей, последний идет в словарь.

```
def min1(**kargs):
  print(kargs)
  for (key,it) in kargs.items():
    print(key,it)

  return


print(min1(**{'x':1,**{'x':2}}))
```

```
{'x': 2}
x 2
None
```

Области видимости.
Функции сработают после попытки

```
def is_even(n):
    return n == 0 if n <= 1 else is_odd(n - 1)


def is_odd(n):
    return n == 1 if n <= 1 else is_even(n - 1)


assert is_even(92)
```

Здесь вызов не сработает так как is_odd еще не добавлена и не именована.

```
def is_even(n):
    return n == 0 if n <= 1 else is_odd(n - 1)


assert is_even(92)


def is_odd(n):
    return n == 1 if n <= 1 else is_even(n - 1)


Traceback (most recent call last):
  File "scopes.py", line 5, in <module>
    assert is_even(92)
  File "scopes.py", line 2, in is_even
    return n == 0 if n <= 1 else is_odd(n - 1)
NameError: name 'is_odd' is not defined
```

Словарь глобальных и локальных объектов.

```
x = 1


def foo():
    y = 2
    print(globals(), type(globals()))
    print(locals(), type(locals()))


foo()

# {'x': 1, 'foo': <function ...>, ... } <class 'dict'>
# {'y': 2} <class 'dict'>
```

```
x = 1
def foo():
    print(x)
    # x = 2

foo()
# prints(1)
```

Python ищет сначала все присвоения локальной переменной.
О всех своих переменных и их использовании знает заранее.

```
x = 1
def foo():
    print(x) # --\ at function compile-to-bytecode time
    x = 2    # <-/

foo()
# UnboundLocalError:
# local variable 'x' referenced before assignment
```

Замыкание (closure) — это функция, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции в окружающем коде и не являющиеся ее параметрами.

```
def fun():
    x = 2
    def bar():
        print(x)
        x = 2
    bar()
    return
fun()
```

UnboundLocalError: local variable 'x' referenced before assignment

```
def fun():
    x = 2
    def bar():
        print(x)
        #x = 2
    bar()
    return
fun()
```

2

В python есть область видимости Global, Local и Enclosing.
Local – область видимости локальных переменных функции.
Enclosing – область видимости для внутренней функции внутри функции.
Локальные после вызова исчезают. Если на них не ссылались из вложенной функции.
Функция обходится полностью сначала и преобразуется в byte code.

```
>>> def mul5(a):
        return mul(5, a)
>>> mul5(2)
10
>>> mul5(7)
35
def mul(a):
  print(a)
  def helper(b):
    print(b)
    return a * b
  return helper

print(mul(5)(2))

>>> new_mul5 = mul(5)
>>> new_mul5
<function mul.<locals>.helper at 0x000001A7548C1158>
>>> new_mul5(2)
10
>>> new_mul5(7)
35
```

Для изменений глобальных и локальных вышестоящей функции.

```
x = 1
def foo():
    global x
    x = 2
    y = 1
    def bar():
        nonlocal y
        y = 2
```

Создает список функций с обращением к enclosing для последнего значения и с дефолтным значением. Примерно то же самое с многомерными массивами и [[]*3]*3. I в первом случае для вложенной функции глобальна, соответственно для функции берется ее последнее принятое значение.

```
def foo():
    res = []
    for i in range(3):
        def bar():
            return i

        res.append(bar)
    return res

for f in foo():
    print(f(), end=" ")
# prints 2 2 2
```

```
def foo():
    res = []
    for i in range(3):
        def bar(i=i):
            return i

        res.append(bar)
    return res
```

То же для x, при добавлении функций для каждой вложенной функции берется последнее присвоенное значение x.

```
def f():
  x = 2
  val = []
  def f1():
    return x
  val.append(f1)
  x = 9
  def f1():
    return x
  val.append(f1)
  return val
z = f()
for f in z:
  print(f)
  print(f())
```

```
<function f.<locals>.f1 at 0x7f2090499790>
9
<function f.<locals>.f1 at 0x7f2090499820>
9
```

Лямбда выражение

```
lambda arguments: expression
```

```
def _(arguments):
    return expression
```

```
lambda a, *args, b=1, **kwargs: 92
```

Функция высшего порядка map.

```
>>> range(3)
range(0, 3)
>>> list(range(3))
[0, 1, 2]
>>> map(lambda x: x + 1, [0, 1, 2])
<map object at 0x7fc54d060da0>
>>> list(map(lambda x: x + 1, [0, 1, 2]))
[1, 2, 3]
```

```
>>> list(map(lambda x, y: x + y, [0, 1, 2], [3, 4, 5, 6]))
[3, 5, 7]
```

Фильтрация по условию

```
>>> list(filter(lambda x: x % 2 == 0, range(10)))
[0, 2, 4, 6, 8]
```

```
>>> list(filter(None, [0, 1, True, False, [], {None}]))
[1, True, {None}]
```

Фильтрация без функции, убирает все Falsy элементы.

Спаривание итерируемых объектов.

```
>>> list(zip("hello", range(10)))
[('h', 0), ('e', 1), ('l', 2), ('l', 3), ('o', 4)]
```

```
assert len(xs) == len(ys)
for x, y in zip(xs, ys):
    ...
```

Идентичный результат, но первый вариант выглядит понятнее и короче.

```
>>> [x**2 for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
>>> list(map(
...     lambda x: x**2,
...     filter(lambda x: x % 2 == 0, range(10)),
... ))
[0, 4, 16, 36, 64]
```

```
res = [
    (x, y)
    for x in range(5)
    if x % 2 == 0
    for y in range(x)
    if y % 2 == 1
]
```

```
[(2, 1), (4, 1), (4, 3)]
```

```
res = []
for x in range(5):
    if x % 2 == 0:
        for y in range(x):
            if y % 2 == 1:
                res.append((x, y))
```

В первом случае множество, во втором словарь

```
>>> {x**2 % 5 for x in range(5)}
{0, 1, 4}
>>> {x: x**2 for x in range(5)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
>>> (x**2 for x in range(5))
<generator object <genexpr> at 0x7fcb9b4dac50>
>>> map(lambda x: x**2, range(5))
<map object at 0x7fcb9b4d6e80>
```

```
x, y, z = map(int, input().split())
```