



# Составление алгоритмов. Применение машинной логики к задачам поиска данных. Оценка времени работы алгоритмов, эффективность кода

Турашова Анна Николаевна  
Преподаватель  
[anna1turashova@gmail.com](mailto:anna1turashova@gmail.com)  
Telegram: @anna1tur



# Проверка домашнего задания



# Задачи

1. Напишите программу, которая получает от пользователя число и выводит обратный отсчет.
2. Напишите программу, которая загадывает число от 1 до 10. Выполнять программу до тех пор, пока пользователь не угадает число.
3. Напишите программу, которая подсчитывает количество цифр до специального символа. На входе строка любой длины состоящая из цифр и спец. символа. Подсчитать количество цифр до специального символа '\$', который остановит подсчет.

'3423\$323' -> 4

'\$' -> 0

'34319\$' -> 5

'8\$' -> 1

4. Написать программу, которая на входе получает начальный и конечный символ. Отобразить строку начиная с начальная символа и завершая конечным символом согласно таблице ascii.

Пример: Начать с символа 'e' и завершить символом 'j'

Результат: 'efghij'



# Задачи

1. Напишите программу для вычисления суммы и среднего n целых чисел. Числа вводит пользователем до тех пор, пока не введет 0, чтобы закончить.

2. Напишите программу, которая получает на входе число и проверяет, является ли оно простым.

Простое число — натуральное (целое положительное число), имеющее ровно два различных натуральных делителя — единицу и самого себя

Алгоритм:

Получить x натуральное число

Установить делитель = 2

Пока x > делителя:

Если нет остатка от деления x на делитель:

Число не простое, остановить цикл

Увеличить делить на 1



# Задачи

1. Напишите программу, которая печатает все числа от 0 до 6, кроме 3 и 6.
2. Напишите программу Python, которая выполняет перебор целых чисел от 1 до 50. Для кратных трем выведите «Fizz» вместо числа, а для кратных пяти выведите «Buzz». Для чисел, кратных трем и пяти, выведите «FizzBuzz».

Пример:

```
fizzbuzz # кратно 3 и 5
```

```
1
```

```
2
```

```
fizz # кратно 3
```

```
4
```

```
buzz # кратно 5
```

```
fizz # кратно 3
```

```
7
```

```
8
```

```
fizz
```

```
buzz
```

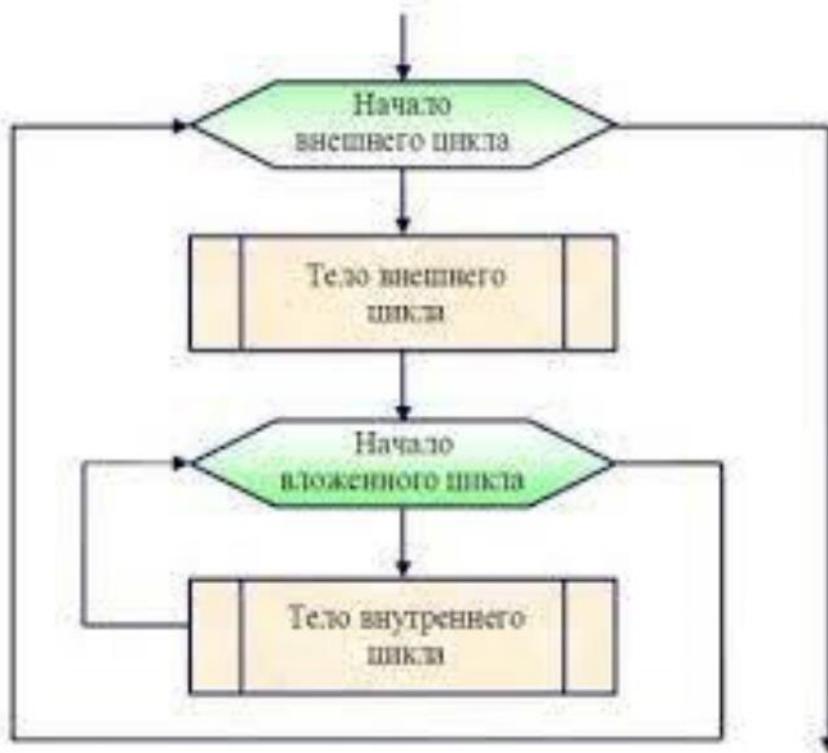
```
11
```



# Вложенные циклы, сложность алгоритма

# Вложенные циклы

- Цикл называется **вложенным**, если он размещается внутри другого **цикла**
- На первом проходе, **внешний цикл** вызывает внутренний, который исполняется до своего завершения, после чего управление передается в тело внешнего цикла
- И так до тех пор, пока не завершится **внешний цикл**.





# Примеры задач

- Написать программу, которая выводит таблицу умножения чисел от 1 до N в следующем виде:

$1 \times 1 = 1 \quad 1 \times 2 = 2 \quad 1 \times 3 = 3 \dots$

$2 \times 1 = 2 \quad 2 \times 2 = 4 \quad 2 \times 3 = 6 \dots$

```
n = int(input())
for i in range(1, n + 1):  # перебираем первый множитель i от 1 до n
    for j in range(1, n + 1):  # перебираем второй множитель j от 1 до n
        print(f"{j}x{i}={(i * j):2}", end="")
    print()
```

- Найти количество простых чисел на отрезке от числа a до числа b. a, b < 10000



# Анализ алгоритмов

## Цели

- Оценить качество алгоритмов с помощью количественных критериев
- Сравнить различные алгоритмы для решения заданной алгоритмической задачи



# Алгоритмическая задача

- входные данные
- необходимый результат.

Алгоритм называют правильным, если на любом допустимом входе он заканчивает работу и выдает результат, удовлетворяющий требованиям задачи.



# Сложность алгоритма

- **Временная сложность** алгоритма — количество элементарных шагов, которые он выполняет.
- **Объемная сложность** алгоритма — количество оперативной памяти, необходимой для его выполнения.

В обоих случаях сложность зависит от размеров входных данных: 100 элементов будет обработано быстрее, чем 1000

Точное время работы алгоритма зависит от

- Процессора
- Типа данных
- Языка программирования
- Множества других параметров

Для алгоритма важна лишь асимптотическая сложность, т. е. сложность при увеличении размера входных данных

# Порядок роста функции

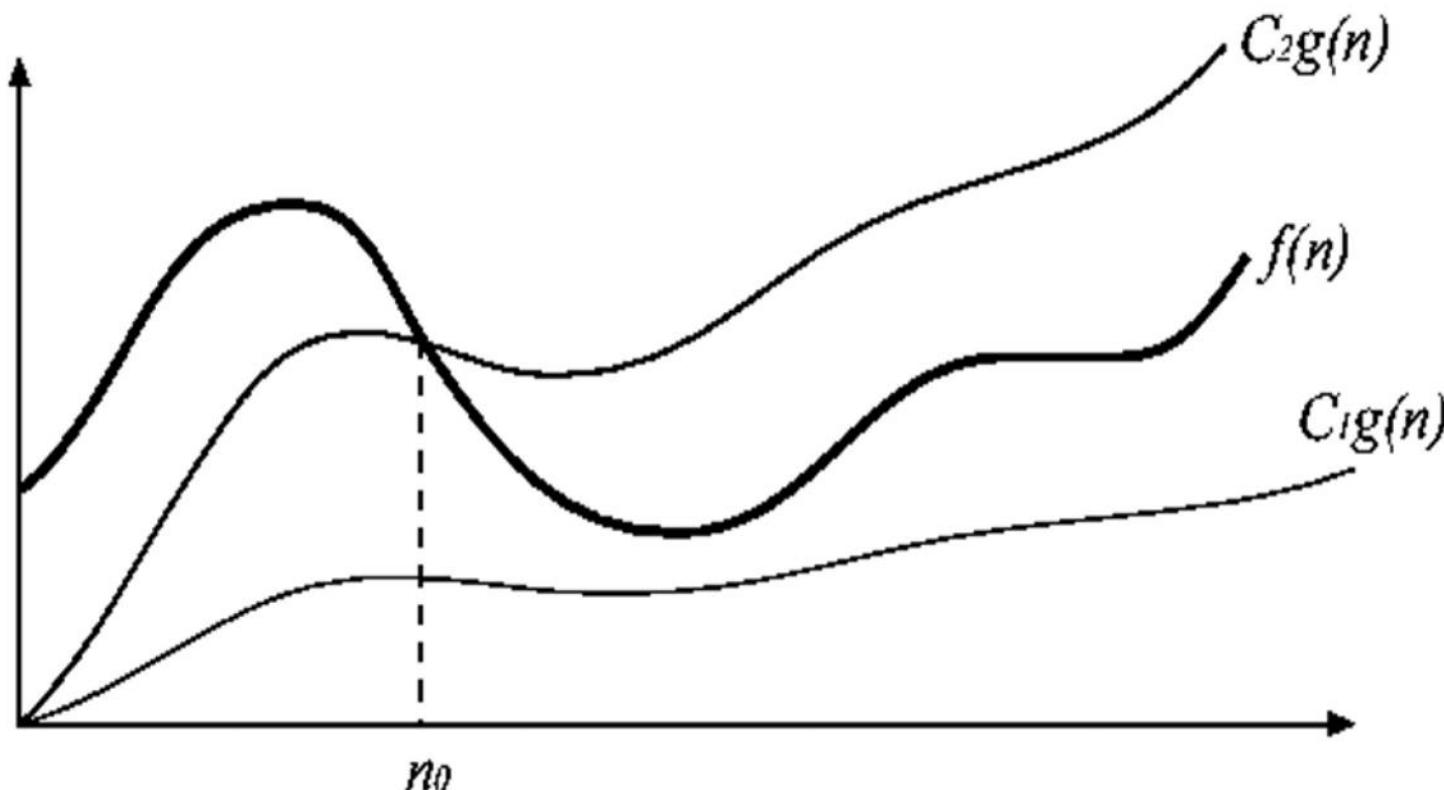


Рис. 1.1.  $f(n) = \Theta(g(n))$



# Временная сложность

- **В лучшем случае** – минимальное число действий
- **В худшем случае** – максимальное число действий
- **В среднем** – среднее число действий



# Примеры вычисления сложности

1. Проверка числа на простоту
2. Найти все делители на интервале



# Классы сложности алгоритмов

- $O(1)$  – константные
- $O(\log n)$  – логарифмические
- $O(n)$  – линейные
- $O(n^2)$  – квадратичные
- $O(n^k)$  – полиномиальные
- $O(2^n)$  – экспоненциальные



# Эффективность алгоритмов

размер сложность	10	20	30	40	50	60
$n$	0,00001 сек.	0,00002 сек.	0,00003 сек.	0,00004 сек.	0,00005 сек.	0,00005 сек.
$n^2$	0,0001 сек.	0,0004 сек.	0,0009 сек.	0,0016 сек.	0,0025 сек.	0,0036 сек.
$n^3$	0,001 сек.	0,008 сек.	0,027 сек.	0,064 сек.	0,125 сек.	0,216 сек.
$n^5$	0,1 сек.	3,2 сек.	24,3 сек.	1,7 минут	5,2 минут	13 минут
$2^n$	0,0001 сек.	1 сек.	17,9 минут	12,7 дней	35,7 веков	366 веков
$3^n$	0,059 сек.	58 минут	6,5 лет	3855 веков	$2 \times 10^8$ веков	$1,3 \times 10^{13}$ веков



# Библиотека time

```
1 import time
2
3 start_time = time.time()
4
5 for i in range(55296):
6     print(f'{i} = {chr(i)}')
7
8 print(time.time() - start_time, 'секунд')
```



# Списки, продолжение



## Обобщение свойств встроенных коллекций в сводной таблице:

Тип коллекции	Мутабельность	Индексированность	Уникальность	Как создаём
Список (list)	+	+	-	<code>[], list()</code>
Кортеж (tuple)	-	+	-	<code>(), tuple()</code>
Строка (string)	-	+	-	<code>''</code> <code>'''</code>
Множество (set)	+	-	+	<code>{}</code> <code>set()</code>
Неизменное множество (frozenset)	-	-	+	<code>frozenset()</code>
Словарь (dict)	+ элементы - ключи + значения	-	+ элементы + ключи - значения	<code>{key: value,}</code> <code>dict()</code>



# Коллекции

## Списки:

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

```
print(range_list[:] is range_list)
```

```
False
```

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

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

```
for collection in collections:
```

```
    # используем функцию format для форматирования строк
```

```
    print('Learning {}...'.format(collection))
```

```
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.



# Коллекции

## Списки:

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']
```



# Коллекции

## Списки:

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]
```



# Коллекции

## Списки:

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', 'Академия АЙТИ']  
print(', '.join(tag_list))
```

python, course, Академия АЙТИ



# Коллекции

## Списки:

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]



# Коллекции

## Списки:

```
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]
```



# Коллекции

## Списки:

### Коллекция:

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



# Коллекции

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  
--> 1 immutables[0] = float  
TypeError: 'tuple' object does not support item assignment
```

```
Traceback (most recent call last) in ()
```

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], [])
```



# Коллекции

Важная особенность кортежей — к ним применяется функция `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
```



# Коллекции

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 случайнym (чтобы было интереснее) количеством элементов.

```
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



# Коллекции

## 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

```
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]
```



# Коллекции

## Пример программы:

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
```



# Коллекции

## Пример программы:

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



# Словари



# Коллекции

## 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']
```



# Коллекции

## Словари:

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

KeyError

—> 1 print(collections\_map['irresistible'])

KeyError: 'irresistible'

Traceback (most recent call last) in ()

```
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



# Коллекции

## Словари:

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 11 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"}
```



# Коллекции

## Словари:

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'}
```



# Коллекции

## Словари:

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'))  
print(unknown_dict)
```

```
default  
{'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
```



# Коллекции

## Словари:

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']
```



# Коллекции

## Словари:

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)
```



# Коллекции

## 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!



# Коллекции

## Словари. Пример программы:

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 помощью метода split(), который разобъёт 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
    else:
        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...}
```



# Коллекции

## Словари. Пример программы:

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))
```



# Коллекции

## Словари. Пример программы:

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))
```



# Коллекции

## Домашняя работа:

Задача 1:

Испорченный гугл: Создать программу, которая будет обрабатывать русский текст, заменяя в нём символы русского языка на схожие по написанию символы английского языка(а,с,х,о,р,е), при этом сохраняя регистр.

Задача 2:

Буквоежка level-up: Отсортировать по алфавиту три(можно больше) отдельных строки убирая общие(между всеми тремя строками) дубли

(итог выводить для каждой отдельной строки)

Задача 3:

Склад: Создать список словарей для инвентаризации чего-либо

Задача 4:

Цензор: создать программу, которая будет цензуривать вводимый текст по заданному шаблону  
(не обязательно цензурировать мат! Шаблон может быть любым!(например слова-паразиты))



Входит в ГК Аплана



АКАДЕМИЯ АЙТИ

Основана в 1995 г.

E-learning  
и очное  
обучение



Ежегодные награды  
Microsoft,  
Huawei, Cisco и  
другие

Направления обучения:

Информационные технологии

Информационная безопасность

ИТ-менеджмент и управление проектами

Разработка и тестирование ПО

Гос. и муниципальное управление

#### Филиалы:

Санкт-Петербург, Казань, Уфа, Челябинск,  
Хабаровск, Красноярск, Тюмень, Нижний  
Новгород, Краснодар, Волгоград, Ростов-на-Дону

Головной офис  
в Москве

Ресурсы более 400  
высококлассных  
экспертов и  
преподавателей

Разработка  
программного  
обеспечения и  
информационных  
систем

Программы по  
импортозамещению

Сеть региональных учебных центров  
по всей России

#### Крупные заказчики



100+  
сотрудников



АКАДЕМИЯ АЙТИ

# Спасибо за внимание!

Центральный офис:

Москва, Варшавское шоссе 47, корп. 4, 7 этаж

Тел: +7 (495) 150-96-00

[academy@it.ru](mailto:academy@it.ru)

[academyit.ru](http://academyit.ru)