Синтаксис объявления функций (1)

- Ограничения на имя функции в Python типичны[1]: можно использовать буквы, подчеркивание _ и цифры от 0 до 9, но цифра не должна стоять на первом месте.

```
>>> def foo():
...     return 42
...
>>> foo()
42
```

- **return** использовать не обязательно, по умолчанию функция возвращает **None**.

```
>>> def foo():
...     42
...
>>> print(foo())
None
```

[1] http://bit.ly/python-identifiers

Синтаксис объявления функций (2)

- Для документации функции используются строковые литералы

```
>>> def foo():
...     """I return 42."""
...     return 42
...
```

- После объявления функции документация доступна через специальный атрибут

```
>>> foo.__doc__
'I return 42.'
```

- В интерпретаторе удобней пользоваться встроенной функцией

```
>>> help(foo)  # или foo? в IPython.
```

Пример

```
>>> def min(x, y):                          #     __o
...     return x if x < y else y            #   _`\<,_
...                                         # (_)/ (_)
>>> min(-5, 12)                            # http://ascii.bike
-5
>>> min(x=-5, y=12)
-5
>>> min(x=-5, z=12)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: min() got an unexpected keyword argument 'z'
>>> min(y=12, x=-5)  # порядок не важен.
-5
```

- Находить минимум произвольного количества аргументов

```
>>> min(-5, 12, 13)
-5
```

- Использовать функцию `min` для кортежей, списков, множеств и других последовательностей

```
>>> xs = {-5, 12, 13}
>>> min(???)
-5
```

- Ограничить минимум произвольным отрезком `[lo, hi]`

```
>>> bounded_min(-5, 12, 13, lo=0, hi=255)
12
```

- По заданным `lo` и `hi` строить функцию `bounded_min`

```
>>> bounded_min = make_min(lo=0, hi=255)
>>> bounded_min(-5, 12, 13)
12
```

Упаковка и распаковка

```
>>> def min(*args):  # type(args) == tuple.
...     res = float("inf")
...     for arg in args:
...         if arg < res:
...             res = arg
...     return res
...
>>> min(-5, 12, 13)
-5
>>> min()
inf
```

Вопрос

Как потребовать, чтобы в `args` был хотя бы один элемент?

```
>>> def min(first, *args):
...     res = first
...     # ...
...
>>> min()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: min() missing 1 required [...] argument: 'first'
```

Вопрос

Как применить функцию `min` к коллекции?

```
>>> xs = {-5, 12, 13}
>>> min(???)
```

- Синтаксис будет работать с любым объектом, поддерживающим протокол итератора.

```
>>> xs = {-5, 12, 13}
>>> min(*xs)
-5
>>> min(*[-5, 12, 13])
-5
>>> min(*(-5, 12, 13))
-5
```

- Об итераторах потом, а пока вспомним про `bounded_min`

```
>>> bounded_min(-5, 12, 13, lo=0, hi=255)
12
```

```
>>> def bounded_min(first, *args, lo=float("-inf"),
...                 hi=float("inf")):
...     res = hi
...     for arg in (first, ) + args:
...         if arg < res and lo < arg < hi:
...             res = arg
...     return max(res, lo)
...
>>> bounded_min(-5, 12, 13, lo=0, hi=255)
12
```

Вопрос

В какой момент происходит инициализация ключевых аргументов со значениями по умолчанию?

Ключевые аргументы: подводные камни

```
>>> def unique(iterable, seen=set()):
...     acc = []
...     for item in iterable:
...         if item not in seen:
...             seen.add(item)
...             acc.append(item)
...     return acc
...
>>> xs = [1, 1, 2, 3]
>>> unique(xs)
[1, 2, 3]
>>> unique(xs)
[]
>>> unique.__defaults__
({1, 2, 3},)
```

Ключевые аргументы: правильная инициализация

```
>>> def unique(iterable, seen=None):
...     seen = set(seen or [])  # None --- falsy.
...     acc = []
...     for item in iterable:
...         if item not in seen:
...             seen.add(item)
...             acc.append(item)
...     return acc
...
>>> xs = [1, 1, 2, 3]
>>> unique(xs)
[1, 2, 3]
>>> unique(xs)
[1, 2, 3]
```

- Если функция имеет фиксированную арность, то ключевые аргументы можно передавать без явного указания имени:

```
>>> def flatten(xs, depth=None):
...     pass
...
>>> flatten([1, [2], 3], depth=1)
>>> flatten([1, [2], 3], 1)
```

- Можно явно потребовать, чтобы часть аргументов **всегда** передавалась как ключевые:

```
>>> def flatten(xs, *, depth=None):
...     pass
...
>>> flatten([1, [2], 3], 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: flatten() takes 1 positional argument [...]
```

Ключевые аргументы: упаковка и распаковка

- Ключевые аргументы, аналогично позиционным, можно упаковывать и распаковывать:

```
>>> def runner(cmd, **kwargs):
...     if kwargs.get("verbose", True):
...         print("Logging enabled")
...
>>> runner("mysqld", limit=42)
Logging enabled
>>> runner("mysqld", **{"verbose": False})
>>> options = {"verbose": False}
>>> runner("mysqld", **options)
```

- Поговорим о присваивании

```
>>> acc = []
>>> seen = set()
>>> (acc, seen) = ([], set())
```

- В качестве правого аргумента можно использовать любой объект, поддерживающий протокол итератора

```
>>> x, y, z = [1, 2, 3]
>>> x, y, z = {1, 2, 3}  # unordered!
>>> x, y, z = "xyz"
```

- Скобки обычно опускают, но иногда они бывают полезны

```
>>> rectangle = (0, 0), (4, 4)
>>> (x1, y1), (x2, y2) = rectangle
```

- В Python 3.0 был реализован[2] расширенный синтаксис распаковки

```
>>> first, *rest = range(1, 5)
>>> first, rest
(1, [2, 3, 4])
```

- * можно использовать в любом месте выражения

```
>>> first, *rest, last = range(1, 5)
>>> last
4
>>> first, *rest, last = [42]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 1 values to unpack
>>> *_, (first, *rest) = [range(1, 5)] * 5
>>> first
1
```

[2] http://python.org/dev/peps/pep-3132

Синтаксис распаковки работает в цикле **for**, например:

```
>>> for a, *b in [range(4), range(2)]:
...     print(b)
...
[1, 2, 3]
[1]
```

```
>>> import dis
>>> dis.dis("first, *rest, last = ('a', 'b', 'c')")
  0 LOAD_CONST               4 (('a', 'b', 'c'))
  3 UNPACK_EX              257
  6 STORE_NAME               0 (first)
  9 STORE_NAME               1 (rest)
 12 STORE_NAME               2 (last)
 15 LOAD_CONST               3 (None)
 18 RETURN_VALUE
```

Мораль

Присваивание в Python работает слева направо.

```
>>> x, (x, y) = 1, (2, 3)
>>> x
???
```

```
>>> dis.dis("first, *rest, last = ['a', 'b', 'c']")
  0 LOAD_CONST               0 (1)
  3 LOAD_CONST               1 (2)
  6 LOAD_CONST               2 (3)
  9 BUILD_LIST               3
 12 UNPACK_EX              257
 15 STORE_NAME               0 (first)
 18 STORE_NAME               1 (rest)
 21 STORE_NAME               2 (last)
 24 LOAD_CONST               3 (None)
 27 RETURN_VALUE
```

Мораль

Синтаксически схожие конструкции могут иметь различную семантику времени исполнения.

- Функции в Python могут принимать произвольное количество позиционных и ключевых аргументов.
- Для объявления таких функций используют синтаксис упаковки, а для вызова синтаксис распаковки

```
>>> def f(*args, **kwargs):
...     pass
...
>>> f(1, 2, 3, **{"foo": 42})
```

- Синтаксис распаковки также можно использовать при присваивании нескольких аргументов и в цикле **for**

```
>>> first, *rest = range(4)
>>> for first, *rest in [range(4), range(2)]:
...     pass
...
```

Дополнительные расширения синтаксиса распаковки

- В Python 3.5 возможности распаковки были в очередной раз расширены[3].
- Изменения затронули распаковку при вызове функции:

```
>>> def f(*args, **kwargs):
...     print(args, kwargs)
...
>>> f(1, 2, *[3, 4], *[5],
...   foo="bar", **{"baz": 42}, boo=24)
(1, 2, 3, 4, 5) {'baz': 42, 'boo': 24, 'foo': 'bar'}
```

- и при инициализации контейнеров:

```
>>> defaults = {"host": "0.0.0.0", "port": 8080}
>>> {**defaults, "port": 80}
{'host': '0.0.0.0', 'port': 80}
>>> [*range(5), 6]  # аналогично для множества и кортежа.
[0, 1, 2, 3, 4, 6]
```

- Скоро в вашей любимой операционной системе!

[3] http://python.org/dev/peps/pep-0448

Области видимости
aka scopes

- В отличие от Java (< 8), C/C++ (< 11) в Python функции — объекты первого класса, то есть с ними можно делать всё то же самое, что и с другими значениями.
- Например, можно объявлять функции внутри других функций

```
>>> def wrapper():
...     def identity(x):
...         return x
...     return identity
...
>>> f = wrapper()
>>> f(42)
42
```

```
>>> def make_min(*, lo, hi):
...     def inner(first, *args):
...         res = hi
...         for arg in (first, ) + args:
...             if arg < res and lo < arg < hi:
...                 res = arg
...         return max(res, lo)
...     return inner
...
>>> bounded_min = make_min(lo=0, hi=255)
>>> bounded_min(-5, 12, 13)
0
```

```
>>> min                  # builtin
<built-in function min>
>>> min = 42             # global
>>> def f(*args):
...     min = 2
...     def g():          # enclosing
...         min = 4       # local
...         print(min)
...
```

Правило LEGB

Поиск имени ведётся не более, чем в четырёх областях видимости: локальной, затем в объемлющей функции (если такая имеется), затем в глобальной и, наконец, во встроенной.

```
>>> min = 42      # ≡ globals()["min"] = 42
>>> globals()
{..., 'min': 42}
>>> def f():
...     min = 2  # ≡ locals()["min"] = 2
...     print(locals())
...
>>> f()
{'min': 2}
```

- Функции в Python могут использовать переменные, определенные во внешних областях видимости.
- Важно помнить, что поиск переменных осуществляется во время исполнения функции, а не во время её объявления.

```
>>> def f():
...     print(i)
...
>>> for i in range(4):
...     f()
...
0
1
2
3
```

- Для присваивания правило LEGB не работает

```
>>> min = 42
>>> def f():
...     min += 1
...     return min
...
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
UnboundLocalError: local variable 'min' referenced [...]
```

- По умолчанию операция присваивания создаёт локальную переменную.
- Изменить это поведение можно с помощью операторов **`global`** и **`nonlocal`**.

- Позволяет модифицировать значение переменной из глобальной области видимости

```
>>> min = 42
>>> def f():
...     global min
...     min += 1
...     return min
...
>>> f()
43
>>> f()
44
```

- Использование **global** порочно, почти всегда лучше заменить **global** на thread-local объект.

- Позволяет модифицировать значение переменной из объемлющей области видимости

```
>>> 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
```

- Прочитать мысли разработчиков на эту тему можно по ссылке http://python.org/dev/peps/pep-3104.

- В Python четыре области видимости: встроенная, глобальная, объемлющая и локальная.
- Правило LEGB: поиск имени осуществляется от локальной к встроенной.
- При использовании операции присваивания имя считается локальным. Это поведение можно изменить с помощью операторов `global` и `nonlocal`.

Функциональное программирование

Функциональное программирование: анонимные функции

- Python **не** функциональный язык, но в нём есть элементы функционального программирования.
- Анонимные функции имеют вид

```
>>> lambda arguments: expression
```

  и эквивалентны по поведению

```
>>> def <lambda>(arguments):
...         return expression
```

- Всё, сказанное про аргументы именованных функций, справедливо и для анонимных

```
>>> lambda foo, *args, bar=None, **kwargs: 42
<function <lambda> at 0x100fb9730>
```

- Применяет функцию к каждому элементу последовательности[4]

```
>>> map(indentity, range(4))
<map object at 0x100fc4c88>
>>> list(map(identity, range(4)))
[0, 1, 2, 3]
>>> set(map(lambda x: x % 7, [1, 9, 16, -1, 2, 5]))
{1, 2, 5, 6}
>>> map(lambda s: s.strip(), open("./HBA1.txt"))
<map object at 0x100fc4cc0>
```

- или последовательностей, количество элементов в результате определяется длиной наименьшей из последовательностей

```
>>> list(map(lambda x, n: x ** n,
...          [2, 3], range(1, 8)))
[2, 9]
```

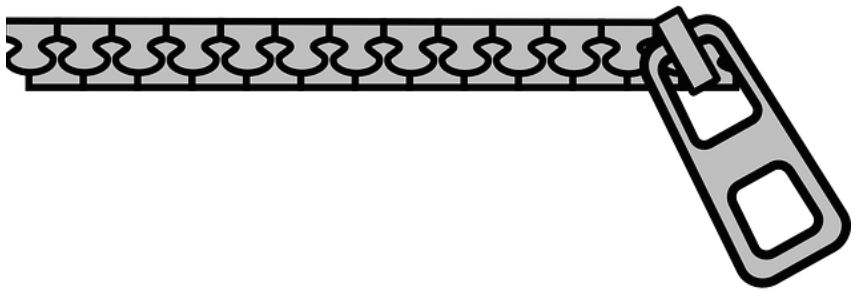
[4] Правильное слово *iterable*, т. е. объект поддерживающий протокол итератора.

- Убирает из последовательности элементы, не удовлетворяющие предикату

```
>>> filter(lambda x: x % 2 != 0, range(10))
<filter object at 0x1011edfd0>
>>> list(filter(lambda x: x % 2 != 0, range(10)))
[1, 3, 5, 7, 9]
```

- Вместо предиката можно передать **None**, в этом случае в последовательности останутся только *truthy* значения

```
>>> xs = [0, None, [], {}, set(), "", 42]
>>> list(filter(None, xs))
[42]
```



- Строит последовательность кортежей из элементов нескольких последовательностей

```
>>> list(zip("abc", range(3), [42j, 42j, 42j]))
[('a', 0, 42j), ('b', 1, 42j), ('c', 2, 42j)]
```

- Поведение в случае последовательностей различной длины аналогично `map`.

```
>>> list(zip("abc", range(10)))
[('a', 0), ('b', 1), ('c', 2)]
```

Вопрос

Как выразить `zip` через `map`?

Функциональное программирование: генераторы списков

- Пришли в Python из языка ABC, который позаимствовал их из языка SETL

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

- Компактная альтернатива комбинациям `map` и `filter`

```
>>> list(map(lambda x: x ** 2,
...          filter(lambda x: x % 2 == 1,
...                 range(10))))
[1, 9, 25, 49, 81]
```

- Могут быть вложенными

```
>>> nested = [range(5), range(8, 10)]
>>> [x for xs in nested for x in xs]  # flatten
[0, 1, 2, 3, 4, 8, 9]
```

```
>>> {x % 7 for x in [1, 9, 16, -1, 2, 5]}
{1, 2, 5, 6}
>>> date = {"year": 2014, "month": "September", "day": ""}
>>> {k: v for k, v in date.items() if v}
{'month': 'September', 'year': 2014}
>>> {x: x ** 2 for x in range(4)}
{0: 0, 1: 1, 2: 4, 3: 9}
```

Функциональное программирование: резюме

- Наличие элементов функционального программирования позволяет компактно выражать вычисления.
- В Python есть типичные для функциональных языков:
  - анонимные функции `lambda`,
  - функции `map`, `filter` и `zip`,
  - генераторы списков.
- Синтаксис Python также поддерживает генерацию других типов коллекций: множеств и словарей.

- PEP 8 содержит[5] стилистические рекомендации по оформлению кода на Python.
- Базовые рекомендации[6]
  - 4 пробела для отступов;
  - длина строки не более 79 символов для кода и не более 72 символов для документации и комментариев;
  - `lower_case_with_underscores` для переменных и имен функций, `UPPER_CASE_WITH_UNDERSCORES` для констант.

[5] http://python.org/dev/peps/pep-0008
[6] http://pocoo.org/internal/styleguide

- Унарные операции без пробелов, бинарные с одним пробелом с обеих сторон

```
exp = -1.05
value = (item_value / item_count) * offset / exp
```

- Для списков или кортежей с большим количеством элементов используйте перенос сразу после запятой

```
items = [
    'this is the first', 'set of items',
    'with more items', 'to come in this line',
    'like this'
]
```

- Не пишите тело оператора на одной строке с самим оператором

```
if bar: x += 1                  # Плохо
while bar: do_something()       # Плохо

if bar:                         # Лучше
    x += 1
while bar:                      # Лучше
    do_something()
```

- Не используйте Йода-сравнения в **if** и **while**

```
if 'md5' == method:             # Плохо
    pass

if method == 'md5'              # Лучше
    pass
```

- Для сравнения на равенство
  - объектов используйте операторы `==` и `!=`,
  - синглтонов используйте **`is`** и **`is not`**,
  - булевых значений используйте сам объект или оператор **`not`**, например

```
if foo:                          while not bar:
    # ...                            # ...
```

  - Проверяйте отсутствие элемента в словаре с помощью оператора **`not in`**

```
if not key in d:  # Плохо
if key not in d:  # Лучше
```

- Не используйте пробелы до или после скобок при объявлении и вызове функции

```
def foo (x, y):  # Плохо
    pass

foo( 42, 24 )    # Плохо
```

- Документируйте функции следующим образом

```
def something_useful(arg, **options):
    """One-line summary.

    Optional longer description of the function
    behaviour.
    """
```

- Разделяйте определения функций двумя пустыми строками.