Лекция 5: Встроенные коллекции и модуль `collections`

Сергей Лебедев

sergei.a.lebedev@gmail.com

5 октября 2015 г.

- Встроенных коллекций в Python немного: `tuple`, `list`, `set` и `dict`.
- Мы уже кратко обсуждали базовые методы работы с ними.
  - Создать коллекцию можно с помощью литералов или конструктора типов:

```
>>> tuple(), (0, ) * 2
((), (0, 0))
>>> list(), [0] * 2
([], [0, 0])
```

```
>>> set()
{}
>>> dict(), {}
({}, {})
```

  - Функция `len` возвращает длину переданной коллекции.
  - `elem in collection` и `elem not in collection` проверяют, содержится ли в коллекции элемент,
  - Удалить элемент по ключу или по индексу можно с помощью оператора **`del`**.
- О чём тут ещё можно говорить?

Кортеж

- Литералы кортежа — обычные скобки, почти всегда их можно и нужно опускать:

```
>>> x, y = point
>>> date = "October", 5
```

- Эта рекомендация не касается одноэлементных кортежей:

```
>>> # Плохо
>>> xs = 42,
>>> x, = xs
>>> x
42
```

```
>>> # Лучше
>>> xs = (42, )
>>> [x] = xs
>>> x
42
```

- С помощью слайсов можно выделить подпоследовательность в любой коллекции, в частности, в кортеже:

```
>>> person = ("George", "Carlin", "May", 12, 1937)
>>> name, birthday = person[:2], person[2:]
>>> name
('George', 'Carlin')
>>> birthday
('May', 12, 1937)
```

- Избавиться от “магических” констант помогут именованные слайсы:

```
>>> NAME, BIRTHDAY = slice(2), slice(2, None)
(None, 2, None)
>>> person[NAME]
('George', 'Carlin')
>>> person[BIRTHDAY]
('May', 12, 1937)
```

- Напоминание: функция `reversed` принимает одну последовательность и возвращает другую, перечисляющую элементы первой в обратном порядке:

```
>>> tuple(reversed((1, 2, 3)))
(3, 2, 1)
```

- Эту операцию также можно выразить через слайс с отрицательным шагом:

```
>>> (1, 2, 3)[::-1]
(3, 2, 1)
```

Вопрос

Зачем нужна “лишняя” функция `reversed`?

- Кортежи можно конкатенировать с помощью бинарной операции `+`. Результатом конкатенации *всегда* является новый кортеж:

```
>>> xs, ys = (1, 2), (3, )
>>> id(xs), id(ys)
(4416359176, 4413860384)
>>> id(xs + ys)
4416309504
```

- Сравнение кортежей происходит в лексикографическом порядке, причём длина учитывается, только если одна последовательность является префиксом другой:

```
>>> (1, 2, 3) < (1, 2, 4)
True
>>> (1, 2, 3, 4) < (1, 2, 4)
True
>>> (1, 2) < (1, 2, 42)
True
```

- Функция `namedtuple` возвращает тип кортежа, специализированный на фиксированное множество полей:

```
>>> from collections import namedtuple
>>> Person = namedtuple("Person", ["name", "age"])
>>> p = Person("George", age=77)
>>> p._fields
('name', 'age')
>>> p.name, p.age
('George', 77)
```

- Несколько полезных методов при работе с именованными кортежами:

```
>>> p._asdict()
OrderedDict([('name', 'George'), ('age', 77)])
>>> p._replace(name="Bill")
Person(name='Bill', age=77)
```

Список

- Напоминание: синтаксис инициализации создаёт список указанной длины и заполняет его начальным значением:

```
>>> [0] * 2
[0, 0]
>>> [""] * 2
["", ""]
```

- Важно понимать, что копирование начального значения при этом не происходит:

```
>>> chunks = [[0]] * 2  # матрица 2 x 1 из нулей
>>> chunks
[[0], [0]]
>>> chunks[0][0] = 42
>>> chunks
[[42], [42]]
```

- Как правильно инициализировать `chunks`?

Методы работы со списками: добавление элементов

- Методы `append` и `extend` добавлют в конец списка один элемент или произвольную последовательность соответственно:

```
>>> xs = [1, 2, 3]
>>> xs.append(42)          # ==> [1, 2, 3, 42]
>>> xs.extend({-1, -2})   # ==> [1, 2, 3, 42, -2, -1]
```

- Вставить элемент перед элементом с указанным индексом можно с помощью метода `insert`:

```
>>> xs = [1, 2, 3]
>>> xs.insert(0, 4)       # ==> [4, 1, 2, 3]
>>> xs.insert(-1, 42)     # ==> [4, 1, 2, 42, 3]
```

- Можно также заменить подпоследовательность на элементы другой последовательности:

```
>>> xs = [1, 2, 3]
>>> xs[:2] = [0] * 2      # ==> [0, 0, 3]
```

- Конкатенация списков работает аналогично конкатенации кортежей: результатом *всегда* является новый список.

```
>>> xs, ys = [1, 2], [3]
>>> id(xs), id(ys)
(4416336136, 4416336008)
>>> id(xs + ys)
4415376136
```

- В отличие от кортежей списки поддерживают *inplace* конкатенацию:

```
>>> xs += ys  # ≈ xs = xs.extend(ys)
>>> id(xs)
4416336136
```

- Вопрос для любителей `+=`:

```
>>> xs = []
>>> def f():
...     xs += [42]
...
>>> f()
```

Методы работы со списками: удаление элементов

- С помощью оператора **`del`** можно удалить не только один элемент по его индексу, но и целую подпоследовательность:

```
>>> xs = [1, 2, 3]
>>> del xs[:2]
>>> xs  # ==> [3]
```

```
>>> xs = [1, 2, 3]
>>> del xs[:]
>>> xs  # ==> []
```

- Иногда при удалении элемента по индексу может быть удобно также получить его значение:

```
>>> xs = [1, 2, 3]
>>> xs.pop(1)
2
>>> xs  # ==> [1, 3]
```

- Удалить первое вхождение элемента в списке можно с помощью метода `remove`:

```
>>> xs = [1, 1, 0]
>>> xs.remove(1)
>>> xs  # ==> [1, 0]
```

- Уже знакомая нам функция `reversed` работает со списками:

```
>>> list(reversed([1, 2, 3]))
[3, 2, 1]
```

- Можно также перевернуть список *inplace*:

```
>>> xs = [1, 2, 3]
>>> xs.reverse()
>>> xs
[3, 2, 1]
```

- Обратите внимание, что в отличие от функции `reversed` *inlpace* операция возвращает **None**, подсказывая пользователю, что список был изменён.

- Аналогичные взаимоотношения у функции `sorted` и метода `sort`[1]:

```
>>> xs = [3, 2, 1]
>>> sorted(xs), xs
([1, 2, 3], [3, 2, 1])
>>> xs.sort()
>>> xs
[1, 2, 3]
```

- Функции `sorted` и методу `sort` можно опционально указать направление сортировки, а также функцию-ключ:

```
>>> xs = [3, 2, 1]
>>> xs.sort(key=lambda x: x % 2, reverse=True)
>>> xs
[3, 1, 2]
```

[1] CPython использует довольно хитрый алгоритм сортировки Timsort за авторством Тима Питерса. В репозитории CPython можно найти документ, подробно описывающий мотивацию и основные идеи Timsort: http://bugs.python.org/file4451/timsort.txt.

```
>>> stack = []
>>> stack.append(1)
>>> stack.append(2)
>>> stack.pop()
2
>>> stack
[1]
```

```
>>> q = []
>>> q.append(1)
>>> q.append(2)
>>> q.pop(0)
1
>>> q
[2]
```

Вопрос

В чём проблема с `q.pop(0)`?

- Тип deque реализует двустороннюю очередь:

```
>>> from collections import deque
>>> q = deque()
```

- Добавление и удаление элемента с обеих сторон очереди работает за константное время, индексирование — за время, линейное от размера очереди.
- Резюме операций с deque:

```
>>> q = deque([1, 2, 3])
>>> q.appendleft(0)
>>> q
deque([0, 1, 2, 3])
>>> q.append(4)
>>> q
deque([0, 1, 2, 3, 4])
>>> q.popleft()
0
>>> q[0]
1
```

Несколько слов об ограниченной deque

- Конструктор `deque` принимает опциональный аргумент `maxlen`, ограничивающий максимальную длину очереди.
- При добавлении элемента к ограниченной очереди лишние элементы “вываливаются” с противоположной стороны:

```
>>> q = deque([1, 2], maxlen=2)
>>> q.appendleft(0)
>>> q
deque([0, 1], maxlen=2)
>>> q.append(2)
>>> q
deque([1, 2], maxlen=2)
```

Множество

- Базовые операции при работе с множествами:

```
>>> xs, ys, zs = {1, 2}, {2, 3}, {3, 4}
>>> set.union(xs, ys, zs)          # xs | ys | zs
{1, 2, 3, 4}
>>> set.intersection(xs, ys, zs)  # xs & ys & zs
set()
>>> set.difference(xs, ys, zs)    # xs - ys - zs
{1}
```

- Операции сравнения множеств:

```
>>> xs.isdisjoint(ys)
False
>>> xs <= ys              # xs ⊆ ys
False
>>> xs < xs               # xs ⊂ xs
False
>>> xs | ys >= xs         # xs∪ys ⊇ xs
True
```

Методы работы с множествами: добавление элементов

- Добавить один элемент в множестве можно с помощью метода `add`, добавить последовательность элементов — с помощью метода `update`:

```
>>> seen = set()
>>> seen.add(42)
>>> seen
{42}
>>> seen.update([1, 2])
>>> seen
{1, 2, 42}
```

- Метод `update` принимает произвольное количество аргументов:

```
>>> seen.update([], [1], [2], [3])
>>> seen
{1, 2, 42, 3}
```

Методы работы с множествами: удаление элементов

- Метод `remove` удаляет из множества существующий элемент или поднимает исключение, если элемент во множестве не содержится:

```
>>> seen = {1, 2, 3}
>>> seen.remove(3)
>>> seen
{1, 2}
>>> seen.remove(100500)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 100500
```

- В отличие от метода `remove` метод `discard` удаляет элемент, только если он содержится во множестве:

```
>>> seen.discard(100500)
>>> seen
{1, 2}
```

- Удалить все элементы из множества можно с помощью метода `clear`.

- Напоминание: множество в Python — это хеш-сет, то есть оно может содержать только элементы, которые можно захешировать.
- Можно ли сделать множество множеств?

```
>>> {set(), set()}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
```

- Тип `frozenset` описывает неизменяемое множество:

```
>>> {frozenset(), frozenset()}
{frozenset()}
```

Капитан сообщает

Объекты типа `frozenset` поддерживают все операции типа `set` кроме операций добавления и удаления элементов.

Словарь

- Конструктор `dict` позволяет создать словарь без использования литералов:

```
>>> d = dict(foo="bar")
>>> dict(d)                  # (shallow) копия
{'foo': 'bar'}
>>> dict(d, boo="baz")  # копирование с добавлением ключей
{'boo': 'baz', 'foo': 'bar'}
```

- Можно также сконструировать словарь из последовательности ключей и значения по умолчанию:

```
>>> dict.fromkeys(["foo", "bar"])
{'foo': None, 'bar': None}
>>> dict.fromkeys("abcd", 0)
{'d': 0, 'a': 0, 'b': 0, 'c': 0}
```

Вопрос

Эквивалентны ли эти два выражения?

```
dict.fromkeys("abcd", [])    {ch: [] for ch in "abcd"}
```

Методы работы со словарями: проекции

- Методы `keys`, `values` и `items` возвращают проекции содержимого словаря:

```
>>> d = dict.fromkeys(["foo", "bar"], 42)
>>> d.keys()
dict_keys(['foo', 'bar'])
>>> d.values()
dict_values([42, 42])
>>> d.items()
dict_items([('foo', 42), ('bar', 42)])
```

- Проекции поддерживают стандратные операции последовательности:

```
>>> len(d.items())
2
>>> 42 in d.values()
True
```

- Проекция `keys` дополнительно реализует некоторые операции множества:

```
>>> d.keys() & {"foo"}
{'foo'}
```

- Проекции можно использовать для итерации в цикле **for** или генераторе:

```
>>> {v for v in d.values()}
{42}
```

- Модифицировать содержимое словаря в процессе итерации нельзя:

```
>>> for k in d:
...     del d[k]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: dictionary changed size during iteration
```

- Если очень хочется, можно сделать копию проекции и итерироваться по ней:

```
>>> for k in set(d):
...     del d[k]
...
>>> d
{}
```

Напоминание: получить значение элемента по ключу можно с помощью синтаксиса `d[key]` или метода `get`:

```
>>> d = {"foo": "bar"}
>>> d["foo"]
'bar'
>>> d["boo"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'boo'
>>> d.get("boo", 42)
42
```

Вопрос

Зачем нужен метод `get`, если можно написать

```
>>> if key not in d:
...     value = default
... else:
...     value = d[key]
```

Методы работы со словарями: добавление элементов

- Напоминание: записать значение по ключу можно с помощью синтаксиса `d[key] = value`.
- Метод `setdefault` позволяет за один запрос к хеш-таблице проверить, есть ли в ней значение по некоторому ключу и, если значения нет, установить его в заданное.

```
>>> d = {"foo": "bar"}
>>> d.setdefault("foo", "???")
'bar'
>>> d.setdefault("boo", 42)
42
>>> d
{'boo': 42, 'foo': 'bar'}
```

- Метод `update` добавляет словарь элементы переданной последовательности пар или словаря:

```
>>> d = {}
>>> d.update([("foo", "bar")], boo=42)
>>> d
{'boo': 42, 'foo': 'bar'}
```

Методы работы со словарями: удаление элементов

- Напоминение: удалить значение по ключу можно с помощью оператора `del d[key]`.
- Метод `pop` удаляет значение по ключу и возвращает его в качестве результата, а метод `clear` удаляет из словаря все значения:

```
>>> d = {"foo": "bar", "boo": 42}
>>> d.pop("foo")
'bar'
>>> d
{'boo': 42}
>>> d.clear()
>>> d
{}
```

- Допустим, мы хотим хранить направленный граф в виде “списка” смежности:

```
>>> g = {"a": {"b"}, "b": {"c"}}
>>> g["a"]
{'b'}
```

- Как добавить в граф ребра `("b", "a")` и `("c", "a")`?

```
>>> g["b"].add("a")
>>> g["c"].add("a")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'c'
```

- Избавит от боли `defaultdict` — словарь с функцией-инициализатором:

```
>>> from collections import defaultdict
>>> g = defaultdict(set, **{"a": {"b"}, "b": {"c"}})
>>> g["c"].add("a")
>>> g
defaultdict(<class 'set'>,
            {'b': {'c'}, 'c': {'a'}, 'a': {'b'}})
```

- Порядок ключей в обычном словаре не определён:

```
>>> d = dict([("foo", "bar"), ("boo", 42)])
>>> list(d)
['boo', 'foo']
```

- `OrderedDict` — словарь с ключами, упорядоченными по времени добавления:

```
>>> from collections import OrderedDict
>>> d = OrderedDict([("foo", "bar"), ("boo", 42)])
>>> list(d)
['foo', 'boo']
```

- Изменение значения по ключу не влияет на порядок ключей в словаре:

```
>>> d["boo"] = "???"  # не изменит порядок ключей
>>> d["bar"] = "???"
>>> list(d)
['foo', 'boo', 'bar']
```

[2] http://python.org/dev/peps/pep-0372

- Тип `Counter` — это специализация словаря для подсчёта объектов, которые можно захешировать:

```
>>> from collections import Counter
>>> c = Counter(["foo", "foo", "foo", "bar"])
>>> c["foo"] += 1
>>> c
Counter({'foo': 4, 'bar': 1})
```

- Счётчик поддерживает все методы словаря, а также реализует несколько дополнительных:

```
>>> c.pop("foo")
4
>>> c["boo"]  # не поднимает исключение
0
```

- Метод `elements` перечисляет элементы счётчика в произвольном порядке. Элементы, для которых частота равна нулю или отрицательна, игнорируются:

```
>>> c = Counter(foo=4, bar=-1)
>>> list(c.elements())
['foo', 'foo', 'foo', 'foo']
```

- Метод `most_common` возвращает заданное число самых частых элементов:

```
>>> c.most_common(1)
[('foo', 4)]
```

- Методы `substract` и `update` позволяют поэлементно обновить значения счётчика:

```
>>> c.update(["bar"])
>>> c
Counter({'foo': 4, 'bar': 0})
>>> c.subtract({"foo": 2})
>>> c
Counter({'foo': 2, 'bar': 0})
```

```
>>> c1 = Counter(foo=4, bar=-1)
>>> c2 = Counter(foo=2, bar=2)
>>> c1 + c2  # c1[k] + c2[k]
Counter({'foo': 6, 'bar': 1})
>>> c1 - c2  # c1[k] - c2[k]
Counter({'foo': 2})
>>> c1 & c2  # min(c1[k], c2[k])
Counter({'foo': 2})
>>> c1 | c2  # max(c1[k], c2[k])
Counter({'foo': 4, 'bar': 2})
```

Замечание

Результат любой из бинарных операций всегда содержит только ключи с положительными частотами.