

Python

Лекция 2

Преподаватель: Дмитрий Косицин
BSU FAMCS (Fall'18)

Особенности работы с коллекциями

Упаковка и распаковка переменных.
Особенности итерирования по нескольким коллекциям. Слайсы.
Функциональная обработка коллекций.

© Dmitry Kasitsyn

BSU FAMCS (Fall'18) • 3

Производные коллекции и алгоритмы

© Dmitry Kasitsyn

BSU FAMCS (Fall'18) • 19

Модули и пакеты

© Dmitry Kasitsyn

Selfing Math Solutions (Spring'18) • 24

Функции

© Dmitry Kasitsyn

Selfing Math Solutions (Spring'18) • 32

Области видимости переменных

Области видимости переменных. Замыкания. Лямбда-функции. Функционалы из
стандартной библиотеки.

© Dmitry Kasitsyn

Selfing Math Solutions (Spring'18) • 40

Логирование

© Dmitry Kasitsyn

Selfing Math Solutions (Spring'18) • 51

Форматирование строк

© Dmitry Kasitsyn

Selfing Math Solutions (Spring'18) • 55

Работа с командной строкой

© Dmitry Kasitsyn

Selfing Math Solutions (Spring'18) • 59

Ответы и полезные ссылки

© Dmitry Kasitsyn

BSU FAMCS (Fall'18) • 63

Особенности работы с КОЛЛЕКЦИЯМИ

...

Упаковка и распаковка переменных.

Особенности итерирования по нескольким коллекциям. Слайсы.

Функциональная обработка коллекций.

Упаковка переменных

Упаковка – создание кортежа / списка / т.п.

```
>>> x = [1, 2]
>>> x0 = x[0]
>>> x1 = x[1]
>>> print(x[0], x[1])
1, 2
```

Пример распаковки кортежа (пары значений):

```
>>> for x, y in zip(range(5), range(5, 10)):
>>>     print(x, y)
```

Распаковка переменных

Кортеж можно распаковать автоматически:

```
>>> x0, x1 = x
>>> print(x[0], x[1])
1, 2
```

Для обмена переменных местами – упаковать и распаковать в другом порядке:

```
>>> x[0], x[1] = (x[1], x[0])    # скобки не обязательны
>>> print(x[0], x[1])
2, 1
```

Распаковка переменных

Важно! Присваивание выполняется справа налево, но подвыражения в присваивании – в порядке следования:

```
>>> x = [1, 2]
>>> i = 0
>>> i, x[i] = 1, 1
>>> print(x)
[1, 1]
```

Распаковывать можно **list**, **tuple**, **set**, а также итераторы.

Распаковка переменных

Допускается вложенность:

```
>>> ((x, _), z) = [[1, 2], 3]
>>> print x, z
1, 3
```

Важно! Символ `,` (запятая) является элементом синтаксиса не только кортежей – например, при бросании исключений распаковывать кортеж неявно нельзя.

В Python 3 допустима частичная распаковка ([PEP-3132](#)):

```
>>> head, *middle, tail = range(5)
>>> print head, middle, tail
0, [1, 2, 3], 4
```

Итерирование по нескольким коллекциям

Итерирование по двум коллекциям – функция **zip**:

```
>>> for x, y in zip([1, 2, 3], [-3, -2, -1]):  
>>>     print(x % y == 0, end=' ')  
False True True
```

Важно! **zip** возвращает новый список кортежей в Python 2.x и генератор в Python 3.x.

Вопрос: что будет, если коллекции разной длины?

enumerate(x) – сокращение **zip**(**range**(**len**(x), x))

Слайсы

Можно обращаться сразу к нескольким элементам коллекции:

```
>>> x = list(range(10))
>>> print(x[0:10:2])
0 2 4 6 8
```

Замечание. Вспомните `range(0, 10, 2)`

Слайсы за пределами коллекции или некорректные:

```
>>> x[100:110]
[] # тип соответствует типу переменной x

>>> x[0:10:-2]
[]
```

Способы задания слайсов

Следующие записи эквивалентны:

- `x[9:0:-2]`
- `x[-1:0:-2]`
- `x[:None:-2]`

Слайс – это объект **slice**

```
>>> x[slice(9, 0, -2)] == x[9:0:-2]
```

True

Замечания по работе со слайсами

Слайсу можно дать имя и связать с переменной

```
>>> even_indices_slice = slice(None, None, 2)
>>> print(x[even_indices_slice])
```

Слайсам можно присваивать, в том числе iterable с иным количеством элементов:

```
>>> x = [1, 2, 3]
>>> x[0:2] = [7, 6, 5]
>>> print(x)
[7, 6, 5, 3]
```

Замечания по работе со словарями

Особенности создания словарей с ключами, имеющими одинаковый хэш:

```
>>> x = {1: 'a', True: 'b', 1.0: 'c'}  
>>> assert x == {1.0: 'c'}
```

Методы для итерирования в Python 2.x:

- **keys()**, **values()**, **items()** – возвращают списки ключей, значений и пар ключ-значение
- **iterkeys()**, **itervalues()**, **iteritems()** – возвращают, соответственно, итераторы
- **viewkeys()**, **viewvalues()**, **viewitems()** – возвращают view-объекты

В Python 3.x методы **keys()**, **values()** и **items()**, возвращают на самом деле *view*-объекты. Такие объекты отражают изменения в исходной коллекции.

Замечания по работе со списками

Три способа создать список, содержащий три списка:

```
>>> x = [[], [], []]
>>> y = [[]] * 3
>>> z = []
>>> for _ in range(3):
>>>     z.append([])
([[], [], [], [], [], []])
```

использовать ";" (semicolon) нельзя!

```
>>> x[0].append(1); y[0].append(2); z[0].append(3)
```

Замечания по работе со списками

```
>>> print(x, y, z)
([[1], [], []], [[2], [2], [2]], [[3], [], []])
```

Wow!.. Лучше использовать другой способ!

```
>>> y_pretty = [[] for _ in range(3)]
>>> y_pretty[0].append(2)
>>> print(y_pretty)
[[2], [], []]
```

Такая конструкция называется list-comprehension.

Функциональный подход

Основные функции для работы с последовательностями:

- **map** – применить функцию к каждому элементу последовательности;
- **filter** – оставить только те элементы, для которых переданная функция возвращает **True** (предикатом может быть **None**);
- **all** – возвращает **True**, если все элементы преобразуются к **True**;
- **any** – возвращает **True**, если хотя бы один элемент **True**.

Важно! В Python 2.x функции **map** и **filter** возвращают списки, в то время как в Python 3.x – генераторы.

Comprehensions

Comprehensions допускают вложенные **for** и одно **if** выражение:

```
>>> def is_odd(x):  
>>>     return bool(x % 2)  
  
>>> s1 = filter(is_odd, range(10))  
>>> s2 = [x for x in range(10) if is_odd(x)]  
>>> assert s1 == s2
```

Tuple-comprehension нету. Выражение с круглыми скобками – генератор

```
>>> s3 = list(x for x in range(10) if is_odd(x))  
>>> assert s1 == s2 == s3
```


Comprehensions

Есть comprehension-выражения для создания словарей и множеств:

```
>>> d = {x: y**2
...      for x in range(2)
...      for y in range(x + 1) }
>>> assert d == {0: 0, 1: 1}
```

```
>>> s = {1 for _ in range(10) }
>>> assert s == set([1])
```

Замечание. Перед **for** может стоять любое допустимое выражение, в том числе содержащее некоторое преобразование (см. y^{**2}) или условие.

Замечание. Comprehension-выражения допускают произвольное количество **for** и **if**, причем *не* обязательно поочередно: после **for** допустимо несколько **if** ов.

Statement *del*

Statement **del** имеет несколько смысловых нагрузок:

- «разрывает связь» между переменной и объектом (здесь также задействуется счетчик ссылок объекта)
- удаляет элемент, атрибут или слайс (за удаление отвечает сам объект)

```
>>> class X(object):  
>>>     a = None  
>>>  
>>> del X.a  
>>>  
>>> y, z = [1, 2, 3], {'x': 0}  
  
>>> del y[0], z['x']
```

Вопрос: как с помощью **del** очистить список, не удалив при этом объект?

Производные коллекции и алгоритмы ...

Встроенные коллекции

В Python реализованы следующие коллекции:

- **deque** – дек, двухсторонняя очередь
- **defaultdict** – словарь, который возвращает значение по умолчанию в случае отсутствия ключа
- **Counter** – реализация **defaultdict**, когда для всех ключей значение по умолчанию – ноль
- **OrderedDict** – словарь, сохраняющий порядок вставки элементов
- **namedtuple** – именованный кортеж
- **Queue** – потокобезопасная очередь
- **array** – массив, хранящий данные определенного C-совместимого типа

Подробнее о defaultdict

Подробнее о **defaultdict**:

```
>>> import collections
>>> def f(): return 0
>>> x = collections.defaultdict(f)
>>> print(x[2])
0
```

Важно! Конструктор **defaultdict** требует не число, а объект, при вызове которого будет возвращаться объект.

Подробнее о namedtuple

Подробнее о **namedtuple** (именованный кортеж):

```
>>> Point = collections.namedtuple('Point', ['x', 'y'])
>>> p = Point(1, 2)
>>> print(p.x == p[0] == 1 and p.y == p[1] == 2)
True
```

Методы **namedtuple**:

- *_fields* – вернет имена полей ('x', 'y')
- *_asdict()* – вернет **OrderedDict** с соответствующими ключами и значениями
- *_replace(x=new_value, ...)* – вернет **namedtuple** с замененными значениями

Алгоритмы стандартной библиотеки

В стандартной библиотеке реализованы алгоритмы по работе с кучей и упорядоченным списком:

- **heapq** – модуль, содержащий функции по созданию кучи (heap), добавлению элементов, взятию k-максимальных
- **bisect** – модуль, содержащий функцию бинарного поиска элемента по списку, а также вставки элемента в упорядоченную последовательность

Модули и пакеты

...

Схема импорта модулей

При вызове «import x» происходит следующее:

1. find module
2. load module # an object loaded_module is created
 - create an object type of ModuleType
 - read source
 - compile source
 - execute source
3. sys.modules['x'] = loaded_module
4. <this_module>.x = loaded_module

Загрузка и перезагрузка модуля

reload(loaded_module) – перезагружает модуль, не создавая новый объект.

Сравните:

```
>>> from module import x  
>>> # then use x directly
```

```
>>> import module  
>>> # then use module.x
```

Важно! Это не то же самое, что удалить модуль и заново загрузить. В случае **reload** (`importlib.reload` в Python 3) все объекты в модуле *пересоздаются!* Более того, **reload** имеет множество подводных камней.

Загрузка модулей

При загрузке создаются и вызываются default-значения функций (вопрос: что произойдет?):

```
>>> def f (x=g ()) :  
>>>     pass
```

```
>>> def g () :  
>>>     return 0
```

Для изменения поведения при исполнении модуля от поведения при импорте используется следующее:

```
>>> if __name__ == '__main__':  
>>>     # code to be executed if module is an entry point
```

Импорт модулей

Модуль можно загружать по имени:

```
>>> import importlib
```

```
>>> module_instance = importlib.import_module('module_name')
```

Есть возможность загружать source, compiled (.pyc) и dynamic (.pyd, .so) модули по полному пути (см. *imp* в Python 2 и *importlib.util* в Python 3)

Очередность загрузки:

- package
- module
- namespace ([PEP 420](#), Python 3.3+)

Пакеты и пространства имен

Package – папка с модулями, где присутствует файл `__init__.py`.

Namespace – файл `__init__.py` отсутствует.

Разница будет для вложенных папок: package вложенный обнаружится, а для namespace нужно явно прописать путь в **sys.path**.

Напоминание. `sys.path.append(x)` равносильно `sys.path += x`, но отличается от `sys.path = sys.path + x`

Замечание. Узнать имя файла из модуля можно обратившись к переменной `__file__`, имя модуля – к `__name__`.

ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ

Модули можно загружать прямо из *zip*-архива.

Можно использовать относительные импорты ([PEP-328](#)).

Для **moduleX.py** верны следующие относительные импорты:

```
package/  
  __init__.py  
  subpackage1/  
    __init__.py  
    moduleX.py  
    moduleY.py  
  subpackage2/  
    __init__.py  
    moduleZ.py  
  moduleA.py
```

```
from .moduleY import spam  
from .moduleY import spam as ham  
from . import moduleY  
from ..subpackage1 import moduleY  
from ..subpackage2.moduleZ import eggs  
from ..moduleA import foo  
from ...package import bar  
from ...sys import path
```

Дополнительные возможности.

Замечания

Относительные импорты не столь распространены ввиду худшей переносимости.

У модуля также может присутствовать *docstring* – его следует располагать вверху файла в тройных кавычках.

Модуль `__future__` является директивой компилятору создать `.pyc` файл, используя другие инструкции.

Существуют дополнительные механизмы: `path hooks`, `metapath`, `module finders and loaders`, etc.

Функции

...

Синтаксис функций

Определение функции:

```
def f(positional, positional_with_default=default_value,  
      *variadic_arguments, **keyword_arguments):  
    pass
```

В Python нету перегрузки функций – используются значения по умолчанию и динамическая типизация.

Значение по умолчанию вычисляется единожды при определении функции, а потому *должно* быть неизменяемым.

ВЫЗОВ ФУНКЦИЙ

Примеры вызовов:

```
>>> def f(x, y=0, *args, **kwargs):  
>>>     return x, y, args, kwargs
```

```
>>> f()      # TypeError  
>>> f(1)     # x: 1, y: 0, args: tuple(), kwargs: {}  
>>> f(1, 2)   # x: 1, y: 2, args: tuple(), kwargs: {}  
>>> f(1, 2, 3) # x: 1, y: 2, args: tuple(3), kwargs: {}
```

Допустима распаковка аргументов при вызове функции:

```
>>> f(* (1, 2, 3, 4))  
# x: 1, y: 2, args: tuple(3, 4), kwargs: {}
```

Передача аргументов по КЛЮЧЕВЫМ СЛОВАМ

В Python 3.5+ ([PEP-448](#)) допустима передача нескольких аргументов для распаковки:

```
>>> f(* (1, 2, 3), * (5, 6))  
# x: 1, y: 2, args: tuple(3, 5, 6), kwargs: {}
```

Аргументы можно передавать по ключевым словам (порядок произвольный):

```
>>> f(y=1, x=2) # x: 2, y: 1, args: tuple(), kwargs: {}
```

Минусы:

- возможно более медленное выполнение ([Issue 27574](#))
- проблемы с переименованием

Передача аргументов по КЛЮЧЕВЫМ СЛОВАМ

Передаваемые по ключевым словам аргументы, для которых нет имен, помещаются в `kwargs`:

```
>>> f(x=1, z=2)    # x: 1, y: 0, args: tuple(), kwargs: {'z': 2}
```

Замечание. Порядок `kwargs` гарантируется с Python 3.6 ([PEP-468](https://peps.python.org/pep-468/)).

Допустима распаковка аргументов по ключевым словам:

```
>>> f(**{'x': 1, 'z': 2})  
# x: 1, y: 0, args: tuple(), kwargs: {'z': 2}
```

ИСКЛЮЧИТЕЛЬНО КЛЮЧЕВЫЕ аргументы

В Python 3 функции могут принимать аргументы исключительно по ключевому слову ([PEP-3102](#)):

```
>>> def f(*skipped, some_value=0):  
>>>     pass
```

```
>>> f(1, 2, 3)      # skipped: (1, 2, 3), some_value: 0  
>>> f(some_value=100)  # skipped: tuple(), some_value: 100
```

Важно! Если значение по умолчанию не указано, функция обязана вызываться с данным ключевым аргументом, иначе возникнет **TypeError**.

Замечание. Имя variadic аргумента может быть опущено.

Алгоритм маппинга аргументов

Значения назначаются аргументам функции по порядку:

- На позиционные слоты
- На `variadic` аргумент (в кортеж неименованных элементов)
- Переданные по ключевым словам – либо на позиционные, либо в словарь

Важно! Если аргумент позиционный аргумент не был передан или ключевой аргумент был передан более 1 раза, возникнет **`TypeError`**.

Вопрос: что будет, если имя `variadic` аргумента опущено (вместо `*args` оставлен просто символ `"*"`), но в функцию переданы `variadic` аргументы?

Полный алгоритм маппинга аргументов также описан в [PEP-3102](#).

Документация функций

Описание функции и их аргументов производится в *docstring* ([PEP-257](https://pep.python.org/pep-0257/)):

```
def complex(real=0.0, imag=0.0):  
    """Form a complex number.  
  
    Keyword arguments:  
    real -- the real part (default 0.0)  
    imag -- the imaginary part (default 0.0)  
    """  
    # some code here ...
```

Документация может быть получена вызовом функции **help(f)** или
взятием аргумента **f.__doc__**

Области видимости переменных

...

Области видимости переменных. Замыкания. Лямбда-функции. Функционалы из стандартной библиотеки.

Области видимости переменных

Области видимости переменных определяются функциями:

- *built-in* – встроенные общедоступные имена (доступны через модуль `builtins` или `__builtins__`, например, **sum**, **abs** и т.д.)
- *global* – переменные, определенные глобально для модуля
- *enclosing* – переменные, определенные в родительской функции
- *local* – локальные для функции переменные

Локальные переменные в функциях могут в них свободно изменяться, *enclosing*, *global* и *built-in* – только читаться ([PEP-227](#)).

Области видимости переменных

Пример:

```
>>> abs(2)    # built-in
>>> abs = dir  # global, overrides
>>> def f():
>>>     abs = sum    # enclosing
>>>     def g():
>>>         abs = max    # local
```

Для справки. (Нужно крайне редко). Для переопределения **abs** из функции **g** в функции **f** используется ключевое слово ***nonlocal***, для переопределения глобальной переменной **abs** – ключевое слово ***global*** ([PEP-3104](#)).

```
>>> global abs
>>> abs = max    # переопределит abs, глобальный для модуля
```

Переменные в циклах

Циклы *не имеют* своей области видимости: как только переменная была создана, она становится доступной и после цикла.

```
>>> for i in range(10):  
>>>     if i > 5:  
>>>         break  
>>> print(i)  
6
```

Замечание. В Python 3.4 и ниже так же «утекали» переменные из comprehension-выражений. Баг был исправлен в Python 3.5.

```
>>> x = [i for i in range(10)]  
>>> print(i)
```

```
NameError: name 'i' is not defined # Python 3.6
```

Локальные и глобальные переменные

В Python можно получить доступ ко всем локальным и глобальным переменным:

- **locals()** – словарь видимых локальных переменных
- **globals()** – аналогичный словарь глобальных переменных

Словари автоматически обновляются интерпретатором.

Вопрос: можно ли модифицируя эти словари добавлять, изменять или удалять соответствующие переменные?

Замыкания

В функции доступны переменные, определенные уровнями выше – они *замыкаются*.

```
>>> def make_adder(x):  
>>>     def adder(y):  
>>>         return x + y  
>>>     return adder  
  
>>> add_five = make_adder(5)  
>>> add_five(10) # 15
```

Важно! Значение замкнутой переменной получается *каждый раз* при вычислении выражения.

Пример замыкания

```
>>> x = 2
```

```
>>> def make_adder():  
>>>     def adder(y):  
>>>         return x + y  
>>>     return adder
```

```
>>> add_x = make_adder()  
>>> add_x(-2)    # 0
```

```
>>> del x    # delete 'x' - unbind variable name with object  
>>> add_x(-2)    # NameError: name 'x' is not defined
```

Lambda-функции

Lambda-функции в Python допускают в себе одно лишь выражение:
lambda arguments: expression

Эквивалентно:

```
def <lambda_name>(arguments):  
    return expression
```

Вопрос: как будет выглядеть **lambda**, которая ничего не принимает и не возвращает?

Важно! С точки зрения bytecode **lambda** аналогична функции с тем же кодом, но при использовании **def** объект-функция еще получает имя.

Пример lambda-функций

Функция, возвращающая сумму аргументов:

```
>>> lambda x, y: x + y
```

Пример списка lambda-функций:

```
>>> collection_of_lambdas = [lambda: i*i for i in range(6)]  
>>>  
>>> for f in collection_of_lambdas:  
>>>     print(f())
```

Вопрос: что будет выведено в результате выполнения?

Пример lambda-функций

Поскольку вычисление происходит run-time, то для всех созданных функций значение `i` будет равно 5. Переменная `i` была «захвачена» в comprehension-выражении, хоть и вне этого выражения она недоступна (в Python 3.6).

Для «захвата» значения можно создать локальную для **lambda** копию:

```
>>> lambdas = [lambda i=i: i*i for i in range(6)]
```

В модуле **operator** ([Py2](#), [Py3](#)) есть множество функционалов, которыми можно пользоваться наряду с **lambda**-функциями:

```
>>> import operator
>>> # аналог: lambda x, y: x + y
>>> operator.add # operator.__add__
```

Замечания по операторам

Помимо операторов арифметических операций и операций сравнения, есть функционалы для работы с атрибутами и элементами коллекций.

```
f = operator.attrgetter('name.first', 'name.last')  
# the call f(b) returns (b.name.first, b.name.last)
```

```
g = operator.itemgetter(2, 5, 3)  
# the call g(r) returns (r[2], r[5], r[3])
```

```
h = operator.methodcaller('name', 'foo', bar=1)  
# the call h(b) returns b.name('foo', bar=1)
```

Функционалы и `lambda`-функции наряду с обычными функциями используются как аргументы, выполняющие некоторое действие, например, в **map**, **filter**.

Логирование

...

Логирование

```
import sys
import logging
import datetime

logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)
logger.addHandler(logging.handlers.StreamHandler(sys.stdout))

logger.info("current year's %d", datetime.datetime.today().year)
```

Замечание. В Python 3 Handlers и Filters расположены во вложенных модулях, а содержимое модуля datetime перемещено.

Логирование: подробности

Объекты **Logger** объявлены в модуле **logging**.

Создать новый logger можно вызовом **getLogger**, передав ему некоторое имя.

Замечание. Если передать имя существующего логгера в метод **getLogger**, то будет возвращен уже существующий логгер с таким именем.

Логировать сообщение можно с помощью методов **debug**, **info**, **warning**, **error**, **critical** или общего **log**.

Установить уровень чувствительности (verbosity) можно методов **setLevel**.

Логирование: подробности

Добавить обработчик можно методом **addHandler**, фильтр – **addFilter**.

Реализованные обработчики: **StreamHandler**, **FileHandler**, **RotatingFileHandler**, **SocketHandler**, etc.

Замечание. Все handler'ы и filterer'ы имеют базовые классы - **logging.Handler** и **logging.Filterer**.

Конфигурация логгера может быть сохранена в файле и загружена с помощью **logging.config.dictConfig**.

Форматирование строк

...

Форматирование строк

Поддерживается *printf*-style форматирование:

```
>>> 'number of %.3f values in %s is %d' % (0.1234, 'some object', 3)
number of 0.123 values in some object is 3
```

Есть возможность использовать именованные аргументы:

```
>>> 'number of %(name)s is %(count)d' % {'name': 'names', 'count': 2}
number of names is 2
```

Поддерживается новый стиль форматирования строк:

```
>>> 'number of {0:.3f} values in {1} is {2}'.format(
    0.1234, 'some object', 3)
number of 0.123 values in some object is 3
```


Форматирование строк

А еще можно:

- аналогично использовать именованные аргументы: `" { name } "`
- индексировать аргументы: `" { items [0] } "`
- обращаться к атрибутам: `" { point . x } "`
- опускать индексы: `" { } { } "`
- повторять и менять местами индексы: `" { 1 } { 0 } { 1 } "`

Вопрос: что будет, если не совпадает количество аргументов для подстановки? А если нету такого именованного аргумента?

Форматирование строк

Замечание. Если вам нужно подставить множество локальных переменных, можно использовать словари **locals()** и **globals()**, определенные интерпретатором:

```
>>> x = 2
>>> "{x}".format(**locals())
2
```

Вопрос: верно ли, что так хитро можно менять значения локальных переменных?

Замечание. В Python 3 добавлен метод **format_map**, чтобы передавать словарь не распаковывая.

Работа с командной строкой

...

Парсинг аргументов командной строки

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')

parser.add_argument('integers', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate',
                    action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max) ')

args = parser.parse_args()
print(args.accumulate(args.integers))
```

Для парсинга есть **ArgumentParser** в модуле **argparse**.

Методы **parse_args** и **parse_known_args** принимают некоторый список аргументов (по умолчанию **sys.argv**), парсят его и возвращают объект **Namespace**.

Произвольную строку запуска можно разбить на список с помощью модуля **shlex**.

Информации об аргументе добавляется с помощью **add_argument**:

- имена переменных через '-' (dash), '--' (double dash) или без них
- *dest* – имя переменной, в которой хранится значение
- *type* – преобразование типа
- *action* – действие при получении аргумента (*store*, *store_true*, *append*, etc.)
- *nargs* – количество аргументов (1, 1 и более, 0 и более)
- *default* – значение по умолчанию (если не передан)
- *required* – обязательный аргумент
- *choices* – список возможных значений
- *help* – описание аргумента

Ответы и полезные ссылки

...

ОТВЕТЫ

В случае передачи в метод **zip** коллекций (итераторов по коллекциям) разной длины, итерирование закончится при достижении конца наименьшей коллекции.

В случае итераторов важно, какой итератор будет исчерпан первым.

```
>>> i1, i2 = iter(range(5)), iter(range(2))
```

```
>>> for x, y in zip(i1, i2):  
...     pass
```

```
>>> assert next(i1) == 3
```


ОТВЕТЫ

Очистить список, не удалив его самого, можно так:

```
>>> x = []  
>>> del x[:]  
>>> x[:] = []    # эквивалентная запись
```

Строки модуля интерпретируются последовательно. При попытке создания объекта **f** – функции аргументы по умолчанию будут также интерпретированы и сохранены в данном объекте. Поскольку функция **g** объявлена ниже, произойдет исключение **NameError**, что такого имени нету. Обратите внимание, что значения аргументов по умолчанию создаются *только один раз* при загрузке модуля.

ОТВЕТЫ

Аргументы функций

Если функция имеет неименованный variadic параметр (Python 3), но variadic аргументы переданы, то произойдет исключение **TypeError**.

Модифицировать локальные переменные через словарь **locals()** крайне не рекомендуется, хоть исключения и не будет. Следует только получать значения.

Lambda-функция

Пустая lambda-функция имеет следующий вид:

```
>>> lambda: None
```

ОТВЕТЫ

Форматирование строк

Если количество аргументов для подстановки не совпадает с количеством в шаблоне или один из требуемых именованных аргументов не передан, произойдет исключение **TypeError**.

Однако для подстановки можно передать словарь, в котором значений больше, чем требуется. Ошибки в таком случае не будет.

ПОЛЕЗНЫЕ ССЫЛКИ

Множество *shortcuts* можно найти в книге Pilgrim, М. Dive Into Python 3.

Подробнее механизм импортов описан здесь (Python 3):

- <https://docs.python.org/3.7/library/modules.html>
- <https://docs.python.org/3.7/tutorial/modules.html>
- <https://docs.python.org/3.7/reference/import.html>

Для Python 2 документация расположена по следующим ссылкам:

- <https://docs.python.org/2.7/library/modules.html>
- <https://docs.python.org/2.7/tutorial/modules.html>