Лекция 2: Функции

Дмитрий Коргун

dmitry.tbb@ya.ru

1 октября 2018г.

Синтаксис объявления функций

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

```
In [1]: def foo():
   ...:     return 42
   ...:

In [2]: foo()
Out[2]: 42
```

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

```
In [4]: def foo():
   ...:     42
   ...:

In [5]: print(foo())
None
```

https://docs.python.org/3/reference/lexical_analysis.html

Синтаксис объявления функций

- Для документации функции используются строковые литералы

```
In [6]: def foo():
   ...:     """I return 42"""
   ...:     return 42
```

- После объявления функции документация доступна через специальный атрибут

```
In [7]: foo.__doc__
Out[7]: 'I return 42’
```

- В интерпретаторе удобней пользоваться встроенной функцией

```
In [9]: foo?
Signature: foo()
Docstring: I return 42
File:      ~/<ipython-input-6-a8a344c85770>
Type:      function
```

Пример: min

```
In [10]: def min(x, y):
    ...:     return x if x < y else y
    ...:

In [11]: min(-5, 12)
Out[11]: -5

In [12]: min(x=-5, y=12)
Out[12]: -5

In [13]: min(x=-5, z=12)
---------------------------------------------------------------------------
TypeError: min() got an unexpected keyword argument 'z'

In [14]: min(y=12, x=-5)
Out[14]: -5
```

Пример: куда дальше?

- Находить минимум произвольного количества аргументов

```
In [1]: min(-5, 12, 13)
Out[1]: -5
```

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

```
In [2]: xs = {-5, 12, 13}
In [2]: min(???)
Out[2]: -5
```

- Ограничить минимум произвольным отрезком [lo, hi]

```
In [3]: bounded_min(-5, 12, 13, lo=0, hi=255)
Out[3]: 12
```

- По заданным lo и hi строить функцию bounded_min

```
In [4]: bounded_min = make_min(lo=0, hi=255)
In [4]: bounded_min(-5, 12, 13)
Out[4]: 12
```

Упаковка и распаковка

```
In [19]: def min(*args): # type(args) == tuple
    ...:     res = float("inf")
    ...:     for arg in args:
    ...:         if arg < res:
    ...:             res = arg
    ...:     return res
    ...:

In [20]: min(-5, 12, 13)
Out[20]: -5
```

```
In [21]: min()
Out[21]: inf
```

Вопрос

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

```
In [22]: def min(first, *args): # type(args) == typle
    ...:     res = first
    ...:     for arg in args:
    ...:         if arg < res:
    ...:             res = arg
    ...:     return res
    ...:
```

```
In [23]: min()
---------------------------------------------------------------------------
TypeError: min() missing 1 required positional argument:
'first'
```

Вопрос

Как применить min к коллекции?

```
In [24]: xs = {-5, 12, 13}

In [25]: min(???)
```

Распаковка аргументов

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

```
In [26]: min(*{-5, 12, 13})
Out[26]: -5

In [27]: min(*[-5, 12, 13])
Out[27]: -5

In [28]: min(*(-5, 12, 13))
Out[28]: -5
```

- Об итераторах потом, а пока вспомним про bounded_min

Ключевые аргументы: аргументы по умолчанию

```
In [29]: 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)
    ...:

In [30]: bounded_min(-5, 12, 13, lo=0, hi=255)
Out[30]: 12
```

Вопрос

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

Ключевые аргументы: подводные камни

```
In [35]: def unique(iterable, seen=set()):
    ...:     acc = []
    ...:     for item in iterable:
    ...:         if item not in seen:
    ...:             seen.add(item)
    ...:             acc.append(item)
    ...:     return acc
    ...:
```

```
In [36]: xs = [1, 1, 2, 3]

In [37]: unique(xs)
Out[37]: [1, 2, 3]

In [38]: unique(xs)
Out[38]: []

In [39]: unique.__defaults__
Out[39]: ({1, 2, 3},)
```

Ключевые аргументы: правильная инициализация

```
In [41]: 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
    ...:
```

```
In [42]: xs = [1, 1, 2, 3]

In [43]: unique(xs)
Out[43]: [1, 2, 3]

In [44]: unique(xs)
Out[44]: [1, 2, 3]
```

Ключевые аргументы: упаковка и распаковка

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

```
In [51]: def runner(cmd, **kwargs):
    ...:     if kwargs.get("verbose", True):
    ...:         print("Logging enabled")
    ...:
In [52]: runner("mysqld", limit=42)
Logging enabled
In [53]: runner("mysqld", **{"verbose": False})
In [54]: options = {"verbose": False}
In [55]: runner("mysqld", **options)
```

Распаковка и присваивание

- Поговорим о присваивании

```
In [56]: acc = []
In [57]: seen = set()
In [58]: (acc, seen) = ([], set())
```

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

```
In [59]: x, y, z = [1, 2, 3]
In [60]: x, y, z = {1, 2, 3}
In [61]: x, y, z = «xyz"
```

- Скобки обычно опускают, но иногда они бывают полезны

```
In [62]: rectangle = (0, 0), (4, 4)
In [63]: (x1, y1), (x2, y2) = rectangle
```

Расширенный синтаксис распаковки

- В Python 3.0 был реализован2 расширенный синтаксис распаковки

```
In [68]: first, *rest = range(1, 5)
In [69]: first, rest
Out[69]: (1, [2, 3, 4])
```

- * можно использовать в любом месте выражения

```
In [72]: first, *rest, last = range(1, 5)
In [73]: last
Out[73]: 4
In [74]: first, *rest, last = [42]
```

```
ValueError: not enough values to unpack (expected at
least 2, got 1)

In [75]: *_, (first, *rest) = [range(1, 5)] * 5
In [76]: first
Out[76]: 1
```

Синтаксис распаковки работает в цикле for, например:

```
In [77]: for a, *b in [range(4), range(2)]:
    ...:     print(b)
    ...:
[1, 2, 3]
[1]
```

Синтаксис распаковки работает в цикле for, например:

```
In [77]: for a, *b in [range(4), range(2)]:
    ...:     print(b)
    ...:
[1, 2, 3]
[1]
```

Упаковка и распаковка: резюме

- Функции в Python могут принимать произвольное количество позиционных и ключевых аргументов.

- Для объявления таких функций используют синтаксис упаковки, а для вызова синтаксис распаковки

```
In [78]: def f(*args, **kwargs):
    ...:     pass
```

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

```
In [79]: first, *rest = range(5)

In [80]: for first, *rest in [range(4), range(2)]:
    ...:     pass
```

Области видимости (scopes)

Функции внутри функций внутри функций

- 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

```
In [81]: def wrapper():
    ...:     def identity(x):
    ...:         return x
    ...:     return identity

In [82]: f = wrapper()
In [83]: f(42)
Out[83]: 42
```

Финальный аккорд — make_min

```
In [84]: 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 res
    ...:     return inner
```

```
In [85]: bounded_min = make_min(lo=0, hi=255)
In [86]: bounded_min(-5, 12, 13)
Out[86]: 12
```

```
In [89]: max #built-in
Out[89]: <function max>

In [90]: max = 42 # global

In [91]: def f(*args):
    ...:     min = 2 # enclosing
    ...:     def g():
    ...:         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.

- 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.

```
In [6]: def f():
   ...:     print(i)
In [7]: for i in range(4):
   ...:     f()
0
1
2
3
```

Для присваивания правило LEGB не работает

```
In [8]: min = 42
In [9]: def f():
   ...:     min += 1
   ...:     return min
In [10]: f()
---------------------------------------------------------------------------
UnboundLocalError: local variable 'min' referenced before
assignment
```

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

Оператор global

- Позволяет модифицировать значение переменной из глобальной области видимости

```
In [12]: min = 42
In [13]: def f():
    ...:     global min
    ...:     min += 1
    ...:     return min
In [14]: f()
Out[14]: 43
```

- Использовать global не рекомендуется и лучше избегать использования и тем более изменения глобальных объектов.

- Позволяет модифицировать значение переменной из объемлющей области видимости

```
In [15]: def cell(value=None):
    ...:     def get():
    ...:         return value
    ...:     def set(new_value):
    ...:         nonlocal value
    ...:         value = new_value
    ...:     return get, set
In [16]: get, set = cell()
In [17]: set(42)
In [18]: get()
Out[18]: 42
```

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

Области видимости: резюме

- 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.
- При помощи функций globals() и locals() можно получить словари со списками глобальных и локальных объектов

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 эквиваленты по поведению

```
In [23]: def <lambda>(arguments):
    ...:     expression
```

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

```
In [24]: lambda foo, *args, bar=None, **kwargs: 42
Out[24]: <function __main__.<lambda>>
```

- Применяет функцию к каждому элементу последовательности

```
In [5]: map(identity, range(4))
Out[5]: <map at 0x105fe8278>

In [6]: list(map(identity, range(4)))
Out[6]: [0, 1, 2, 3]

In [7]: set(map(lambda x: x % 7, [1, 9, 16, -1, 2, 5]))
Out[7]: {1, 2, 5, 6}
```

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

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

- Убирает из последовательности элементы, не удовлетворяющие предикату

```
In [9]: filter(lambda x: x % 2 != 0, range(10))
Out[9]: <filter at 0x106057668>
In [10]: list(filter(lambda x: x % 2 != 0, range(10)))
Out[10]: [1, 3, 5, 7, 9]
```

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

```
In [11]: xs = [0, None, [], {}, set(), "", 42]
In [12]: list(filter(None, xs))
Out[12]: [42]
```

- Строит последовательность кортежей из элементов нескольких последовательностей

```
In [13]: list(zip("abc", range(3), [42j, 42j, 42j]))
Out[13]: [('a', 0, 42j), ('b', 1, 42j), ('c', 2, 42j)]
```

- Поведение в случае последовательностей различной длины аналогично map.

```
In [14]: list(zip("abc", range(10)))
Out[14]: [('a', 0), ('b', 1), ('c', 2)]
```

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

```
In [15]: [x ** 2 for x in range(10) if x % 2 == 1]
Out[15]: [1, 9, 25, 49, 81]
```

- Компактная альтернатива комбинациям map и filter

```
In [16]: list(map(lambda x: x ** 2,
    ...:               filter(lambda x: x % 2 == 1,
    ...:                      range(10))))
Out[16]: [1, 9, 25, 49, 81]
```

- Могут быть вложенными

```
In [17]: nested = [range(5), range(8, 10)]
In [18]: [x for xs in nested for x in xs]
Out[18]: [0, 1, 2, 3, 4, 8, 9]
```

Функциональное программирование: генераторы множеств и словарей

```
In [19]: {x % 7 for x in [1, 9, 16, -1, 2, 5]}
Out[19]: {1, 2, 5, 6}

In [21]: date = {"year": 2018, "month": "October", "day": ""}

In [22]: {k: v for k, v in date.items() if v}
Out[22]: {'month': 'October', 'year': 2018}
```

```
In [23]: {x: x ** 2 for x in range(4)}
Out[23]: {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.