Оглавление

Неделя 2

Структуры данных и функции

2.1. Коллекции

2.1.1. Списки и кортежи

Коллекция --- это переменная-контейнер, в которой может содержаться какое-то количество объектов, где объекты могут быть одного типа или разного. В случае списков это упорядоченные наборы элементов, которые могут быть разных типов. Сами списки определяются с помощью квадратных скобочек или с помощью вызова литерала `list`. Вы также можете создать список из одинаковых значений с помощью умножения. Несмотря на вышесказанное, чаще всего списки содержат переменные одного типа. Также списки могут содержать другие коллекции, как, например, `user_data`. Однако для таких данных чаще всего используются кортежи, о которых будет сказано позже.

```
empty_list = []
empty_list = list()

none_list = [None] * 10

collections = ['list', 'tuple', 'dict', 'set']

user_data = [
    ['Elena', 4.4],
    ['Andrey', 4.2]
]
```

Для получения длины списка вызывают встроенную функцию `len()`. В Python не нужно явно указывать размер списка или вручную выделять на него память. Размер списка хранится в структуре, с помощью которой реализован тип список, поэтому длина вычисляется за константное время.

```
len(collections)
```

```
4
```

Чтобы обратиться к конкретному элементу списка, мы используем тот же механизм,

что и для строк --- обращаемся к элементу по индексу. Нумерация элементов начинается с нуля.

```
print(collections)

print(collections[0])
print(collections[-1])
```

```
['list', 'tuple', 'dict', 'set']
list
set
```

Можно использовать доступ по индексу для присваивания (изменения элементов):

```
collections[3] = 'frozenset'
print(collections)
```

```
['list', 'tuple', 'dict', 'frozenset']
```

Обращение к несуществующему индексу приводит к ошибке `IndexError: list index out of range`.
С помощью оператора `in` можно проверить, существует ли какой-то объект в списке:

```
'tuple' in collections
```

```
True
```

Срезы в списках работают точно так же, как и в строках. Создадим список из 10 элементов с помощью встроенной функции range и поэкспериментируем на нём со срезами:

```
range_list = list(range(10))
print(range_list)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
range_list[1:3]
```

```
[1, 2]
```

```
range_list[::2]
```

```
[0, 2, 4, 6, 8]
```

```
range_list[::-1]
```

```
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

```
range_list[5:1:-1]
```

```
[5, 4, 3, 2]
```

Важно знать, что при получении среза создаётся новый объект --- новый список:

```
range_list[:] is range_list
```

```
False
```

Как и все коллекции, спсиски поддерживают протокол итерации --- мы можем итерироваться по элементам списка, используя цикл `for`.

```
collections = ['list', 'tuple', 'dict', 'set']

for collection in collections:
    print('Learning {}...'.format(collection))
    # используем функцию format для форматирования строк
```

```
Learning list...
Learning tuple...
Learning dict...
Learning set...
```

Обратите внимание, что итерация производится именно по элементам списка, а не по индексам, как во многих других языках.

Часто бывает нужно получить индекс текущего элемента при итерации. Для этого можно использовать встроенную функцию `enumerate`, которая возвращает индекс и текущий элемент.

```
for idx, collection in enumerate(collections):
    print('#{} {}'.format(idx, collection))
```

```
#0 list
#1 tuple
#2 dict
#3 set
```

Так как списки являются изменяемой структурой данных, мы можем добавлять и удалять элементы. Например, мы можем добавить в наш список `collections` элемент `'OrderedDict'`.

```
collections.append('OrderedDict')

print(collections)
```

```
['list', 'tuple', 'dict', 'set', 'OrderedDict']
```

Если вам нужно расширить список другим списком, вы можете использовать метод `extend`, который добавляет переданный список в конец вашего списка.

```
collections.extend(['ponyset', 'unicorndict'])

print(collections)
```

```
['list', 'tuple', 'dict', 'set', 'OrderedDict', 'ponyset', 'unicorndict']
```

Также можно использовать перегруженный оператор +, который также добавляет переменную в конец вашего списка:

```
collections += [None]

print(collections)
```

```
['list', 'tuple', 'dict', 'set', 'OrderedDict', 'ponyset',
'unicorndict', None]
```

Для удаление элемента из списка можно использовать ключевое слово del.

```
del collections[4]

print(collections)
```

```
['list', 'tuple', 'dict', 'set', 'ponyset', 'unicorndict', None]
```

Часто нам нужно найти минимальный/максимальный элемент в массиве или посчитать сумму всех элементов. Вы можете это сделать при помощи встроенных функций `min`, `max`, `sum`.

```
numbers = [4, 17, 19, 9, 2, 6, 10, 13]

print(min(numbers))
print(max(numbers))
print(sum(numbers))
```

```
2
19
80
```

Часто бывает полезно преобразовать список в строку, для этого можно использовать метод `str.join()`:

```
tag_list = ['python', 'course', 'coursera']

print(', '.join(tag_list))
```

```
python, course, coursera
```

Ещё одна часто встречающаяся операция со списками --- это сортировка. В Python существует несколько методов сортировки.

Для начала создадим случайный список с помощью функции модуля `random`:

```
import random


numbers = []
for _ in range(10): # переменную для итерации называли _, т.к.
                   # сама эта переменная нам не важна
    numbers.append(random.randint(1, 20))

print(numbers)
```

```
[13, 9, 10, 1, 1, 13, 14, 1, 16, 4]
```

Для сортировки списка в Python есть два способа: стандартная функция `sorted`, которая возвращает новый список, полученный сортировкой исходного, и метод списка `.sort()`, который сортирует in-place. Для сортировки используется алгоритм TimSort.

```
print(sorted(numbers))
print(numbers)
```

```
[1, 1, 1, 4, 9, 10, 13, 13, 14, 16]
[13, 9, 10, 1, 1, 13, 14, 1, 16, 4]
```

```
numbers.sort()
print(numbers)
```

```
[1, 1, 1, 4, 9, 10, 13, 13, 14, 16]
```

Если нужно отсортировать список в обратном порядке:

```
print(sorted(numbers, reverse=True))
```

```
[16, 14, 13, 13, 10, 9, 4, 1, 1, 1]
```

```
numbers.sort(reverse=True)
print(numbers)
```

```
[16, 14, 13, 13, 10, 9, 4, 1, 1, 1]
```

Для той же цели можно использовать встроенную функцию `reversed`, которая возвращает так называемый reverse iterator. Об итераторах будет сказано позднее, пока достаточно понимать, что это объект, который поддерживает протокол итерации. Данный объект можно преобразовать в список, и получится список с обратным порядком элементов.

Кроме методов, которые мы обсудили выше, существует также много других, о которых можно прочесть в документации:

- append
- clear
- copy
- count
- extend
- index
- insert
- pop
- remove
- reverse
- sort

Перейдём к кортежам. Кортежи --- это неизменяемые списки (мы не можем ни добавлять, ни удалять элементы из кортежа). Кортежи определяются с помощью круглых скобок или литерала `tuple`.

```
empty_tuple = ()
empty_tuple = tuple()
```

Например, мы можем создать кортеж `immutables` и поместить туда неизменяемые типы.

```
immutables = (int, str, tuple)
```

Если попробовать заменить нулевой элемент на `float`, Python выдаст ошибку, потому что кортежи неизменяемы.

```
immutables[0] = float
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-34-70298ebdccb5> in <module>()
----> 1 immutables[0] = float

TypeError: 'tuple' object does not support item assignment
```

Но несмотря на то, что сами кортежи неизменяемые, объекты внутри них могут быть изменяемыми. Например, если кортеж содержит список, мы можем добавлять элементы в этот список.

```
blink = ([], [])
blink[0].append(0)

print(blink)
```

```
([0], [])
```

Важная особенность кортежей --- к ним применяется фунцкия `hash`, и поэтому они могут использоваться в качестве ключей в словарях, о которых мы поговорим позднее.

```
hash(tuple())
```

```
3527539
```

Будьте внимательны при определении кортежа из одного элемента --- не забывайте писать запятую. Если вы забудете про нее, Python сочтет вашу переменную типом `int`.

```
one_element_tuple = (1,)
guess_what = (1)

type(guess_what)
```

```
int
```

2.1.2. Списки. Пример программы

Разберём задачу на применение списков --- поиск медианы случайного списка. Медиана --- это значение в отсортированном списке, которое лежит ровно посередине, таким образом, половина значений --- слева от него, и половина значений --- справа.

Сначала создадим случайный список со случайным (чтобы было интереснее) количеством элементов.

```
import random


numbers = []
numbers_size = random.randint(10, 15)

# мы не будем использовать переменную, которую используем
# для итерации, поэтому назовём её _
for _ in range(numbers_size):
    numbers.append(random.randint(10, 20))
    # randint возвращает случайное целое
    # число в переданном ей интервале

print(numbers)
```

```
[16, 10, 12, 16, 16, 10, 11, 18, 14, 10]
```

Отсортируем наш список:

```
numbers.sort()
```

```
[10, 10, 10, 11, 12, 14, 16, 16, 16, 18]
```

По определению медианы, она равна среднему элементу в отсортированном списке, если количество элементов нечётное. Если число элементов чётное, то медиана --- это среднее арифметическое от двух средних элементов. Мы заведем переменную `half_size`, в которую положим значение, равное половине длины списка. Также заведём переменную `median`, сначала имеющую значение `None`.

```
half_size = len(numbers) // 2
median = None
```

Теперь запишем условие на чётность элементов и найдём медиану по определению для каждого случая:

```
if numbers_size % 2 == 1:
    median = numbers[half_size]
else:
    median = sum(numbers[half_size - 1:half_size + 1]) / 2
```

Посмотрим, что получилось в итоге:

```
numbers.sort()

half_size = len(numbers) // 2
median = None

if numbers_size % 2 == 1:
    median = numbers[half_size]
else:
    median = sum(numbers[half_size - 1:half_size + 1]) / 2

print(median)
```

```
13.0
```

Чтобы проверить наш результат, можно воспользоваться встроенным модулем `statistics`.

```
import statistics

statistics.median(numbers)
```

```
13.0
```

2.1.3. Словари

Словари являются важнейшей структурой данных в Python-е. Они позволяют хранить данные в формате ключ-значение. Чтобы определить словарь, нужно использовать литерал фигурные скобки или просто вызвать `dict`. Если мы хотим, определяя словарь, сразу добавить в него данные, пишем ключ-значение через двоеточие.

```
empty_dict = {}
empty_dict = dict()

collections_map = {
    'mutable': ['list', 'set', 'dict'],
    'immutable': ['tuple', 'frozenset']
}
```

Доступ к значению по ключу осуществляется за константное время, то есть не зависит от размера словаря. Это достигается с помощью алгоритма хеширования.

Если пытаться получить доступ по ключу, которого не существует, Python выдаст ошибку `KeyError`. Однако, часто бывает полезно попытаться достать значение по ключу из словаря, а в случае отсутствия ключа вернуть какое-то стандартное значение. Для этого есть встроенный метод `get`.

```
print(collections_map['immutable'])
```

```
['tuple', 'frozenset']
```

```
print(collections_map['irresistible'])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-40-fae0f6f2a221> in <module>()
----> 1 print(collections_map['irresistible'])

KeyError: 'irresistible'
```

```
print(collections_map.get('irresistible', 'not found'))
```

```
not found
```

Проверка на вхождения ключа в словарь так же осуществляется за константное время и выполняется с помощью ключевого слова `in`:

```
'mutable' in collections_map
```

```
True
```

Так как словарь является изменяемой структурой данных, мы можем добавлять и удалять элементы из него. Например, мы можем определить словарь `beatles_map`, который содержит знаменитых музыкантов и их инструменты, и добавить в него Ринго с ударными, просто используя доступ по ключу. Чтобы удалить ключ и значение из словаря, можно использовать уже знакомый вам оператор `del`.

```
beatles_map = {
    'Paul': 'Bass',
    'John': 'Guitar',
    'George': 'Guitar',
}

print(beatles_map)
```

```
{'Paul': 'Bass', 'John': 'Guitar', 'George': 'Guitar'}
```

```
beatles_map['Ringo'] = 'Drums'

print(beatles_map)
```

```
{'Paul': 'Bass', 'John': 'Guitar', 'George': 'Guitar', 'Ringo': 'Drums'}
```

```
del beatles_map['John']

print(beatles_map)
```

```
{'Paul': 'Bass', 'George': 'Guitar', 'Ringo': 'Drums'}
```

Также, чтобы добавить какой-то ключ-значение в словарь, можно использовать встроенный метод `update`, который принимает словарь и дополняет им (а также обновляет в случае одинаковых ключей) исходный словарь.

```
beatles_map.update({
    'John': 'Guitar'
})

print(beatles_map)
```

```
{'Paul': 'Bass', 'George': 'Guitar', 'Ringo': 'Drums', 'John': 'Guitar'}
```

Чтобы удалить ключ-значение из словаря и одновременно вернуть значение, используют метод `pop`:

```
# удаляем Ринго, нам возвращаются его ударные
print(beatles_map.pop('Ringo'))

print(beatles_map)
```

```
Drums
{'Paul': 'Bass', 'George': 'Guitar', 'John': 'Guitar'}
```

Часто бывает необходимо не только попробовать проверить, существует ли ключ в словаре, но и в случае неудачи добавить эту новую пару ключ-значение. Для этого есть метод `setdefault`:

```
unknown_dict = {}

print(unknown_dict.setdefault('key', 'default'))
```

```
default
```

```
print(unknown_dict)
```

```
{'key': 'default'}
```

Если вызвать `setdefault` и в качестве дефолтного значения передать `new_default`, вернётся значение, которое уже лежит в словаре --- значение `default`:

```
print(unknown_dict.setdefault('key', 'new_default'))
```

```
default
```

Словари, как и все коллекции, поддерживают протокол итерации. С помощью цикла `for` можно итерироваться по ключам словаря:

```
print(collections_map)

for key in collections_map:
    print(key)
```

```
{'mutable': ['list', 'set', 'dict'], 'immutable': ['tuple', 'frozenset']}
mutable
immutable
```

Если нам нужно итерироваться не по ключам, а по ключам и значениям сразу, можно использовать метод словаря `items`, который возвращает ключи и значения.

```
for key, value in collections_map.items():
    print('{} - {}'.format(key, value))
```

```
mutable - ['list', 'set', 'dict']
immutable - ['tuple', 'frozenset']
```

Если нужно итерироваться по значениям, используйте логично метод `values`, который возвращает именно значения. Также существует симметричный метод `keys`, который возвращает итератор ключей.

```
for value in collections_map.values():
    print(value)
```

```
['list', 'set', 'dict']
['tuple', 'frozenset']
```

Важная особенность словарей в Python-е: они содержат ключи и значения в неупорядоченном виде. Однако, в Python-е существует тип `OrderedDict` (содержится в модуле `collections`), который гарантирует вам, что ключи хранятся именно в том порядке, в каком вы их добавили в словарь.

```
from collections import OrderedDict


ordered = OrderedDict()

for number in range(10):
    ordered[number] = str(number)

for key in ordered:
    print(key)
```

```
0
1
2
3
4
5
6
7
8
9
```

2.1.4. Словари. Пример программы

Разберём следующую задачу на словари: найти 3 самых часто встречающихся слова в Zen of Python.

```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
```

```
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Скопируем этот текст и поместим его в переменную `zen`. После этого заведём переменную `zen_map`, в которой будем хранить слова, которые уже нашли, и то, сколько раз их уже нашли. Будем итерироваться с помошью метода `split()`, который разобьёт нашу строку по пробельным символам. Очищать слова от знаков препинания и пробельных символов будем с помощью метода `strip()`.

```
zen_map = dict()

for word in zen.split():
    cleaned_word = word.strip('.,!-*').lower()
    # добавляем слово, если его ещё нет в zen_map:
    if cleaned_word not in zen_map:
        zen_map[cleaned_word] = 0

    zen_map[cleaned_word] += 1

print(zen_map)
```

```
{'beautiful': 1, 'is': 10, 'better': 8, 'than': 8, 'ugly': 1,
'explicit': 1, 'implicit': 1, 'simple': 1, 'complex': 2,
'complicated': 1, 'flat': 1, 'nested': 1, 'sparse': 1, 'dense': 1,
'readability': 1, 'counts': 1, 'special': 2, 'cases': 1, "aren't": 1,
'enough': 1, 'to': 5, 'break': 1, 'the': 5, 'rules': 1, 'although':
3, 'practicality': 1, 'beats': 1, 'purity': 1, 'errors': 1, 'should':
2, 'never': 3, 'pass': 1, 'silently': 1, 'unless': 2, 'explicitly':
1, 'silenced': 1, 'in': 1, 'face': 1, 'of': 2, 'ambiguity': 1,
'refuse': 1, 'temptation': 1, 'guess': 1, 'there': 1, 'be': 3, 'one':
3, 'and': 1, 'preferably': 1, 'only': 1, 'obvious': 2, 'way': 2,
'do': 2, 'it': 2, 'that': 1, 'may': 2, 'not': 1, 'at': 1, 'first': 1,
"you're": 1, 'dutch': 1, 'now': 2, 'often': 1, 'right': 1, 'if': 2,
'implementation': 2, 'hard': 1, 'explain': 2, "it's": 1, 'a': 2,
'bad': 1, 'idea': 3, 'easy': 1, 'good': 1, 'namespaces': 1, 'are': 1,
'honking': 1, 'great': 1, '': 1, "let's": 1, 'more': 1, 'those': 1}
```

На выходе имеем словарь, в котором ключами являются слова, а значениями --- сколько раз слова встретились в тексте. Теперь найдём самые частотные слова. В переменную `zen_items` поместим список кортежей (ключ, значение) с помощью метода `items()`. Затем отсортируем список по *вторым* элементам в кортеже, используя модуль `operator`. В метод `sorted()` в качестве аргумента `key` передадим `operator.itemgetter(1)` (т.к. мы сортируем по элементам с индексом 1).

```
import operator

zen_items = zen_map.items()
word_count_items = sorted(
    zen_items, key=operator.itemgetter(1), reverse=True
)

print(word_count_items[:3])
```

```
[('is', 10), ('better', 8), ('than', 8)]
```

Как это часто бывает в Python-е, существует встроенный модуль, который поможет вам решить эту задачу намного быстрее. Импортируем `Counter` из модуля `collections`. Теперь осталось только "очистить" слова и передать их в Counter.

```
from collections import Counter


cleaned_list = []
for word in zen.split():
    cleaned_list.append(word.strip('.,-!').lower())

print(Counter(cleaned_list).most_common(3))
```

```
[('is', 10), ('better', 8), ('than', 8)]
```

2.1.5. Множества

Множество в питоне — это неупорядоченный набор *уникальных объектов*. Множества изменяемы и чаще всего используются для удаления дубликатов и всевозможных проверок на вхождение.

Чтобы объявить пустое множество, можно воспользоваться литералом `set` или использовать фигурные скобки, чтобы объявить множество и одновременно добавить туда какие-то элементы.

```
empty_set = set()
number_set = {1, 2, 3, 3, 4, 5}

print(number_set)
```

```
{1, 2, 3, 4, 5}
```

Чтобы проверить, содержится ли объект в множестве, используется уже знакомое нам ключевое слово `in`. Проверка выполняется за константное время, время выполнения операции не зависит от размера множества. Это достигается за счёт хэширования каждого элемента структуры по аналогии со словарями. По полученному от хэш-функции ключу и

происходит поиск объекта. Таким образом, во множествах могут содержаться только хэшируемые объекты.

```
print(2 in number_set)
```

```
True
```

Чтобы добавить элемент в множество, используется метод `add`. Также множества в Python поддерживают стандартные операции над множествами --- такие как объединение, разность, пересечение и симметрическая разность.

Создадим два множества с чётными и нечётными числами до десяти:

```
odd_set = set()
even_set = set()

for number in range(10):
    if number % 2:
        odd_set.add(number)
    else:
        even_set.add(number)

print(odd_set)
print(even_set)
```

```
{1, 3, 5, 7, 9}
{0, 2, 4, 6, 8}
```

Теперь найдём объединение и пересечение этих множеств:

```
union_set = odd_set | even_set
union_set = odd_set.union(even_set)

print(union_set)
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
intersection_set = odd_set & even_set
intersection_set = odd_set.intersection(even_set)

print(intersection_set)
```

```
set()
```

Найдём разность двух множеств:

```
difference_set = odd_set - even_set
difference_set = odd_set.difference(even_set)

print(difference_set)
```

```
{1, 3, 5, 7, 9}
```

Или симметрическую разность:

```
symmetric_difference_set = odd_set ^ even_set
symmetric_difference_set = odd_set.symmetric_difference(even_set)

print(symmetric_difference_set)
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Множества --- изменяемая структура данных, поэтому можно как добавлять туда элементы, так и удалять. Для удаления конкретного элемента существует метод `remove`, для удаления любого элемента можно использовать `pop`. Остальные методы можно посмотреть в `help` или документации.

```
even_set.remove(2)
print(even_set)
```

```
{0, 4, 6, 8}
```

```
even_set.pop()
```

```
0
```

Также в питоне существует неизменяемый аналог типа `set` --- тип `frozenset`.

```
frozen = frozenset(['Anna', 'Elsa', 'Kristoff'])

frozen.add('Olaf')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-64-962f221e1321> in <module>()
      1 frozen = frozenset(['Anna', 'Elsa', 'Kristoff'])
      2
----> 3 frozen.add('Olaf')

AttributeError: 'frozenset' object has no attribute 'add'
```

2.1.6. Множества. Пример программы

Задача на множества: через сколько итераций функция `random.randint(1, 10)` выдаст повтор?

Будем добавлять неповторяющиеся случайные числа в множество `random_set`. Если очередное число уже есть в `random_set` --- выйдем из цикла. Затем посчитаем длину множества (и прибавим 1, т.к. не учли последнее число).

```
import random


random_set = set()

while True:
    new_number = random.randint(1, 10)
    if new_number in random_set:
        break

    random_set.add(new_number)


print(len(random_set) + 1)
```

```
6
```

Таким образом, мы получили повтор через 6 итераций.

2.2. Функции

2.2.1. Функции

Функция --- это блок кода, который можно переиспользовать несколько раз в разных местах программы. Мы можем передавать функции аргументы и получать возвращаемые значения. Чтобы определить функцию в языке Python, нужно использовать литерал `def` и с помощью отступа определить блок кода функции. По PEP8 функции называют `snake_case`-ом.

Объявим функцию, которая возвращает секундную часть текущего времени.

```
from datetime import datetime


def get_seconds():
    """Return current seconds"""
    return datetime.now().second


get_seconds()
```

```
24
```

Чтобы получить документационную строку, можно обратиться к атрибуту `doc`, а имя функции получается с помощью атрибута `name`.

```
get_seconds.__doc__
```

```
'Return current seconds'
```

```
get_seconds.__name__
```

```
'get_seconds'
```

Чаще всего функция определяется с параметрами, т.к. зачастую функции каким-то образом обрабатывают переданные им значения. Определим функцию `split_tags`, которая принимает параметр `tag_string` (например, равный строке с тегами текущего курса). Пусть функция разобьёт эту строку по запятым и вернёт список тегов.

```
def split_tags(tag_string):
    tag_list = []
    for tag in tag_string.split(','):
        tag_list.append(tag.strip())

    return tag_list


split_tags('python, coursera, mooc')
```

```
['python', 'coursera', 'mooc']
```

При вызове этой же функции без параметров получаем ошибку, т.к. функция ожидает заявленный параметр и не может работать без него.

```
split_tags()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-866c00aba286> in <module>()
----> 1 split_tags()

TypeError: split_tags() missing 1 required positional argument: 'tag_string'
```

Выше мы не указывали явно, какого типа параметры функция ожидает, потому что Python --- это язык с динамической типизацией. Но, например, в языке C типы аннотируются, т.е. явно указывается, какого типа должен быть параметр функции и какого типа возвращаемые значения. В Python-е последних версий появилась возможность аннотировать типы, и делается это с помощью двоеточия в случае параметров, а стрелочкой указывают тип возвращаемого значения. Однако, если мы передадим в функцию параметры других типов, код все равно выполнится, потому что Python --- это динамический язык, и аннотация типов призвана лишь помочь программисту или его IDE отловить какие-то ошибки.

```
def add(x: int, y: int) -> int:
    return x + y


print(add(10, 11))
print(add('still ', 'works'))
```

```
21
still works
```

Во многих других языках программирования значения параметра передаются в функцию либо по ссылке, либо по значению (и между двумя этими случаями проводится строгая граница). В Python-е каждая переменная является связью имени с объектом в памяти, и именно эта ссылка на объект передается в функцию. Таким образом, если мы передадим в функцию список и в ходе выполнения функции изменим его, этот список измениться глобально:

```
def extender(source_list, extend_list):
    source_list.extend(extend_list)


values = [1, 2, 3]
extender(values, [4, 5, 6])

print(values)
```

```
[1, 2, 3, 4, 5, 6]
```

Если мы так же попытаемся изменить объект неизменяего типа, он, что логично, не изменится (мы передаем ссылку на объект в памяти, который неизменяем).

```
def replacer(source_tuple, replace_with):
    source_tuple = replace_with


user_info = ('Guido', '31/01')
replacer(user_info, ('Larry', '27/09'))

print(user_info)
```

```
('Guido', '31/01')
```

Однако изменение глобальных переменных внутри функции является плохим тоном, потому что часто бывает не очевидно, какие глобальные объекты как изменяются в каких функциях. В таких ситуациях советуют использовать возвращаемые значения.

В Python-е также существуют именованные аргументы, которые иногда бывают полезны. Если явно указывать имена аргументов, можно передавать их в любом порядке. Кроме того, при вызове функции будет видно, каким аргументам мы присваиваем передаваемые значения.

```
def say(greeting, name):
    print('{} {}!'.format(greeting, name))


say('Hello', 'Kitty')
say(name='Kitty', greeting='Hello')
```

```
Hello Kitty!
Hello Kitty!
```

Важно понимать, что переменные, объявленные вне области видимости функции, нельзя изменять.

```
result = 0

def increment():
    result += 1
    return result

print(increment())
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-10-da69e363a112> in <module>()
      5     return result
      6
----> 7 print(increment())

<ipython-input-10-da69e363a112> in increment()
      2
      3 def increment():
----> 4     result += 1
      5     return result
      6

UnboundLocalError: local variable 'result' referenced before assignment
```

global & nonlocal

В Python-е всё же есть возможность изменять глобальные переменные с помощью `global` или `non local`, но использовать эти особенности не рекомендуется.

Существует также возможность использовать аргументы по умолчанию, которые можно передавать, а можно не передавать. У этих аргументов, могут быть определены какие-то дефолтные значения, которые прописываются при объявлении функции:

```
def greeting(name='it\'s me...'):
    print('Hello, {}'.format(name))


greeting()
```

```
Hello, it's me...
```

Стоит быть внимательными с аргументами по умолчанию, если мы используем в качестве их дефолтного значения объекты изменяемого типа. Например, объявим функцию,

которая прибавляет к списку элемент 1. В качестве значения по умолчанию зададим пустой список:

```
def append_one(iterable=[]):
    iterable.append(1)
    return iterable

print(append_one([1]))
```

```
[1, 1]
```

Что произойдёт, если мы вызовем эту функцию дважды:

```
print(append_one())
print(append_one())
```

```
[1]
[1, 1]
```

Чтобы разобраться, проверим, каковы дефолтные значения параметров функции:

```
print(append_one.__defaults__)
```

```
([1, 1],)
```

Почему так происходит? При определении функции, когда интерпретатор Python-а проходит по файлу с кодом, определяется связь между именем функции и дефолтными значениями. Таким образом, у каждой функции появляется `tuple` с дефолтными значениями. Именно в эти переменные каждый раз и происходит запись. Таким образом, если дефолтные значения являются изменяемыми, в них можно записывать, потому что это обычные переменные.

Чтобы исправить предыдущий пример, возьмём в качестве значения по умолчанию None:

```
def function(iterable=None):
    if iterable is None:
        iterable = []


def function(iterable=None):
    iterable = iterable or []
```

Довольно красивой особенностью Python-а является возможность определения функции, которая принимает разные количества аргументов. Определим функцию `printer`, которая принимает любое количество аргументов --- все аргументы записываются в `tuple args`. Затем функция печатает по порядку все аргументы:

```
def printer(*args):
    print(type(args))

    for argument in args:
        print(argument)


printer(1, 2, 3, 4, 5)
```

```
<class 'tuple'>
1
2
3
4
5
```

Также в аргументах можно развернуть список значений:

```
name_list = ['John', 'Bill', 'Amy']
printer(*name_list)
```

```
<class 'tuple'>
John
Bill
Amy
```

Точно так же это работает в случае со словарями, в данном случае мы можем определить функцию `printer`, которая принимает разное количество именованных аргументов. При этом переменная `kwargs` будет иметь тип `dict`.

```
def printer(**kwargs):
    print(type(kwargs))

    for key, value in kwargs.items():
        print('{}: {}'.format(key, value))


printer(a=10, b=11)
```

```
<class 'dict'>
a: 10
b: 11
```

Точно так же мы можем разыменовывать (разворачивать) словари, используя **:

```
payload = {
    'user_id': 117,
    'feedback': {
        'subject': 'Registration fields',
        'message': 'There is no country for old men'
    }
}

printer(**payload)
```

```
<class 'dict'>
user_id: 117
feedback: {'subject': 'Registration fields', 'message': 'There is no
country for old men'}
```

Это используется практически везде и позволяет вам определять очень гибкие функции, которые принимают различное количество аргументов — именованных и позиционных.

2.2.2. Файлы

Для открытия файлов используется встроенный метод open, которой передаётся путь к файлу --- например, `filename`. Функция open возвращает файловый объект, с которым мы потом можем работать, для того чтобы записывать данные или читать данные из файлов.

```
f = open('filename')
```

Файлы можно открывать по-разному --- на запись, на чтение, на чтение и запись, на дозапись. Делается это с помощью модов, которые также передаются в функцию open. Например, `a` --- это дозапись, `w` --- это, очевидно, запись, `r` — это прочтение, `r+` --- это запись и чтение одновременно. Точно так же можно открывать файл в бинарном виде, то есть работать с бинарными данными --- для этого к моду добавляют букву `b`.

```
text_modes = ['r', 'w', 'a', 'r+']
binary_modes = ['br', 'bw', 'ba', 'br+']
```

```
f = open('filename', 'w')
```

Чтобы записать в файл, применяем к соответствующему файловому объекту метод `write`, передавая ему строку. Метод `write` возвращает количество символов, которые мы записали (или количество байт в случае байтовой строки).

```
f.write('The world is changed.\nI taste it in the water.\n')
```

47

Закрывают файлы так:

```
f.close()
```

В Python принято закрывать файлы, т.к. в противном случае может произойти нечто неприятное --- например, в операционной системе могут закончиться файловые дескрипторы.

Итак, чтобы открыть файл на чтение и запись, нам нужно использовать `r+`. Мы можем читать данные из файла с помощью метода `read`, который по умолчанию читает столько, сколько сможет (если файл слишком большой, он может не поместиться в памяти). Вы также можете указать в методе `read` конкретное количество информации, которое вы хотите прочитать, передав `size`.

```
f = open('filename', 'r+')
f.read()
```

```
'The world is changed.\nI taste it in the water.\n'
```

```
f.tell()
```

```
47
```

Когда мы прочитали весь файл, указатель того, где мы сейчас находимся в файле --- в самом конце (в примере выше --- на 47-ом символе). Если мы попробуем прочитать еще раз, то мы ничего не найдем. Для того чтобы прочитать файл заново, нужно использовать метод `seek` и перенести указатель на начало файла.

```
f.read()
```

```
''
```

```
f.seek(0)
f.tell()
```

```
0
```

```
print(f.read())
f.close() # файлы всегда нужно закрывать
```

```
The world is changed.
I taste it in the water.
```

Для того, чтобы прочесть одну строку из файла, есть метод `readline`, а чтобы разбить файл на строки и поместить их в список --- метод `readlines`.

```
f = open('filename', 'r+')
f.readline()
f.close()
```

```
'The world is changed.\n'
```

```
f = open('filename', 'r+')
f.readlines()
```

```
['The world is changed.\n', 'I taste it in the water.\n']
```

Если закрыть файл, вызов функции `read()` приведёт к ошибке --- закрытый файл нельзя прочитать.

Рекомендуется открывать файлы несколько по-другому --- с помощью контекстного менеджера, который позволяет не заботиться о закрытии файлов. Вы можете открыть файл с помощью оператора `with`, записать файловый объект в переменную `f` и потом работать с файлом внутри этого контекстного блока. После выхода из блока интерпретатор Python закроет файл.

```
with open('filename') as f:
    print(f.read())
```

2.2.3. Функциональное программирование

Функции в Python --- это такие же объекты, как и, например, строки, списки или классы. Их можно передавать в другие функции, возвращать из функций, создавать на лету --- то есть это объекты первого класса.

```
def caller(func, params):
    return func(*params)


def printer(name, origin):
    print('I\'m {} of {}!'.format(name, origin))


caller(printer, ['Moana', 'Motunui'])
```

```
I'm Moana of Motunui!
```

Итак, функции можно передавать в функции. Также их можно создавать внутри других функций.

```
def get_multiplier():
    def inner(a, b):
        return a * b
    return inner


multiplier = get_multiplier()
multiplier(10, 11)
```

```
110
```

Т.к. мы вернули другую функцию, в переменной `multiplier` теперь хранится функция `inner`:

```
print(multiplier.__name__)
```

```
inner
```

Давайте попробуем определить функцию `inner`, которая будет принимать один аргумент и умножать его всегда на то самое число, которое мы передали в `get_multiplier`. Например, мы передаем `get_multiplier` двойку и получаем функцию, которая всегда умножает переданный ей аргумент на двойку. Эта концепция называется "замыканием".

```
def get_multiplier(number):
    def inner(a):
        return a * number
    return inner


multiplier_by_2 = get_multiplier(2)
multiplier_by_2(10)
```

```
20
```

Этот приём очень важен и в дальнейшем будет использоваться в декораторах.

Иногда бывает необходимо применить какую-то функцию к набору элементов. Для этих целей существует несколько стандартных функций. Одна из таких функций — это `map`, которая принимает функцию и какой-то итерабельный объект (например, список) и применяет полученную функцию ко всем элементам объекта.

```
def squarify(a):
    return a ** 2


list(map(squarify, range(5)))
```

```
[0, 1, 4, 9, 16]
```

Обратите внимание на вызов функции `list` вокруг `map`'а, потому что `map` по умолчанию возвращает `map object` (некий итерабельный объект)

.

То же самое можно сделать и без функции `map`, но более длинно:

```
squared_list = []

for number in range(5):
    squared_list.append(squarify(number))

print(squared_list)
```

```
[0, 1, 4, 9, 16]
```

Ещё одна функция, которая часто используется в контексте функционального программирования, это функция `filter`. Функция `filter` позволяет фильтровать по какому-то предикату итерабельный объект. Она принимает на вход функцию-условие и сам итерабельный объект.

```
def is_positive(a):
    return a > 0


list(filter(is_positive, range(-2, 3)))
```

```
[1, 2]
```

Заметим, что несмотря на то, что `map` и `filter` очень мощны, не стоит злоупотреблять ими, т.к. это ухудшает читаемость кода.

Если мы хотим передать в `map` небольшую функцию, которая нам больше не понадобится, можно использовать анонимные функции (или `lambda`-функции). `Lambda` позволяет вам определить функцию in place, то есть без литерала `def`. Сделаем то же самое, что и в предыдущем примере, с помощью `lambda`:

```
list(map(lambda x: x ** 2, range(5)))
```

```
[0, 1, 4, 9, 16]
```

Лямбда-функция --- это как обычная функция, но без имени:

```
type(lambda x: x ** 2)
```

```
function
```

`Lambda` можно применять с `filter`:

```
list(filter(lambda x: x > 0, range(-2, 3)))
```

```
[1, 2]
```

Упражнение: написать функцию, которая превращает список чисел в список строк.

```
def stringify_list(num_list):
    return list(map(str, num_list))


stringify_list(range(10))
```

```
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

Модуль `functools` позволяет использовать функциональные особенности Python-а ещё лучше. Например, в `functools` в последних версиях языка принесли функцию `reduce`, которая позволяет сжимать данные, применяя последовательно функцию и запоминая результат:

```
from functools import reduce


def multiply(a, b):
    return a * b


reduce(multiply, [1, 2, 3, 4, 5])
# reduce умножает 1 на 2, затем результат этого умножения на 3 и т.д.
```

```
120
```

То же самое можно сделать с помощью анонимной функции:

```
reduce(lambda x, y: x * y, range(1, 5))
```

```
24
```

Метод `partial` из `functools` который позволяет немного модифицировать поведение функций, а именно задать функцию с частью параметров исходной функции, а остальные параметры заменить на некоторые дефолтные значения. Например:

```
from functools import partial


def greeter(person, greeting):
    return '{}, {}!'.format(greeting, person)


hier = partial(greeter, greeting='Hi')
helloer = partial(greeter, greeting='Hello')


print(hier('brother'))
print(helloer('sir'))
```

```
Hi, brother!
Hello, sir!
```

До этого момента мы с вами определяли списки стандартным способом, однако в питоне существует более красивая и лаконичная конструкция для создания списков и других коллекций. Раньше мы делали:

```
square_list = []
for number in range(10):
    square_list.append(number ** 2)

print(square_list)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Лучше использовать списочные выражения (list comprehensions), то есть писать цикл прямо в квадратных скобках:

```
square_list = [number ** 2 for number in range(10)]
print(square_list)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Со списочными выражениями код работает немного быстрее.

Точно так же можно написать списочное выражение с некоторым условием:

```
even_list = [num for num in range(10) if num % 2 == 0]

print(even_list)
```

```
[0, 2, 4, 6, 8]
```

С помощью list comprehensions можно определять словари таким образом:

```
square_map = {number: number ** 2 for number in range(5)}

print(square_map)
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Если применять list comprehensions с фигурными скобками, но без двоеточий, мы получим `set`:

```
reminders_set = {num % 10 for num in range(100)}

print(reminders_set)
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Списочные выражения позволяют вам делать вложенные списки for и другие сложные выражения. Тем не менее, делать это не рекомендуется, т.к. это снижает читаемость кода.

Без скобок списочное выражение возвращает генератор --- объект, по которому можно итерироваться (подробнее про генераторы будет рассказано позже).

```
print(type(number ** 2 for number in range(5)))
```

```
<class 'generator'>
```

Ещё одна важная функция --- функция `zip` --- позволяет вам склеить два итерабельных объекта. В следующем примере мы по порядку соединяем объекты из `numList` и `squaredList` в кортежи:

```
num_list = range(7)
squared_list = [x ** 2 for x in num_list]

list(zip(num_list, squared_list))
```

```
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36)]
```

2.2.4. Декораторы

Декоратор --- это функция, которая принимает функцию и возвращает функцию. И ничего более. Например, простейший декоратор принимает функци и возвращает её же:

```
def decorator(func):
    return func


@decorator # синтаксис декоратора
def decorated():
    print('Hello!')
```

Выражение с @ --- всего лишь синтаксический сахар. Мы можем написать то же самое без него:

```
decorated = decorator(decorated)
```

Чуть более сложный декоратор, который меняет функцию на другую:

```
def decorator(func):
    def new_func():
        pass
    return new_func


@decorator
def decorated():
    print('Hello!')


decorated()

print(decorated.__name__)
```

```
new_func
```

Пример: написать декоратор, который записывает в лог результат декорируемой функции. В этом примере с помощью декоратора `logger` мы подменяем декорируемую функцию функцией `wrapped`. Эта функция принимает на вход тот же `num_list` и возвращает тот же результат, что и исходная функция, но кроме этого записывает результат в лог-файл.

```
def logger(func):
    def wrapped(num_list):
        result = func(num_list)
        with open('log.txt', 'w') as f:
            f.write(str(result))

        return result
    return wrapped


@logger
def summator(num_list):
    return sum(num_list)


print('Summator: {}'.format(summator([1, 2, 3, 4])))
```

```
Summator: 10
```

Можно переписать декоратор так, чтобы он мог применяться не только к функциям, которые принимают `num_list`, а к функциям, которые принимают любое количество аргументов:

```
In [5]:
def logger(func):
    def wrapped(*args, **kwargs):
        result = func(*args, **kwargs)
        with open('log.txt', 'w') as f:
            f.write(str(result))

        return result
    return wrapped
```

Из-за того, что с помощью декоратора мы подменили функцию, её имя поменялось.

```
print(summator.__name__)
```

```
wrapped
```

Этот факт иногда мешает при отладке. Чтобы такого не происходило, можно использовать декоратор `wraps` из модуля `functools`. Он подменяет определённые аргументы,

docstring-и и названия так, что функция не меняется:

```
In [5]:
import functools


def logger(func):
    @functools.wraps(func)
    def wrapped(*args, **kwargs):
        result = func(*args, **kwargs)
        with open('log.txt', 'w') as f:
            f.write(str(result))

        return result
    return wrapped

@logger
def summator(num_list):
    return sum(num_list)

print(summator.__name__)
```

```
wrapped
```

Более сложная задача: написать декоратор с параметром, который записывает лог в указанный файл. Для этого `logger` должен принимать имя файла и возвращать декоратор, который принимает функцию и подменяет её функцией `wrapped`, как мы делали до этого. Всё просто:

```
def logger(filename):
    def decorator(func):
        def wrapped(*args, **kwargs):
            result = func(*args, **kwargs)
            with open(filename, 'w') as f:
                f.write(str(result))
            return result
        return wrapped
    return decorator


@logger('new_log.txt')
def summator(num_list):
    return sum(num_list)

# без синтаксического сахара:
# summator = logger('log.txt')(summator)


summator([1, 2, 3, 4, 5, 6])

with open('new_log.txt', 'r') as f:
    print(f.read())
```

```
21
```

Посмотрим, что будет, если применить сразу несколько декораторов:

```
def first_decorator(func):
    def wrapped():
        print('Inside first_decorator product')
        return func()
    return wrapped


def second_decorator(func):
    def wrapped():
        print('Inside second_decorator product')
        return func()
    return wrapped
```

```
@first_decorator
@second_decorator
def decorated():
    print('Finally called...')

# то же самое, но без синтаксического сахара:
# decorated = first_decorator(second_decorator(decorated))


decorated()
```

```
Inside first_decorator product
Inside second_decorator product
Finally called...
```

Видим, что сначала вызвался сначала первый декоратор, потом второй. Разберём это подробнее. Функция `second_decorator` возвращает новую функцию `wrapped`, таким образом, функция подменяется на `wrapped` внутри `second_decorator`-а. После этого вызывается `first_decorator`, который принимает функцию полученную из `second_decorator`-а `wrapped` и возвращает ещё одну функцию `wrapped` заменяя `decorated` на неё. Таким образом, итоговая функция `decorated` — это функция `wrapped` из `first_decorator` вызывающая функцию из `second_decorator`-а.

Ещё один пример на применение декораторов. Обратите внимание, что сначала теги идут в том же порядке, что и декораторы, а затем в обратном. Это происходит потому, что декораторы вызываются один внутри другого.

```
def bold(func):
    def wrapped():
        return "<b>" + func() + "</b>"
    return wrapped


def italic(func):
    def wrapped():
        return "<i>" + func() + "</i>"
    return wrapped


@bold
@italic
def hello():
    return "hello world"


# hello = bold(italic(hello))


print(hello())
```

```
<b><i>hello world</i></b>
```

2.2.5. Генераторы

Простейший генератор --- это функция в которой есть оператор `yield`. Этот оператор возвращает результат, но не прерывает функцию. Пример:

```
def even_range(start, end):
    current = start
    while current < end:
        yield current
        current += 2

for number in even_range(0, 10):
    print(number)
```

```
0
2
4
6
8
```

Генератор `even_range` прибавляет к числу двойку и делает с ним операцию `yield`, пока `current < end`. Каждый раз, когда выполняется `yield`, возвращается значение

current, и каждый раз, когда мы просим следующий элемент, выполнение функции возвращается к последнему моменту, после чего она продолжает исполняться. Чтобы посмотреть, как это происходит на самом деле, можно воспользоваться функцией next, которая действительно применяется каждый раз при итерации.

```
ranger = even_range(0, 4)

next(ranger)
```

```
0
```

```
next(ranger)
```

```
2
```

```
next(ranger)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-6-9065b0f81b55> in <module>()
----> 1 next(ranger)

StopIteration:
```

Мы получили ошибку, т.к. у генератора больше нет значений, которые он может выдать.

Можем проверить, что функция действительно прерывается каждый раз после выполнения yield:

```
def list_generator(list_obj):
    for item in list_obj:
        yield item
        print('After yielding {}'.format(item))


generator = list_generator([1, 2])

next(generator)
```

```
1
```

```
next(generator)
```

```
After yielding 1
2
```

```
next(generator)
```

```
After yielding 2
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-10-1d0a8ea12077> in <module>()
----> 1 next(generator)

StopIteration:
```

Когда применяются генераторы? Они нужны, например, тогда, когда мы хотим итерироваться по большому количеству значений, но не хотим загружать ими память. Именно поэтому стандартная функция `range()` реализована как генератор (впрочем, так было не всегда).

Приведём классический пример про числа Фибоначчи:

```
def fibonacci(number):
    a = b = 1
    for _ in range(number):
        yield a
        a, b = b, a + b

for num in fibonacci(10):
    print(num)
```

```
1
1
2
3
5
8
13
21
34
55
```

С таким генератором нам не нужно помнить много чисел Фибоначчи, которые быстро растут --- достаточно помнить два последних числа.

Еще одна важная особенность генераторов --- это возможность передавать генератору какие-то значения. Эта особенность активно используется в асинхронном программировании, о котором будет речь позднее. Пока определим генератор `accumulator`, который хранит общее количество данных и в бесконечном цикле получает с помощью оператора `yield` значение. На первой итерации генератор возвращает начально значение `total`. После этого мы можем послать данные в генератор с помощью метода генератора `send`. Поскольку генератор остановил исполнение в некоторой точке, мы можем послать в эту точку значение, которое запишется в `value`. Далее, если `value` не было передано, генератор выходит из цикла, иначе прибавляем его к `total`.

```
def accumulator():
    total = 0
    while True:
        value = yield total
        print('Got: {}'.format(value))

        if not value: break
        total += value

generator = accumulator()

next(generator)
```

```
0
```

```
print('Accumulated: {}'.format(generator.send(1)))
```

```
Got: 1
Accumulated: 1
```

```
print('Accumulated: {}'.format(generator.send(1)))
```

```
Got: 1
Accumulated: 2
```

```
print('Accumulated: {}'.format(generator.send(1)))
```

```
Got: 1
Accumulated: 3
```

```
next(generator)
```

```
Got: None
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-18-1d0a8ea12077> in <module>()
----> 1 next(generator)

StopIteration:
```