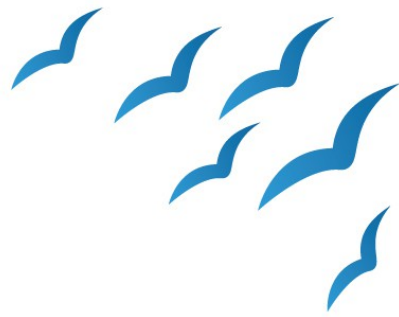


# Коллекции





# Оглавление

1. Список (`list`)
2. Словарь (`dict`)
3. Множество (`set`)
4. Замороженное множество (`frozenset`)
5. Кортеж (`tuple`)



| Тип коллекции                            | Изменяемость                        | Индексированность | Уникальность                        | Как создаём  |
|--|-------------------------------------|-------------------|-------------------------------------|--|
| Список<br>(list)                         | +                                   | +                 | -                                   | <code>[]</code><br><code>list()</code>                               |
| Кортеж<br>(tuple)                        | -                                   | +                 | -                                   | <code>()</code> ,<br><code>tuple()</code>                            |
| Строка<br>(string)                       | -                                   | +                 | -                                   | <code>" "</code><br><code>" "</code>                                 |
| Множество<br>(set)                       | +                                   | -                 | +                                   | <code>{elm1, elm2}</code><br><code>set()</code>                      |
| Неизменяемое<br>множество<br>(frozenset) | -                                   | -                 | +                                   | <code>frozenset()</code>   |
| Словарь<br>(dict)                        | + элементы<br>- ключи<br>+ значения | -                 | + элементы<br>+ ключи<br>- значения | <code>{}</code><br><code>{key: value,}</code><br><code>dict()</code> |



## Список(List)

- Список — это упорядоченный набор элементов, перечисленных через запятую, заключённый в квадратные скобки
- Элементы списка могут быть разных типов, в отличие от элементов массива (`array`), но, как правило, используются списки из элементов одного типа
- Список может содержать одинаковые элементы, в отличие от множества (`set`)
- Список можно изменить после создания, в отличие от кортежа (`tuple`)
- Список может содержать другие списки



## Список (mutable)

Создать список можно двумя способами:

Вызывать функцию `list()`

```
lst = list()
```

Использовать квадратные скобки

```
lst = [] → Задали пустой список
```

Пример:

```
lst = list([1, 4, 5])
```

```
lst = list("hello")
```

```
lst = [1, 4, 5]
```



## Элементы списка разных типов

Пример:

```
>>> lst = [10, True, [1,2], "#ffffff"]
```

```
>>> type(lst)
```

```
<class 'list'>
```



## Список – изменяемый тип данных

Так как список - изменяемый тип данных, то мы можем заменить определённый элемент в нём на другой.

```
>>> mass = [4, 3, 2, 1]
```

```
>>> mass[0] = 8
```

```
>>> [8, 3, 2, 1]
```



## Создание копии(клона) списка

```
>>> a = [1, 3, 5, 7]
```

```
>>> b = a[:]
```

Вопрос!

Что за оператор **[:]** ?

Второй способ:

```
>>> a = [1, 3, 5, 7]
```

```
>>> b = list(a)
```





## Присвоение списка

В случае, если вы выполните простое присвоение списков друг другу, то переменной **b** будет присвоена ссылка на тот же элемент данных в памяти, на который ссылается **a**

```
>>> a = [1, 3, 5, 7]
```

```
>>> b = a
```

```
>>> b[0] = 10
```

```
>>> print(a)
```

```
[10, 3, 5, 7]
```



# Сложение массивов

При сложении происходит объединение множеств массива

```
>>> a = [1, 2]
```

```
>>> b = [3, 4]
```

```
>>> c = a + b
```

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



## Размножение списка

Мультипликация списка происходит при умножение его на число.

```
>>> a = [1, 2]
```

```
>>> b*2
```

```
[1, 2, 1, 2]
```



## Нахождение значения в списке in , not in

```
>>> a = [1,2]
```

```
>>> b = 2
```

```
>>> b in a
```

```
True
```

```
>>> 19 not in a
```

```
True
```

```
>>> 1 not in a
```

```
False
```



## Получить размер списка - len()

```
>>> mass = [1, 2, 3]
```

```
>>> len(mass)
```

```
3
```

# предпо<sup>л</sup>ожение

Догадка, предварительная мысль.



list

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

# Python List Methods



WTMatter





## Метод `append(x)`

Добавление элемента в список осуществляется с помощью метода `append()`

```
>>> a = []  
>>> a.append(3)  
>>> a.append("hello")  
>>> print(a)  
[3, 'hello']
```





## Метод `clear()`

Метод `clear()` удаляет все элементы из списка.

```
>>> a = [1, 2, 3, 4, 5]
```

```
>>> print(a)
```

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

```
>>> a.clear()
```

```
>>> print(a)
```

```
[]
```



## Метод `copy()`

Возвращает копию списка. Эквивалентно `a[:]`

```
>>> a = [1, 7, 9]
```

```
>>> b = a.copy()
```

```
>>> print(b)
```



## Метод count (x)

Возвращает количество вхождений элемента **x** в список.

```
>>> a=[1, 2, 2, 3, 3]
```

```
>>> print(a.count(2))
```

```
2
```



## Метод `extend(L)`

Расширяет существующий список за счет добавления всех элементов из списка `L`.

Эквивалентно команде `a[len(a) :] = L`

```
>>> a = [1, 2]
>>> b = [3, 4]
>>> a.extend(b)
>>> print(a)
[1, 2, 3, 4]
```



## Метод `index(x[, start[, end]])`

Возвращает индекс элемента.

```
>>> a = [1, 2, 3, 4, 5]
```

```
>>> a.index(4)
```

```
3
```



## Метод `insert(i, x)`

Вставить элемент `x` в позицию `i`. Первый аргумент – индекс элемента после которого будет вставлен элемент `x`.

```
>>> a = [1, 2]
>>> a.insert(0, 5)
>>> print(a)
[5, 1, 2]
>>> a.insert(len(a), 9)
>>> print(a)
[5, 1, 2, 9]
```



## Метод `list.pop([i])`

Удаляет элемент из позиции `i` и возвращает его. Если использовать метод без аргумента, то будет удален последний элемент из списка.

```
>>> a = [1, 2, 3, 4, 5]
```

```
>>> print(a.pop(2))
```

```
3
```

```
>>> print(a.pop())
```

```
5
```

```
>>> print(a)
```

```
[1, 2, 4]
```



## Метод `remove(x)`

Удаляет первое вхождение элемента `x` из списка.

```
>>> a = [1, 2, 3]
```

```
>>> a.remove(1)
```

```
>>> print(a)
```

```
[2, 3]
```





## Метод `reverse()`

Изменяет порядок расположения элементов в списке на обратный.

```
>>> a = [1, 3, 5, 7]
```

```
>>> a.reverse()
```

```
>>> print(a)
```

```
[7, 5, 3, 1]
```



## Метод `sort()`

Сортирует элементы в списке по возрастанию. Для сортировки в обратном порядке используйте флаг `reverse=True`.  
Дополнительные возможности открывает параметр `key`, за более подробной информацией обратитесь к документации.

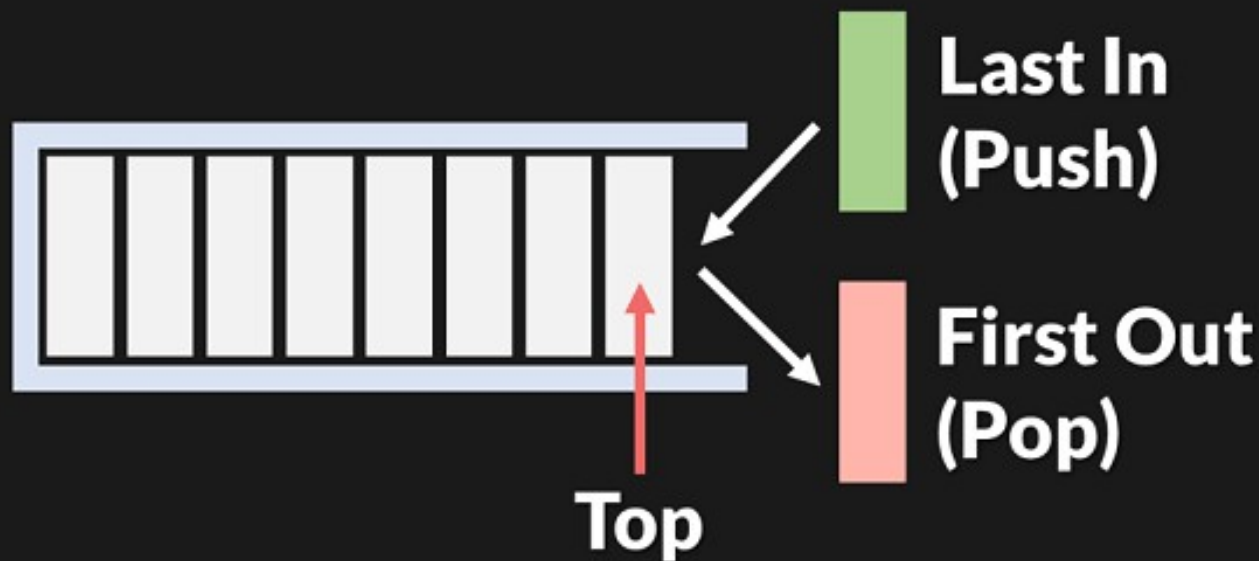
```
>>> a = [1, 4, 2, 8, 1]
>>> a.sort()
>>> print(a)
[1, 1, 2, 4, 8]
```



Как запомнить все эти методы ?



# Python Stack Implementation

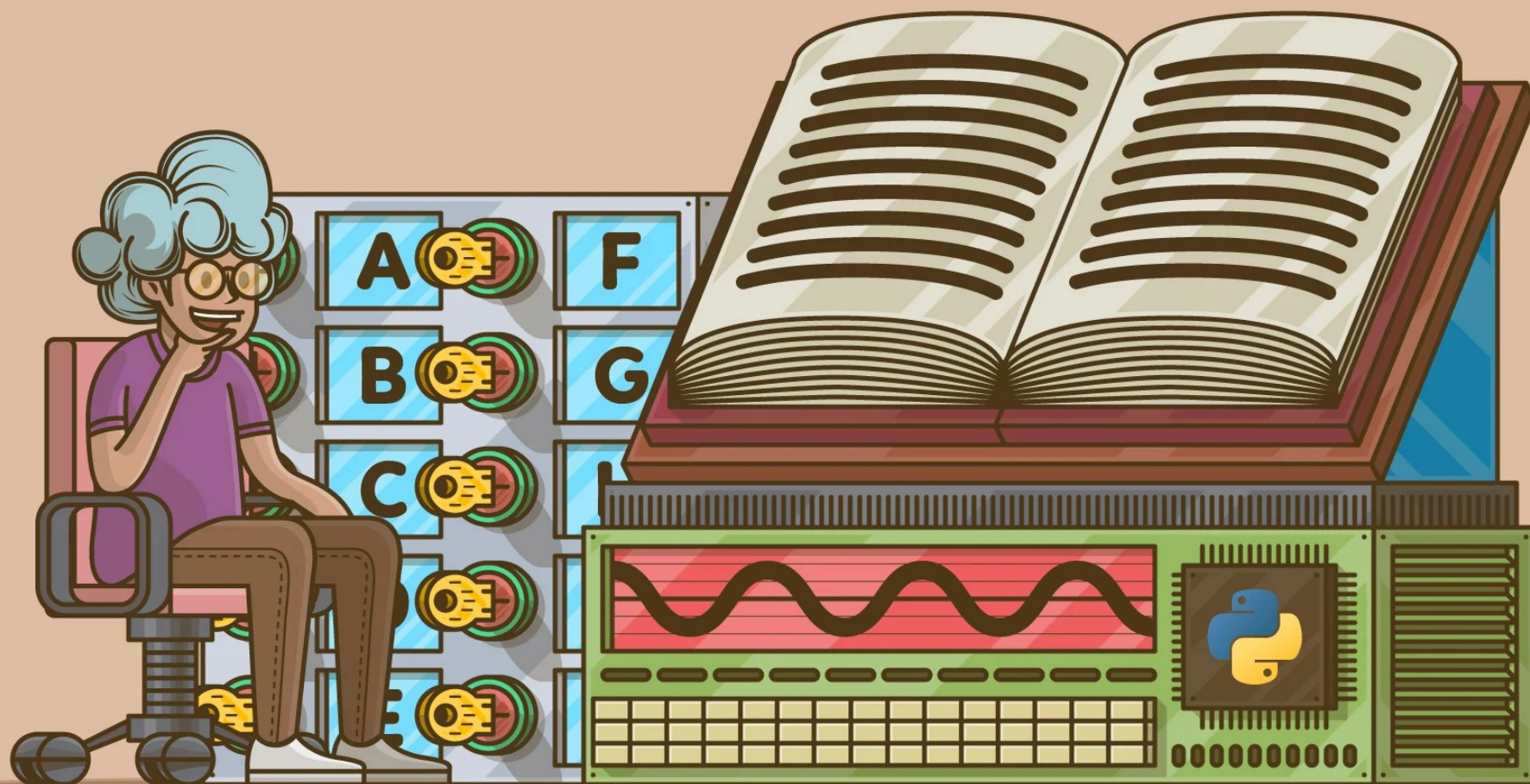


# Срезы(слайсы) в массивах

Срез как и в строках задается тройкой чисел `[start:stop:step]` `start` – индекс с которой нужно начать выборку, `stop` – конечный индекс, `step` – шаг. При этом необходимо помнить, что выборка не включает элемент определяемый индексом `stop`.

```
>>> # Получить копию списка
>>> a[:]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> # Получить первые пять элементов списка
>>> a[0:5]
[0, 1, 2, 3, 4]
>>> # Получить элементы с 3-го по 7-ой
>>> a[2:7]
[2, 3, 4, 5, 6]
>>> # Взять из списка элементы с шагом 2
>>> a[::2]
[0, 2, 4, 6, 8]
>>> # Взять из списка элементы со 2-го по 8-ой с шагом 2
>>> a[1:8:2]
[1, 3, 5, 7]
```

# Словарь



Real Python



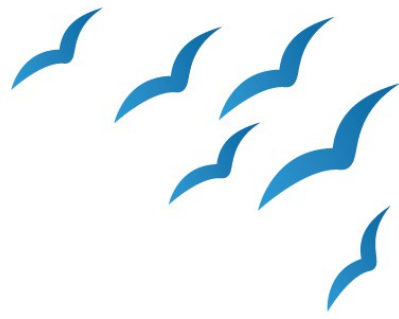
# Определение словаря

```
dict = {k:v}
```

Словарь задается парой **ключ: значение**,

```
dic = {  
    <key>: <value>,  
    <key>: <value>,  
    .  
    .  
    .  
    <key>: <value>  
}
```

## Пример 1:



```
person = {  
    'name': 'Маша',  
    'login': 'masha',  
    'age': 25,  
    'email': 'masha@yandex.ru',  
    'password': 'fhei23jj~'  
}  
  
print (type (person) )  
<class 'dict'>
```

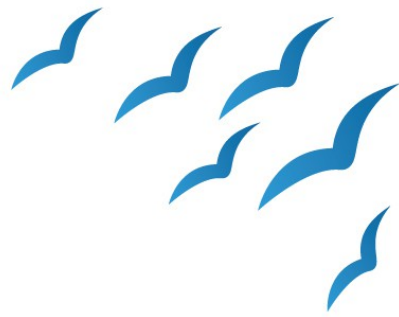




## Пример 2:

#Словарь, где ключи являются  
целыми числами.

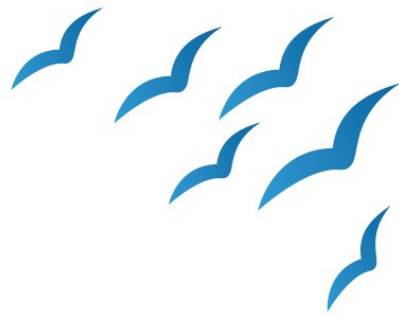
```
dict_sample = {  
    1: 'mango',  
    2: 'coco'  
}
```



## Пример 3:

```
# Hmm... если ключи состоят из  
примитивных типов то могу ли я  
сделать так ?
```

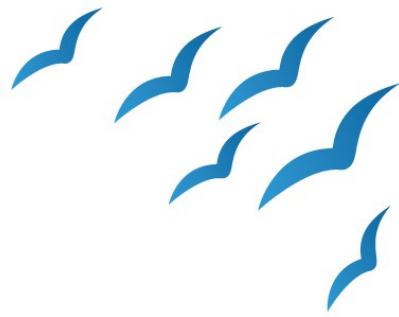
```
dict_sample = {  
    True: 'mango',  
    False: 'coco'  
}
```



## Пример 4:

```
# .. пойдём дальше
```

```
dict_sample = {  
    None: 'mango',  
    None: 'coco'  
}
```



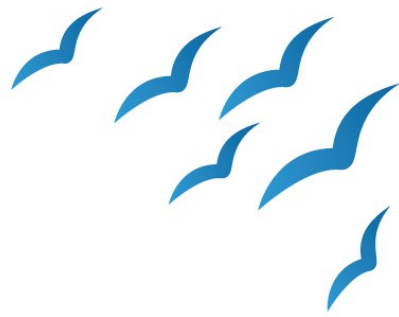
# Другие способы создания

```
{ 'name': 'Маша', 'age': 16 }      # литеральным  
выражением
```

```
person = {}      # динамическое присваение по ключам  
person['name'] = 'Маша'  
person['age'] = 16
```

!!!

```
dict(name='Маша', age=16) # через конструктор dict
```



# Доступ к элементу по ключу

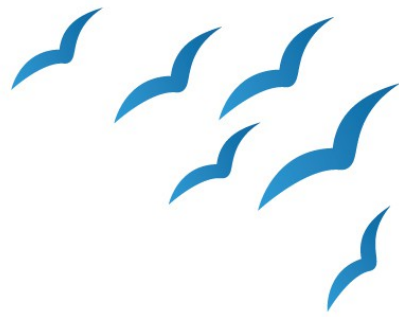
```
dict = { k: v }
```

```
>>> person['name']
```

Маша

# Замена значения

```
>>> person['name'] = 'Даша'
```



# Добавление нового элемента

```
dict = { k: v, k2: v2 }
```

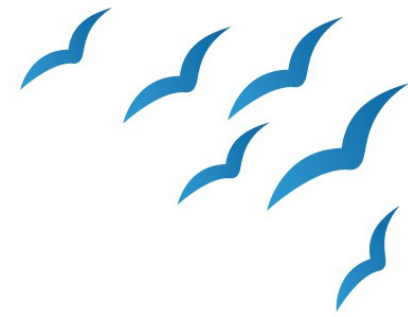
```
>>> person['surname'] = 'Медведьева'
{
    'name': 'Даша',
    'login': 'masha',
    'age': 25, 'email': 'masha@yandex.ru',
    'password': 'fhei23jj~',
    'surname': 'Медведьева'
}
```



# Удаление элемента

dict = { k: v, :  }

```
>>> del person['login']  
{  
    'name': 'Даша',  
    'age': 25,  
    'email': 'masha@yandex.ru',  
    'password': 'fhei23jj~',  
    'surname': 'Медведева'  
}
```



# Проверка на наличие ключа

```
dict = { "A": v }
```

"A"? ↗

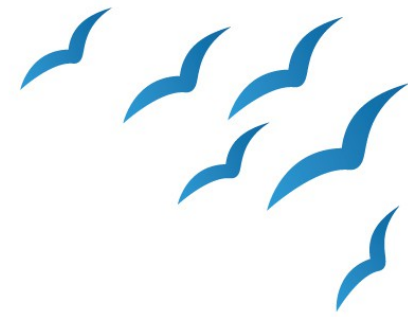
```
>>> 'name' in person
```

```
True
```

```
print('Ключ есть') if ('name' in person)  
else print('Ключа нет')
```



# Длина словаря в Python



dict = { k : v , k2 : v2 }

len = 2 ↗

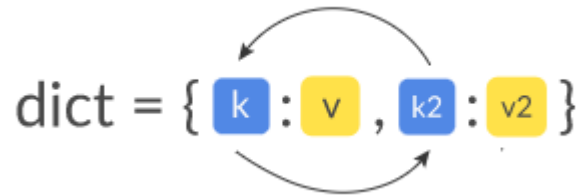
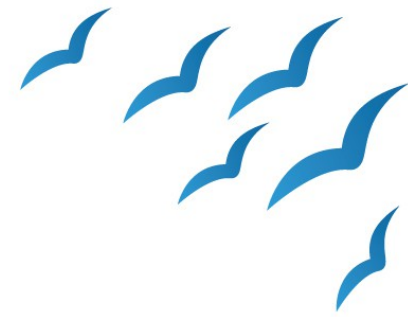
Количество записей мы можем получить, воспользовавшись функцией len()

```
>>> num_of_items = len(person)
```

```
>>> print(num_of_items)
```

```
>>> 5
```

# Сортировка словаря

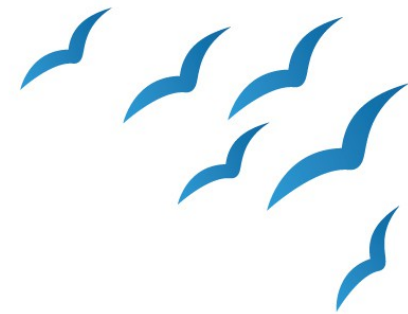


```
statistic_dict = {'b': 10, 'd': 30, 'e': 15,  
'c': 14, 'a': 33}
```

```
for key in sorted(statistic_dict):  
    print(key)
```

```
a  
b  
c  
d  
e
```

# Итерирование словаря



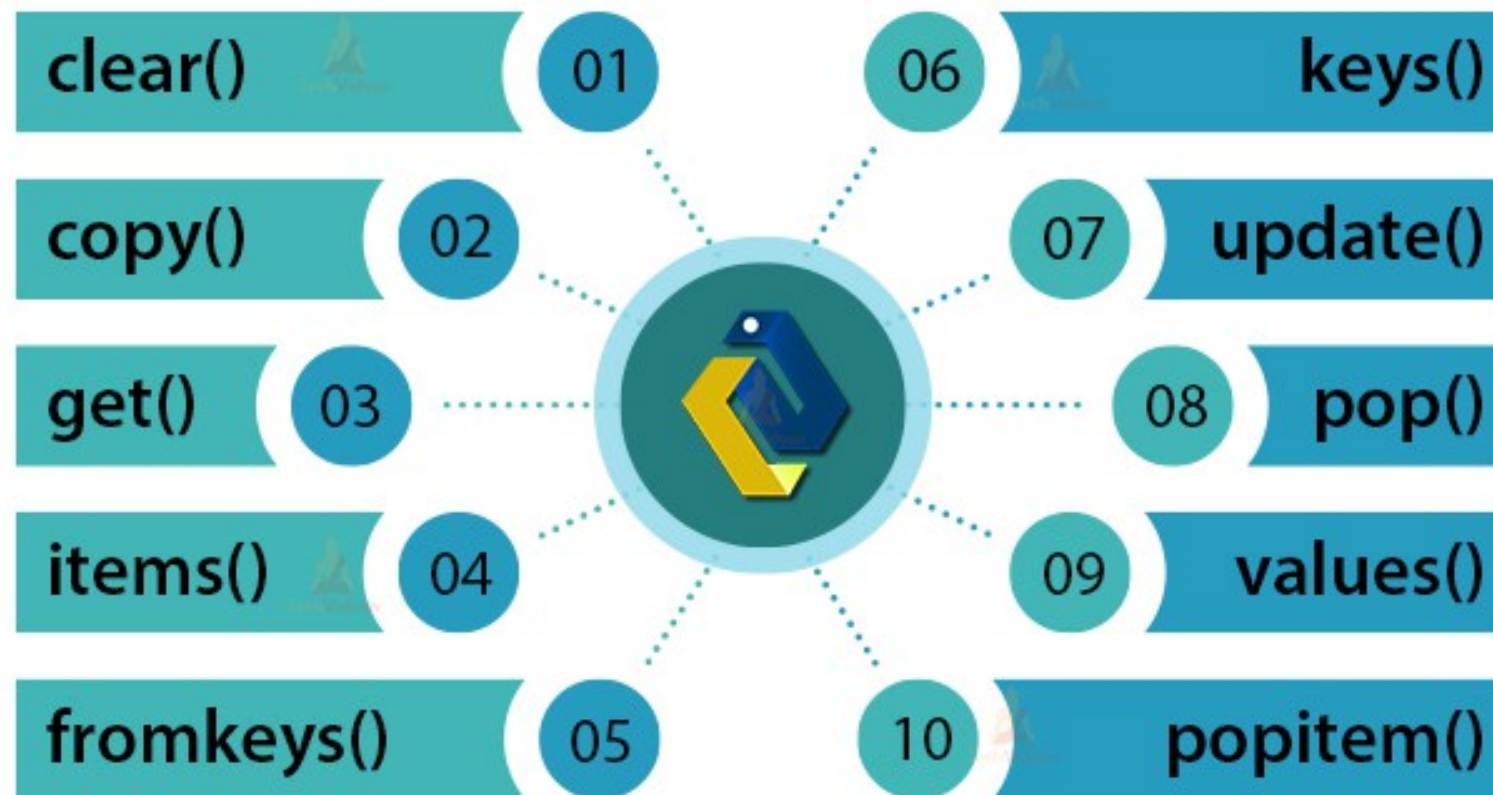
dict = { k : v , k2 : v2 , k3 : v3 }

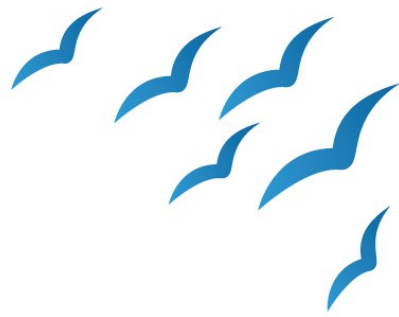


```
statistic_dict = {'b': 10, 'd': 30, 'e': 15,  
'c': 14, 'a': 33}
```

```
for key, val in statistic_dict.items():  
    print(key)  
    print(val)
```

# Python Dictionary Methods

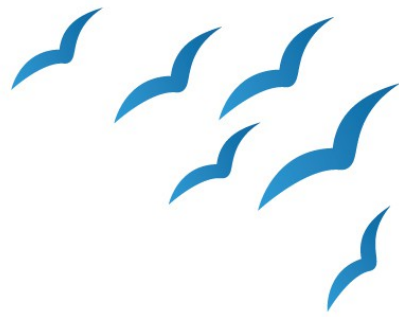




# clear()

Метод производит удаление всех элементов из словаря.

```
>>> x = {'one': 0, 'two': 20, 'three': 3,  
'four': 4}  
>>> x.clear()  
>>> x  
  
# {}
```



# copy()

Метод создает копию словаря.

```
>>> x = {'one': 0, 'two': 20, 'three': 3,  
        'four': 4}
```

```
>>> y = x.copy()
```

```
>>> y
```

```
{'one': 0, 'two': 20, 'three': 3, 'four': 4}
```



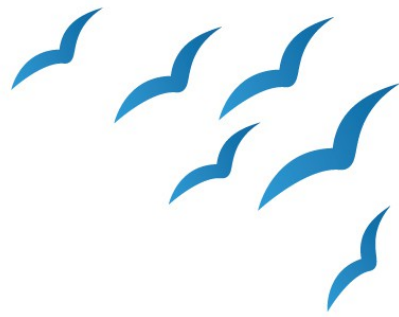
# get(key[, default])

Метод dict.get() возвращает значение для ключа key, если ключ находится в словаре, если ключ отсутствует то вернет значение default.

Если значение default не задано и ключ key не найден, то метод вернет значение None.

Метод dict.get() никогда не вызывает исключение KeyError, как это происходит в операции получения значения словаря по ключу dict[key].

```
>>> x = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> x.get('two', 0)
# 2
>>> x.get('ten', 0)
# 0
```



# items()

Метод `dict.items()` возвращает новый список кортежей вида (key, value), состоящий из элементов словаря.

```
>>> x = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> items = x.items()
>>> items
```

```
dict_items([('one', 1), ('two', 2), ('three', 3),
('four', 4)])
```



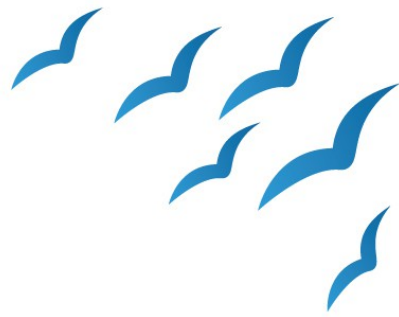


# items()

Метод `dict.items()` возвращает новый список кортежей вида (key, value), состоящий из элементов словаря.

```
>>> x = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> items = x.items()
>>> items
```

```
dict_items([('one', 1), ('two', 2), ('three', 3),
('four', 4)])
```



# fromkeys(iterable[, value])

Метод dict.fromkeys() встроенного класса dict() создает новый словарь с ключами из последовательности iterable и значениями, установленными в value.

```
>>> x = dict.fromkeys(['one', 'two', 'three',  
    'four'])
```

```
>>> x
```

```
{'one': None, 'two': None, 'three': None, 'four':  
None}
```

```
>>> x = dict.fromkeys(['one', 'two', 'three',  
    'four'], 0)
```

```
>>> x
```

```
{'one': 0, 'two': 0, 'three': 0, 'four': 0}
```

# keys()



Метод `dict.keys()` возвращает новый список-представление всех ключей , содержащихся в словаре `dict`.

```
>>> x = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> keys = x.keys()
>>> keys
dict_keys(['one', 'two', 'three', 'four'])
```

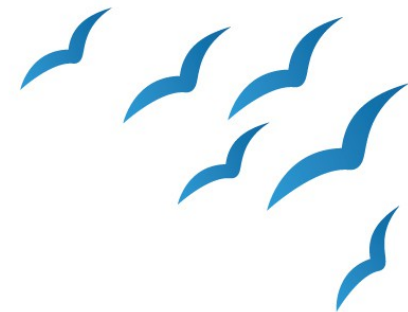
# keys()



Список-представление ключей `dict_keys`, является динамичным объектом. Это значит, что все изменения, такие как удаление или добавление ключей в словаре сразу отражаются на этом представлении.

```
# Производим операции со словарем 'x', а все
# отражается на списке-представлении `keys`
>>> x = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> keys = x.keys()
>>> del x['one']
>>> keys
dict_keys(['two', 'three', 'four'])
>>> x
{'two': 2, 'three': 3, 'four': 4}
```

# update() - объединение словарей



dict1 = { k1: v1 }  
dict2 = { k2: v2 }



```
showcase_1 = { 'Apple': 2.7, 'Grape': 3.5,  
               'Banana': 4.4 }
```

```
showcase_2 = { 'Orange': 1.9, 'Coconut': 10 }  
showcase_1.update(showcase_2)
```

```
print(showcase_1)
```

```
> { 'Apple': 2.7, 'Grape': 3.5, 'Banana': 4.4,  
    'Orange': 1.9, 'Coconut': 10 }
```

# pop(key[, default ])



Метод dict.pop() вернет значение ключа key, а также удалит его из словаря dict. Если ключ не найден, то вернет значение по умолчанию default.

```
>>> x = {'one': 0, 'two': 20, 'three': 3}
>>> x.pop('three')
3
>>> x
{'one': 0, 'two': 20}
>>> x.pop('three', 150)
150
>>> x.pop('three')
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# KeyError: 'ten'
```

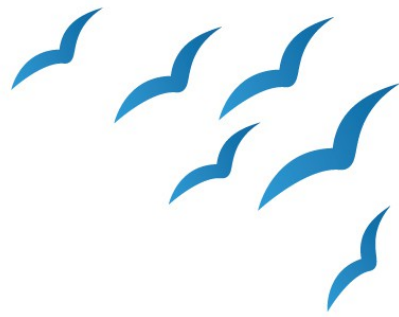


# values()

Метод `dict.values()` возвращает новый список-представление всех значений `dict_values`, содержащихся в словаре `dict`.

Список-представление значений `dict_values`, является динамичным объектом. Это значит, что все изменения, такие как удаление, изменение или добавление значений в словаре сразу отражаются на этом представлении.

```
>>> x = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> values = x.values()
>>> values
# dict_values([1, 2, 3, 4])
```



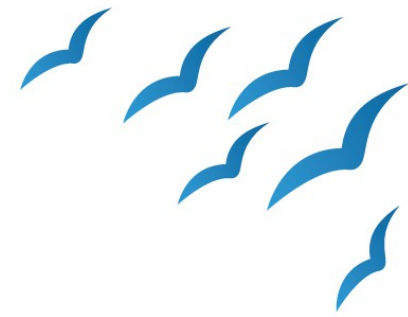
# popitem()

Метод `dict.popitem()` удалит и вернет двойной кортеж `(key, value)` из словаря `dict`. Пары возвращаются с конца словаря, в порядке **LIFO** (последним пришёл – первым ушёл)

```
>>> x = {'one': 0, 'two': 20, 'three': 3}
>>> x.popitem()
('four', 4)
>>> x.popitem()
('three', 3)
>>> x.popitem()
('two', 20)
>>> x.popitem()
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# KeyError: 'popitem(): dictionary is empty'
```



# IMPORTANT METHODS IN PYTHON



| SET   | LIST   | DICTIONARY   |
|---|--|--|
| <ul style="list-style-type: none"><li>• <b>add()</b></li><li>• <b>clear()</b></li><li>• <b>pop()</b></li><li>• <b>union()</b></li><li>• <b>issuperset()</b></li><li>• <b>issubset()</b></li><li>• <b>intersection()</b></li><li>• <b>difference()</b></li><li>• <b>isdisjoint()</b></li><li>• <b>setdiscard()</b></li><li>• <b>copy()</b></li></ul> | <ul style="list-style-type: none"><li>• <b>append()</b></li><li>• <b>copy()</b></li><li>• <b>count()</b></li><li>• <b>insert()</b></li><li>• <b>reverse()</b></li><li>• <b>remove()</b></li><li>• <b>sort()</b></li><li>• <b>pop()</b></li><li>• <b>extend()</b></li><li>• <b>index()</b></li><li>• <b>clear()</b></li></ul> | <ul style="list-style-type: none"><li>• <b>copy()</b></li><li>• <b>clear()</b></li><li>• <b>fromkeys()</b></li><li>• <b>items()</b></li><li>• <b>get()</b></li><li>• <b>keys()</b></li><li>• <b>pop()</b></li><li>• <b>values()</b></li><li>• <b>update()</b></li><li>• <b>setdefault()</b></li><li>• <b>popitem()</b></li></ul> |

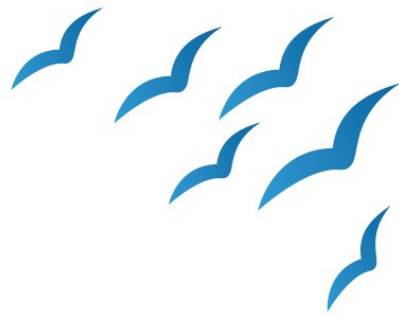
# Множества set



Множество — неупорядоченный набор элементов. Каждый элемент в множестве уникален (т. е. повторяющихся элементов нет) и неизменяем.

```
>>> data_scientist =  
set(['Python', 'R', 'SQL', 'Pandas', 'Git'])
```

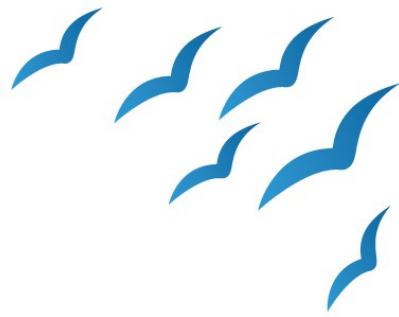
```
>> data_engineer =  
set(['Python', 'Java', 'Hadoop', 'SQL', 'Git' ])
```



# Задание множества

Что будет при дублировании значения ?

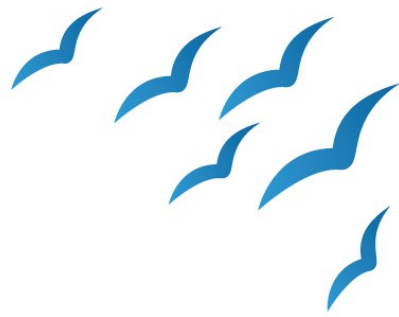
```
>>> data_scientist =  
set(['Python', 'R', 'R', 'SQL', 'Pandas', 'Git'])  
  
>>> type(data_scientist)  
<class 'set'>
```



# Задание множества

Мы также можем создать множество с элементами разных типов. Например:

```
>>> mixed_set = {2.0, "Nicholas", (1, 2, 3)}  
>>> print(mixed_set)  
{'Nicholas', 2.0, (1, 2, 3)}
```

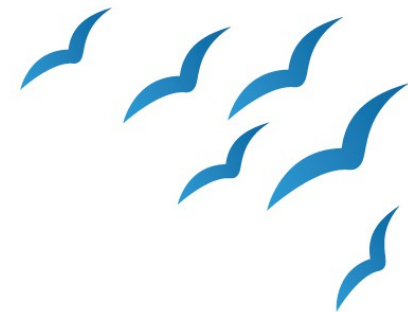


# Задание множества

Мы также можем создать множество из списков.

```
>>> num_set = set([1, 2, 3, 4, 5, 6])  
>>> print(num_set)
```

# Итерирование множества



```
months = {"Jan", "Feb", "March", "Apr",  
"May", "June", "July", "Aug", "Sep", "Oct",  
"Nov", "Dec"}
```

```
for m in months:
```

```
    print(m)
```

```
# проверка на членство в множестве
```

```
print("May" in months)
```

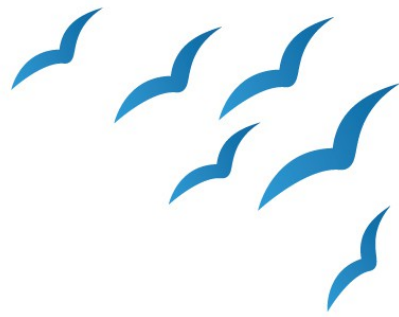


# Добавление элементов

```
months = set(["Jan", "March", "Apr", "May",  
"June", "July", "Aug", "Sep", "Oct", "Nov",  
"Dec"])
```

```
months.add("Feb")
```

```
print(months)
```



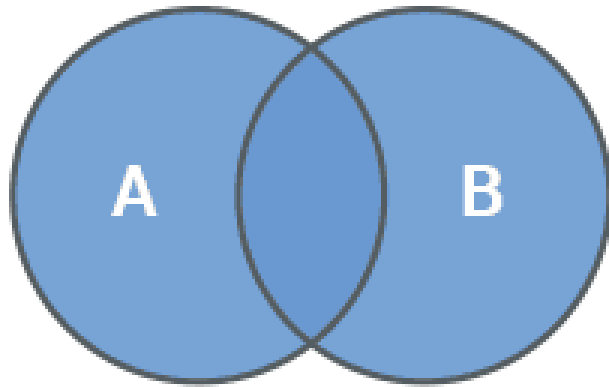
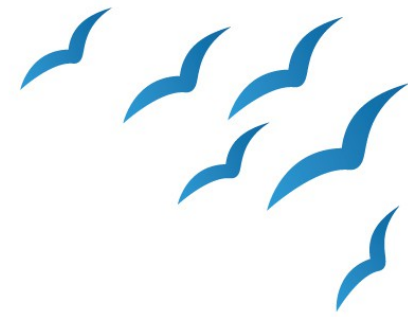
# Удаление элемента из МНОЖЕСТВ

```
>>>num_set = {1, 2, 3, 4, 5, 6}  
>>>num_set.discard(3)  
>>>print(num_set)  
{1, 2, 4, 5, 6}
```

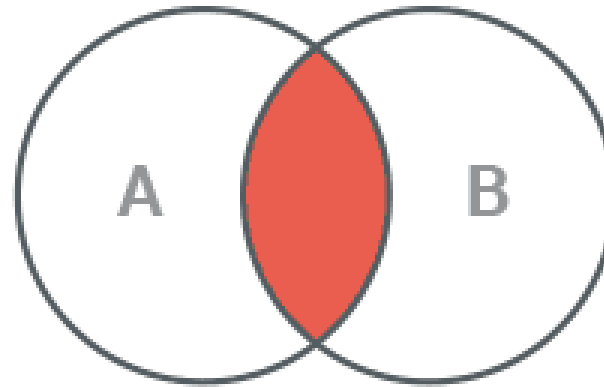
Метод `num_set.remove(7)`  
аналогичный но вызовет ошибку при  
отсутствии элемента.



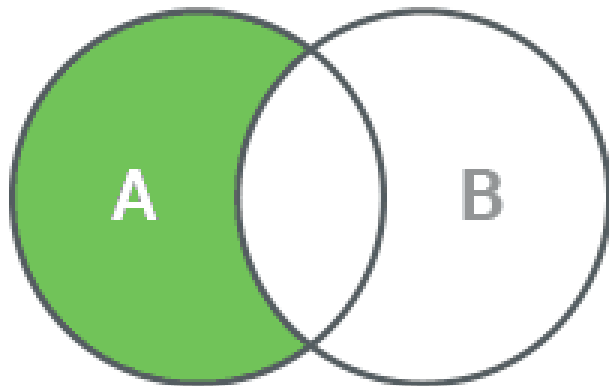
# Из теории множеств



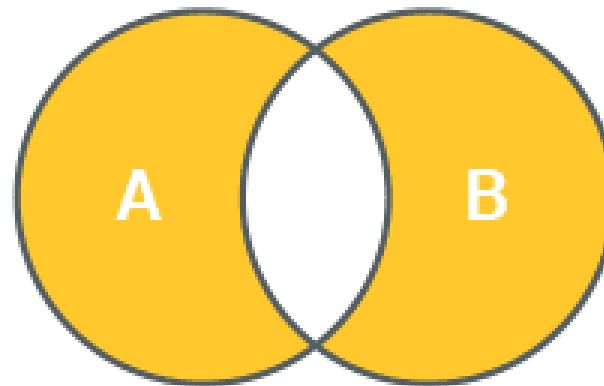
*Union*



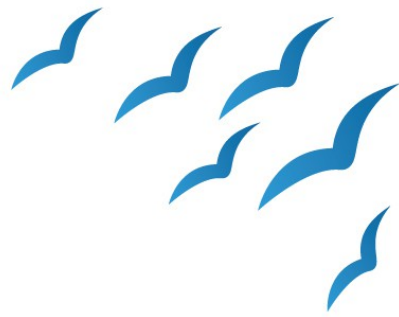
*Intersection*



*Difference*

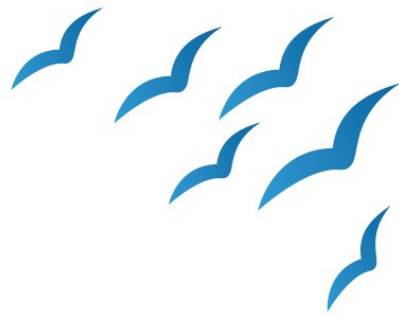


*Symmetric Difference*



# Объединение множеств

```
>>> months_a = set(["Jan", "Feb", "March", "Apr",  
"May", "June"])  
>>> months_b = set(["July", "Aug", "Sep", "Oct",  
"Nov", "Dec"])  
  
>>> all_months = months_a.union(months_b)  
print(all_months)  
{'Oct', 'Jan', 'Nov', 'May', 'Aug', 'Feb', 'Sep',  
'March', 'Apr', 'Dec', 'June', 'July'}
```



# union() или оператор |

Объединение может состоять из более чем двух множеств

```
x = {1, 2, 9}
```

```
y = {4, 5, 6}
```

```
z = {7, 8, 9}
```

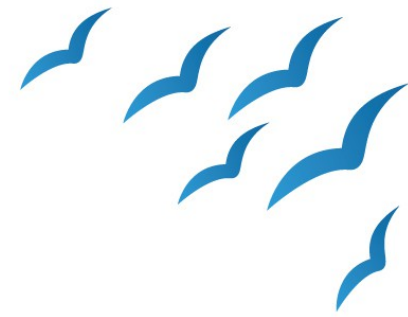
```
output = x.union(y, z)
```

```
print(output)
```

```
Python
```

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

```
print(x | y | z )
```



# Пересечение множеств

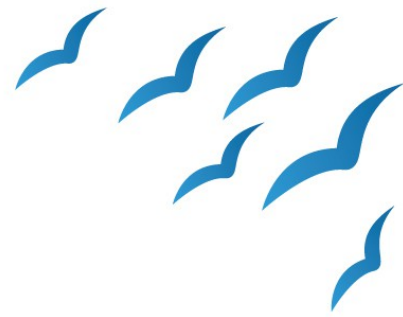
```
x = {1, 2, 3}
y = {4, 3, 6}
z = x.intersection(y)
print(z) #
```

```
x = {1, 2, 3}
y = {4, 3, 6}
```

```
print(x & y)
```

3

# Разница между множествами



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

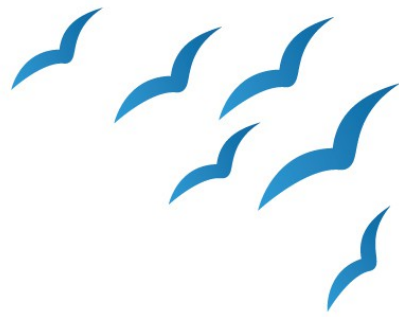
```
set_b = {4, 5, 6, 7, 8}
```

```
diff_set = set_a.difference(set_b)
```

```
print(diff_set)
```

```
{1, 2, 3}
```

```
print(set_a - set_b)
```



# Симитричная разницa

```
set_a = {1, 2, 3, 4, 5}
set_b = {4, 5, 6, 7, 8}
symm_diff = set_a.symmetric_difference(set_b)

print(symm_diff)
{1, 2, 3, 6, 7, 8}

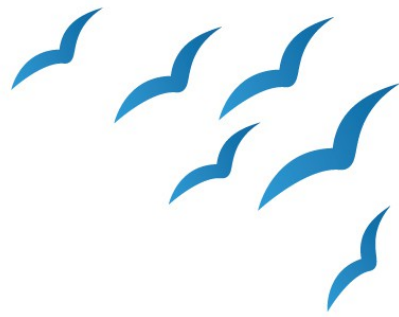
print(set_a ^ set_b)
```



# Сравнение множеств

Чтобы проверить, является ли множество А дочерним от В, мы можем выполнить следующую операцию:

```
months_a = set(["Jan", "Feb", "March", "Apr", "May",  
"June"])  
months_b = set(["Jan", "Feb", "March", "Apr", "May",  
"June", "July", "Aug", "Sep", "Oct", "Nov", "Dec"])  
  
# Чтобы проверить, является ли множество В  
# подмножеством А  
subset_check = months_a.issubset(months_b)  
  
# Чтобы проверить, является ли множество А  
# родительским множеством  
superset_check = months_b.issuperset(months_a)  
  
print(subset_check)  
print(superset_check)
```



# Метод `isdisjoint()`

Этот метод проверяет, является ли множество пересечением или нет. Если множества не содержат общих элементов, метод возвращает `True`, в противном случае — `False`.

```
names_a = {"Nicholas", "Michelle", "John",  
           "Mercy"}  
  
names_b = {"Jeff", "Bosco", "Teddy", "Milly"}  
  
x = names_a.isdisjoint(names_b)  
  
print(x)  
  
True
```



# Frozenset в Python



Frozenset (замороженное множество) – это неизменные МНОЖЕСТВА.

```
X = frozenset ([1, 2, 3, 4, 5, 6])
```

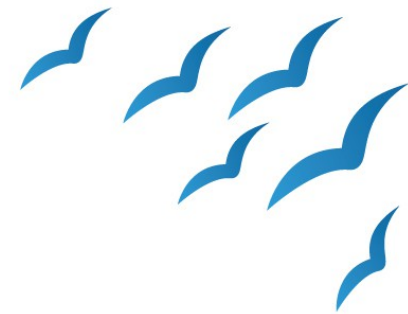
```
Y = frozenset ([4, 5, 6, 7, 8, 9])
```

```
print (X)
```

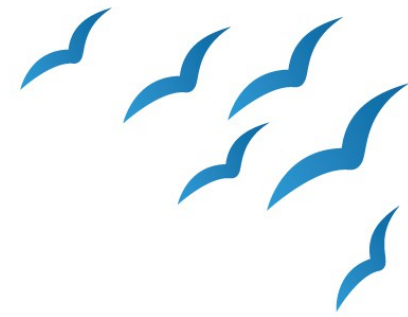
```
print (Y)
```

Какие методы для него не определены?

# Картежи



- Они являются упорядоченными коллекциями произвольных объектов
- Поддержка доступа по индексу
- Неизменяемые последовательности
- Имеют фиксированную длину
- Представляют из себя массив ссылок на объекты



# Упаковка кортежа

```
# пустой кортеж
```

```
empty_tuple = ()
```

```
# кортеж из 4-х элементов разных типов
```

```
four_el_tuple = (36.6, 'Normal', None, False)
```

```
type(four_el_tuple)
```

```
<class 'tuple'>
```

# Упаковка единственного элемента



```
tuple_one = ('a',)
```

```
tuple_two = 'b',
```

```
string_three = 'c'
```

```
print(type(is_tuple))
```

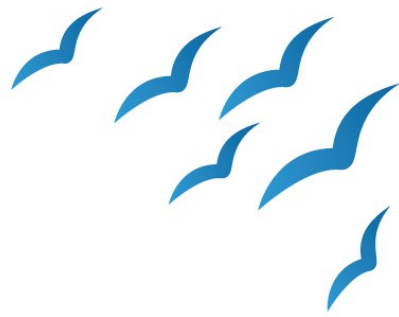
```
print(type(is_tuple_too))
```

```
print(type(string_three))
```

```
<class 'tuple'>
```

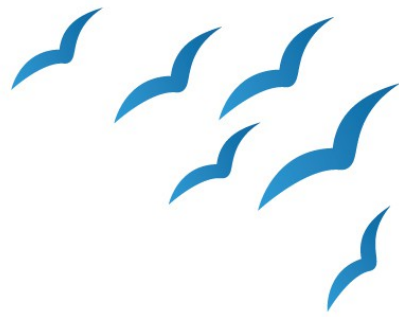
```
<class 'tuple'>
```

```
<class 'str'>
```



# Природа множественного присваивения

```
x, y = 100, 200
~
( x, y ) = (100, 200)
print(x)
100
print(y)
200
x, y = 'ML'      -   ?
```



# Вложенные кортежи

# пример tuple, что содержит вложенные элементы

```
nested_elem_tuple = (('one', 'two'),  
['three', 'four'], {'five': 'six'},  
(('seven', 'eight'), ('nine', 'ten')))  
print(nested_elem_tuple)
```



# Множественное присвоение(распаковка)

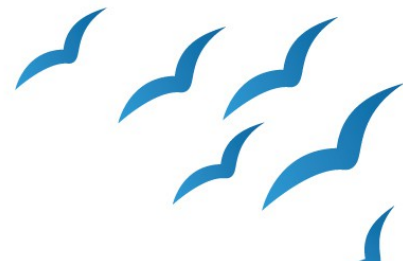
```
notes = ('Do', 'Re', 'Mi', 'Fa', 'Sol', 'La',  
         'Si')
```

```
do, re, mi, fa, sol, la, si = notes
```

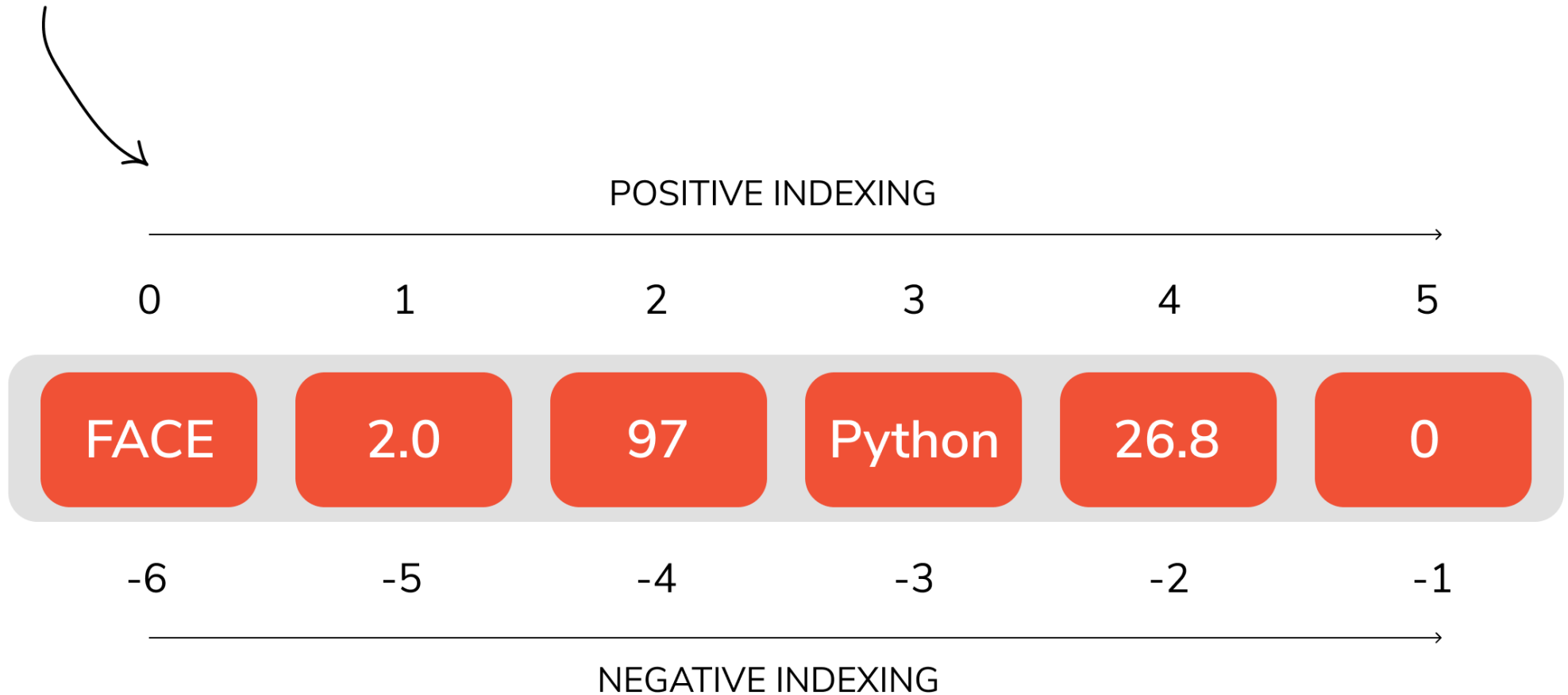
```
print(mi)
```

```
Mi
```

# Индексация



Tuple = ('FACE', 2.0, 97, 'Python', 26.8, 0 )





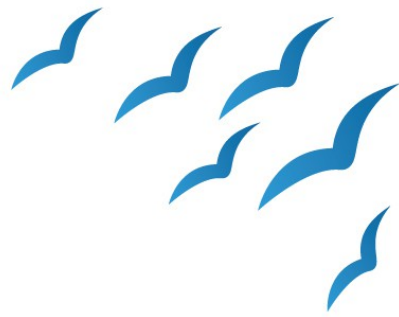


# Доступ к элементам

```
>>> a = (1, 2, 3, 4, 5)
>>> print(a[0])
1
>>> print(a[1:3])
(2, 3)
```

**# Что произойдет ?**

```
>>> a[1] = 3
```



# Операции

Сложение

```
(1, 2) + (3, 4)
```

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

Умножение

```
(1, 2) * 3
```

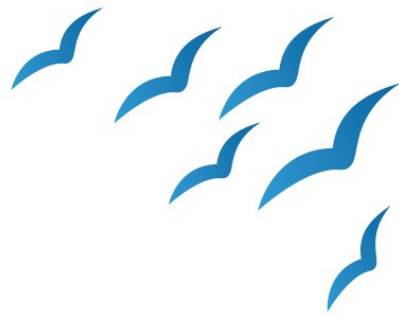
```
(1, 2, 1, 2, 1, 2)
```

Вхождение в кортеж

```
t_str = ('spam',)
```

```
'spam' in t_str
```

```
True
```



# Сравнение

```
tuple_A = 2 * 2,  
tuple_B = 2 * 2 * 2,  
tuple_C = 'a',  
tuple_D = 'z',
```

```
# при сравнении кортежей, числа сравниваются по  
значению
```

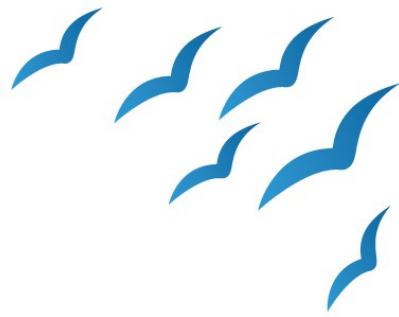
```
print(tuple_A < tuple_B)
```

```
> True
```

```
# строки в лексикографическом порядке
```

```
print(tuple_C < tuple_D)
```

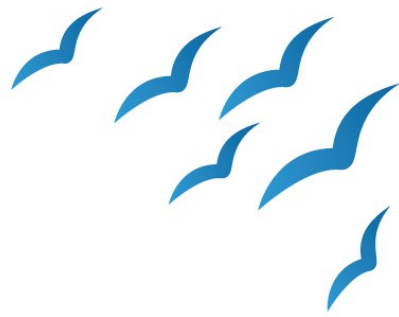
```
> True
```



# Итерирование кортежа

```
language_token = ('if', 'for', 'while', 'list',  
'dict', 'tuple', 'set')
```

```
# Вывести все элементы кортежа  
for word in my_tuple:  
    print(word)
```



# Сортировка

```
not_sorted_tuple = (10**5, 10**2, 10**1, 10**4,  
10**0, 10**3)
```

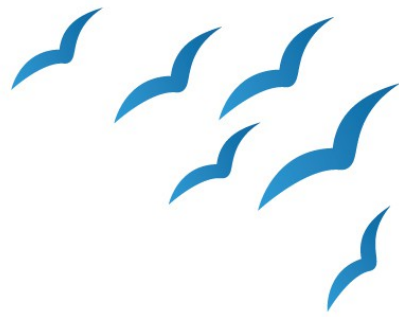
```
print(not_sorted_tuple)
```

```
> (100000, 100, 10, 10000, 1, 1000)
```

```
sorted_tuple = tuple(sorted(not_sorted_tuple))
```

```
print(sorted_tuple)
```

```
> (1, 10, 100, 1000, 10000, 100000)
```



# Удаление

```
some_useless_stuff = ('sad', 'bad things', 'trans  
fats')
```

```
del some_useless_stuff
```

```
print(some_useless_stuff)
```

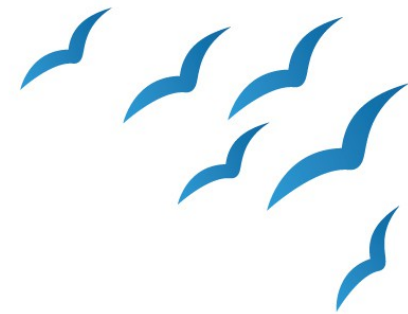
```
>
```

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

```
    print(some_useless_stuff)
```

```
NameError: name 'some_useless_stuff' is not  
defined
```

# Срезы



Слайсы кортежей **`tuple[start:fin:step]`**

Где `start` — начальный элемент среза (включительно), `fin` — конечный (не включительно) и `step` — "шаг" среза.

```
float_tuple = (1.1, 0.5, 45.5, 33.33, 9.12, 3.14, 2.73)
```

```
print(float_tuple[0:3])
```

```
> (1.1, 0.5, 45.5)
```

```
# выведем элементы с шагом 2
```

```
print(float_tuple[-7::2])
```

```
> (1.1, 45.5, 9.12, 2.73)
```

# Python Tuple Methods

tuple.  
    → count()  
    → index()







# Индекс заданного элемента **index(value, start, stop)**

```
rom = ('I', 'II', 'III', 'IV', 'V', 'VI', 'VII',  
      'VIII', 'IX', 'X')
```

```
print(rom.index('X'))
```

9

```
str = ('aa', 'bb', 'aa', 'cc')
```

```
print(str.index('aa'))
```

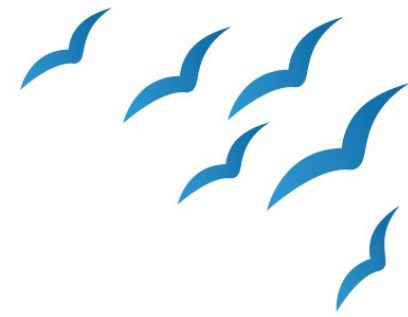
0

```
str = ('aa', 'bb', 'aa', 'cc' )
```

```
print(str.index('aa', 1, len(str)))
```

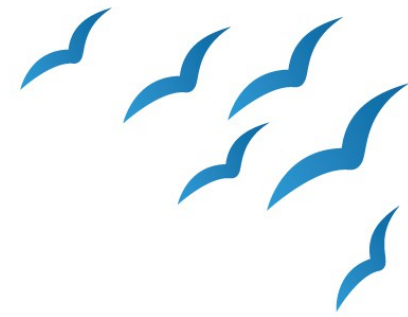
```
print(str.index('aa', 1,))
```

2



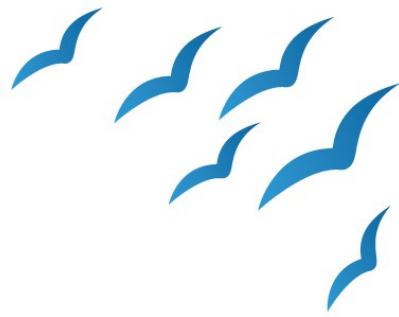
# Число вхождений элемента **count()**

```
t_str = ('aa', 'bb', 'aa', 'cc')  
print(t_str.count('aa'))  
2
```



# Длина кортежа **len()**

```
python_ = ('p', 'y', 't', 'h', 'o', 'n')  
print(len(python_))
```



# Преобразование tuple → str

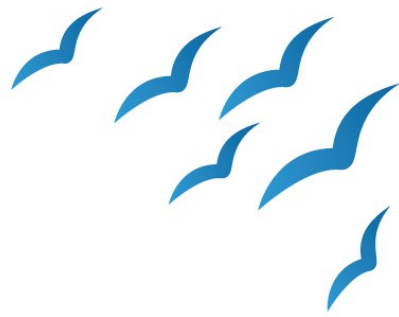
#Кортеж в строку

```
game_name = ('Breath', ' ', 'of', ' ', 'the',  
            ' ', 'Wild')
```

```
game_name = ' '.join(game_name)
```

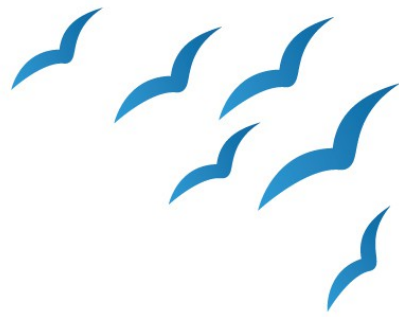
```
print(game_name)
```

```
Breath of the Wild
```



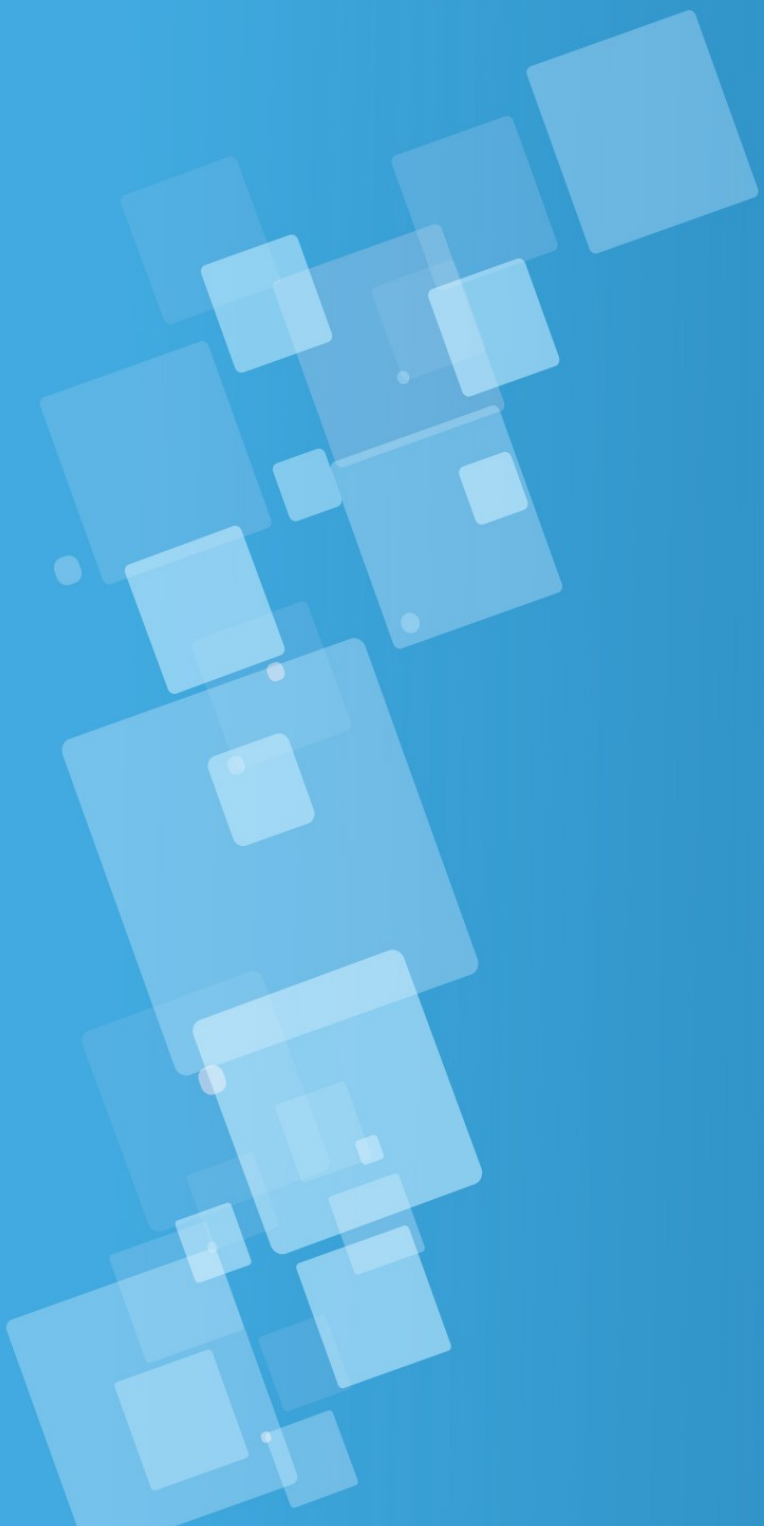
# Преобразование tuple → list

```
dig_tuple = (1111, 2222, 3333)
print(dig_tuple)
> (1111, 2222, 3333)
dig_list = list(dig_tuple)
print(dig_list)
[1111, 2222, 3333]
```



# Преобразование tuple → dict

```
# Преобразование через генератор  
словарей  
person = (('name', 'Piter' ), ('age', 100))  
p_dict = dict((x, y) for x, y in person)  
print(p_dict)  
{'name': 'Piter', 'age': 100}
```



# Функции часть I



# Определение функции

# объявление функции `my_function()`

```
def my_function([параметр1, параметр2, ...]):  
    # тело функции
```

```
    # возвращаемое значение  
    return result # необязательно
```

```
# вызов функции  
my_function([аргумент1 , аргумент2, ...] )
```

```
type(my_function)  
<class 'function'>    - еще один тип в Python
```





## Пример:

#Определение функции:

```
def summ(x, y):  
    result = x + y  
    return result
```

#ВЫЗОВ функции

```
a = 100
```

```
b = -50
```

```
answer = summ(a, b)
```

```
print(answer)
```

```
50
```



# Вызов функции

Можно ли так вызвать функцию ?

```
summ(10)
```

```
def summ(x):  
    print(x)
```

**NameError:** name 'summ' is not defined



## Что вернет функция без return:

#Определение функции:

```
def summ(x, y):  
    result = x + y
```

#ВЫЗОВ функции

```
answer = summ(100, -50)  
print(answer)      - ?
```

answer is None

True



# Что вернет функция в отсутствие аргументов ?

#Определение функции:

```
def summ(x, y):  
    result = x + y  
    return result;
```

#ВЫЗОВ функции

```
answer = summ(100)
```

```
TypeError: summ() missing 1 required positional  
argument: 'y'
```



## Параметры по умолчанию

Для некоторых параметров в функции можно указать значение по умолчанию, таким образом если для этого параметра не будет передано значение при вызове функции, то ему будет присвоено значение по умолчанию.

```
def premium(salary, percent=10):  
    p = salary * percent / 100  
    return p
```

```
result = premium(60000)  
print(result)
```

```
6000.0
```

```
print(premium(60000, 20))
```

```
12000.0
```



# Пустая функция

```
def empty(var1, var2):  
    pass
```

```
result = empty(8, 10)  
print(result)
```

**None**



# Именованные параметры


Иногда происходит такая ситуация, что функция требует большое количество аргументов. Пример:

```
def my_func(arg1, arg2, arg3, arg4, arg5, arg6):  
    pass # оператор, если кода нет
```

```
result = my_func(1, 2, 3, 4, 5, 6)
```

В такой ситуации код не всегда удобно читать и тяжело понять, какие переменные к каким параметрам относятся.

```
result = my_func(arg2=2, arg1=1, arg4=4, arg5=5,  
                arg3=3, arg6=6)
```



# Функция с неограниченным количеством позиционных аргументов

**\*args** – произвольное число позиционных аргументов


```
def manyargs(var1, *args):  
    print(type(args))  
    print(args)
```

```
result = manyargs(4, 9, 1, 3, 3, 1)
```

```
<class 'tuple'>
```

```
(9, 1, 3, 3, 1)
```





# Функции с неограниченным количеством именованных аргументов

**\*\*kwargs** – произвольное число именованных аргументов.

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

```
manykwargs(name='Piter', age=20)
```

```
<class 'dict'>
```

```
{'name': 'Piter', 'age': 20}
```

Можно ли мне по другому назвать параметр **\*\*kwargs** ?

Можно, только вас никто не поймет!



## Все вместе.

**\*args** – произвольное число позиционных аргументов

**\*\*kwargs** – произвольное число именованных аргументов.

```
def many_all(var1, *args, **kwargs):  
    print(var1)  
    print(args)  
    print(kwargs)
```

```
many_all(10, 34, 77, name='Piter', age=20)
```

```
10
```

```
(34, 77)
```

```
{'name': 'Piter', 'age': 20}
```

Можно ли мне по другому назвать параметр **\*\*kwargs** ?

Можно, только вас никто не поймет!



# Scope (область видимости)

**Область видимости** указывает интерпретатору, когда наименование (или переменная) видима. Другими словами, область видимости определяет, когда и где вы можете использовать свои переменные, функции и т.д. Если вы попытаетесь использовать что-либо, что не является в вашей области видимости, вы получите ошибку `NameError`. Python содержит три разных типа области видимости:

- \* Локальная область видимости
- \* Глобальная область видимости
- \* Нелокальная область видимости (была добавлена в Python 3)



# Локальная область видимости

Локальная область видимости (`local`) — это блок кода или тело любой функции Python или лямбда-выражения. Эта область Python содержит имена, которые вы определяете внутри функции.

```
x = 100
```

```
def doubling(y):
```

```
    z = y*y
```

```
doubling(x)
```

```
print(z)
```

```
NameError: name 'z' is not defined
```



# Глобальная область видимости

Глобальная область видимости(global). В Python есть ключевое слово `global`, которое позволяет изменять изнутри функции значение глобальной переменной. Оно записывается перед именем переменной, которая дальше внутри функции будет считаться глобальной.

```
x = 100
```

```
def doubling(y):
```

```
    global x
```

```
    x = y*y
```

```
doubling(x)
```

```
print(x)
```

```
10000
```



## Нелокальная область видимости

В Python 3 было добавлено новое ключевое слово под названием **nonlocal**. С его помощью мы можем добавлять переопределение области во внутреннюю область.

```
def counter():  
    num = 0  
    def incrementer():  
        num += 1  
        return num  
    return incrementer
```

Если вы попыгаете запустить этот код, вы получите ошибку **UnboundLocalError**, так как переменная num ссылается прежде, чем она будет назначена в самой внутренней функции.



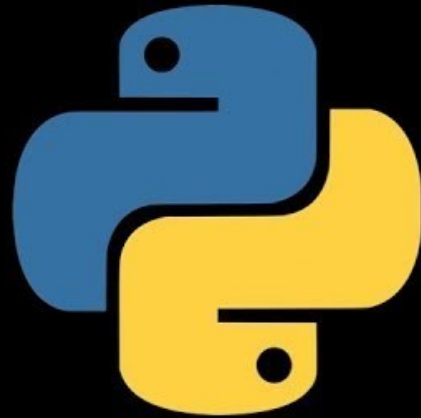
# Нелокальная область видимости

Добавим `nonlocal` в наш код:

```
def counter():  
    num = 0  
    def incrementer():  
        nonlocal num  
        num += 1  
        return num  
    return incrementer()
```

```
inc = counter()
```

```
print(inc) → 1
```



# PYTHON

# PROGRAMMING