

## Тема 3. Функции

1	Функции .....	1
1.1	Именные функции.....	1
1.2	Области видимости переменных в функциях .....	12
1.3	Алгоритм создания функции .....	15
2	Полезные инструменты .....	17
2.1	Импортирование.....	17
2.2	Запуск скрипта с параметрами .....	19
2.3	Генераторы списков и словарей .....	21
2.4	Модуль random как генератор псевдослучайных чисел .....	23
2.5	Конструкция yield .....	27
2.6	Модуль functools .....	28
2.7	Модуль itertools .....	29
2.8	Модуль math .....	32
2.9	Исключения в python .....	33
	Сводная таблица «Функции builtins» .....	37

# 1 Функции

## 1.1 Именные функции

В Python функция – это объект. Он принимает аргументы, выполняет с ними определённые операции и возвращает результат (значение). Функция определяется с помощью инструкции **def**, после которой следует имя функции. В Python они относятся к объектам первого класса, то есть к элементам, которые могут передаваться в качестве параметра, возвращаться из функции, присваиваться переменной. Пример:

```
def my_sum(arg_1, arg_2):  
    return arg_1 + arg_2  
  
print(my_sum(20, 30))  
print(my_sum("abra", "kadabra"))
```

Результат:

```
50  
abrakadabra
```

В примере представлена простейшая функция, которая принимает два параметра. В зависимости от типов данных параметров, результатом функции может быть число или строка. Инструкция **return** указывает, что функция должна вернуть.

Функция может содержать вложенные функции и возвращать объекты различных типов (списки, словари, функции).

Пример:

```
def ext_func(var_1):  
    def int_func(var_2):  
        return var_1 + var_2  
    return int_func  
  
f_obj = ext_func(200) # f_obj - функция  
print(f_obj(300))
```

Результат:

```
500
```

Структура функции определяется спецификой задачи. Оператор **return** не используется, если функция выполняет некоторые действия, но не возвращает значения. В этом случае возвращаемое значение — **None**.

Пример:

```
def my_func():  
    pass  
  
print(my_func())
```

Результат:

```
None
```

Здесь в теле функции реализован оператор-заглушка. Его использование равноценно отсутствию операции. **Pass** может применяться в тех случаях, когда код на текущий момент не написан.

### ***Оператор return***

О назначении этого оператора уже говорилось выше. Функции могут принимать данные и возвращать определённый результат после их обработки. При этом для выхода из функции и передачи результата в точку вызова применяется оператор **return**.

#### ***return со значением***

Если при выполнении логики функции интерпретатор Python встречает оператор **return**, то забирает значение, определённое после оператора, и выполняет выход из функции.

Рассмотрим следующий пример — расчёт полной площади цилиндра:

Пример:

```
def s_calc():  
    r_val = float(input("Укажите радиус: "))  
    h_val = float(input("Укажите высоту: "))  
    # площадь боковой поверхности цилиндра:  
    s_side = 2 * 3.14 * r_val * h_val  
    # площадь одного основания цилиндра:  
    s_circle = 3.14 * r_val ** 2  
    # полная площадь цилиндра:  
    s_full = s_side + 2 * s_circle  
    return s_full  
  
s_val = s_calc()  
print(s_val)
```

### Результат:

```
Укажите радиус: 4
Укажите высоту: 3
175.84
```

Здесь в главную ветку из функции возвращается значение локальной переменной **s\_full**, не сама переменная, а её значение. Это число, результат вычисления площади цилиндра. Подробнее о локальных и глобальных переменных поговорим позднее.

В главной ветке программы значение получает глобальная переменная **s\_val**. Выражение **s\_val = s\_calc()** работает следующим образом:

1. Вызов функции **s\_calc()**.
2. Возврат из функции значения.
3. Присвоение полученного значения переменной **s\_val**.

**Важно!** Необязательно присваивать некоторой переменной вычисленное в функции значение. Его можно вывести напрямую на экран.

### Пример:

```
print(s_calc())
```

Число, вычисленное в **s\_calc()**, получает функция **print()**. Например, вы написали только **s\_calc()**, при этом не присвоили полученные данные некоторой переменной или передали куда-то далее в программе. В этом случае синтаксической ошибки не будет, но данные будут потеряны.

### *return без значения*

В функции можно реализовать несколько операторов **return**. Но по итогам работы функций может быть выполнен только один. Это оператор **return**, его поток выполнения программы достигнет первым.

### Пример:

```
def s_calc():
    try:
        r_val = float(input("Укажите радиус: "))
        h_val = float(input("Укажите высоту: "))
    except ValueError:
        return
    s_side = 2 * 3.14 * r_val * h_val
    s_circle = 3.14 * r_val ** 2
```

```
s_full = s_side + 2 * s_circle
return s_full

print(s_calc())
```

В этом примере предусмотрена ситуация, когда пользователь вводит некорректные данные. Например, вместо числа (радиус, высота) он ввёл строку. При этом реализована обработка исключения, которое возникнет при попытке выполнения арифметической операции умножения со строками. В ветке **except** при возникновении исключения осуществляется выход из функции без вычисления площади цилиндра.

Результат:

```
Укажите радиус: radius
None
```

В результате функция возвращает объект типа **None**. Такое значение оператор **return** возвращает по умолчанию, но можно указать его явно: **return None**.

**Важно!** Если в функции отсутствует оператор **return**, не значит, что она ничего не возвращает. На самом деле возвращает. Только это значение не присваивается какой-либо переменной и не выводится на экран. В Python любая функция что-то возвращает. Если оператор **return** отсутствует, то возвращаемое значение — **None**.

### ***Возврат набора значений***

В Python возможно использование оператора **return**, возвращающего из функции несколько объектов. Достаточно указать их через запятую после оператора **return**.

Пример:

```
def s_calc():
    try:
        r_val = float(input("Укажите радиус: "))
        h_val = float(input("Укажите высоту: "))
    except ValueError:
        return
    s_side = 2 * 3.14 * r_val * h_val
    s_circle = 3.14 * r_val ** 2
    s_full = s_side + 2 * s_circle
    return s_side, s_full
```

```
s_side_val, s_full_val = s_calc()
print(f"Площадь боковой пов-ти - {s_side_val}; Полная площадь - {s_full_val}")
```

### Результат:

```
Укажите радиус: 4
Укажите высоту: 3
Площадь боковой пов-ти - 75.36; Полная площадь - 175.84
```

Функция `s_calc()` возвращает два значения, присваиваемые переменным `s_side_val` и `s_full_val`. Подобное групповое присвоение — важная характеристика Python.

Смысл в том, что перечисление значений через запятую формирует объект типа кортеж (tuple). Присваивая кортеж сразу набору переменных, его элементы сопоставляются переменным. Происходит своего рода распаковка.

То есть, когда функция возвращает набор объектов, на деле она возвращает объект-кортеж с этими объектами. Они упаковываются в кортеж перед возвратом. Если за оператором **return** следует только одна переменная, её тип сохраняется в исходном состоянии.

### *Аргументы функций*

Функция может принимать любое количество параметров или не принимать их вообще. Параметры могут быть позиционные и именованные, обязательные и необязательные.

### Пример:

```
# позиционные параметры
def first_func(var_1, var_2, var_3):
    return var_1 + var_2 + var_3

print(first_func(10, 20, 30))

# именованные параметры
def second_func(var_2, var_1, var_3):
    print(f"var_2 - {var_2}; var_1 - {var_1}; var_3 - {var_3}")

second_func(var_1=10, var_2=20, var_3=30)
```

### Результат:

```
60
var_2 - 20; var_1 - 10; var_3 - 30
```

Пример:

```
#                обязательные                параметры
def first_func(var_1, var_2, var_3):
    return var_1 + var_2 + var_3

print(first_func(10, 20, 30))

# var_2 и var_3 - необязательные параметры
def second_func(var_1, var_2=20, var_3=30):
    return var_1 + var_2 + var_3

print(second_func(10))
```

Результат:

```
60
60
```

Функция может принимать неопределённое число позиционных параметров. В этом случае при описании функции используется конструкция **\*args**.

Пример:

```
def my_func(*args):
    return args

print(my_func(10, "text_1", 20, "text_2"))
```

Результат:

```
(10, 'text_1', 20, 'text_2')
```

Из примера следует, что **args** представляет собой кортеж, содержащий переданные в функцию аргументы. С переменной **args** можно выполнять те же операции, что и с кортежем.

Функция может принимать и неопределённое число именованных параметров. Тогда используется конструкция **\*\*kwargs**.

Пример:

```
def my_func(**kwargs):
    return kwargs
```

```
print(my_func(el_1=10, el_2=20, el_3="text"))
```

Результат:

```
{'el_1': 10, 'el_2': 20, 'el_3': 'text'}
```

Переменная `kwargs` хранит словарь. С ним можно выполнять привычные для словаря операции.

Важно! Операторы `*` и `**` в Python можно использовать и с другими именами переменных. То есть имена **args** и **kwargs** необязательны. Но помните, что хороший стиль программирования подразумевает использование имён **args** и **kwargs**, так как сразу становится понятно о назначении таких переменных.

Пример:

```
def my_func(**kwargs):  
    return kwargs  
  
print(my_func(el_1=10, el_2=20, el_3="text"))
```

Результат:

```
{'el_1': 10, 'el_2': 20, 'el_3': 'text'}
```

### *Анонимные функции (lambda)*

Это функции, содержащие только одно выражение, но выполняющиеся быстрее именованных функций. При этом используется оператор **lambda**. При использовании **lambda**-функций их необязательно присваивать некоторой переменной, как в случае с именованными функциями. **lambda**-функции, в отличие от именованных, не требуют оператора **return**, в остальном — идентичны именованным.

Пример:

```
my_func = lambda p_1, p_2: p_1 + p_2  
  
print(my_func(2, 5))  
print(my_func("abra", "kadabra"))  
  
print((lambda p_1, p_2: p_1 + p_2)(2, 5))  
print((lambda p_1, p_2: p_1 + p_2)("abra", "kadabra"))  
  
new_func = lambda *args: args  
print(new_func(10, 20, 30, 40))
```

Результат:



```
7
abrakadabra
7
abrakadabra
(10, 20, 30, 40)
```

Другое название **lambda**-функции — анонимная или несвязанная.

Ещё пример:

```
def named_func(param):
    return param ** 0.5

print(named_func(100))

square_rt = lambda param: param ** 0.5
print(square_rt(100))
```

Результат:

```
10.0
10.0
```

### *Ещё раз о встроенных функциях*

В языке Python предусмотрены встроенные функции. Их логика работы скрыта от разработчиков, а имена зарезервированы. Достаточно знать, какие данные эти функции могут принимать и какой результат возвращать. С частью функций мы уже познакомились ранее (**input()**, **type()**, **int()**, **str()**, **float()**, **bool()**). Переводная версия документации, в которой описаны встроенные функции и их назначение, доступна по [ссылке](#).

Рассмотрим ещё две группы встроенных функций.

### *Функции для операций с символами*

Функция	Описание
ord()	Принимает Unicode-символ и возвращает соответствующий код (целое число)
chr()	Принимает целое число и возвращает Unicode-символ, соответствующий переданному числу (коду)
len()	Принимает любой объект-последовательность (строка, набор байтов, список, кортеж) или объект-коллекцию (словарь, множество) и возвращает число элементов последовательности

Пример:

```
print(ord("g"))  
print(chr(103))  
print(len("abracadabra"))
```

Результат:

```
103  
g  
11
```

### *Математические функции*

Функция	Описание
abs()	Принимает целое число или число с плавающей точкой. Возвращает абсолютное значение числа (по модулю)
round()	Принимает число с плавающей точкой. Округляет число до ближайшего целого числа. Может принимать число знаков после запятой, до которых необходимо выполнить округление
divmod()	Принимает два числа, возвращает также два числа (частное и остаток от деления чисел)
pow()	Принимает два числа. Позволяет возвести первое число в указанную степень
max()	Принимает итерируемый объект и возвращает самый большой элемент
min()	Принимает итерируемый объект и возвращает наименьший элемент
sum()	Суммирует элементы последовательности

Пример:

```
# abs()  
print(abs(2))  
print(abs(-2))
```

Результат:

```
2  
2
```

Пример:

```
# round()
print(round(2.6743))
print(round(-2.678))
print(round(2.6743, 2))
print(round(-2.678, 2))
```

Результат:

```
3
-3
2.67
-2.68
```

Пример:

```
# divmod()
print(divmod(4, 2))
print(divmod(5, 2))
```

Результат:

```
(2, 0)
(2, 1)
```

Пример:

```
# pow()
print(pow(2, 4))
```

Результат:

```
16
```

Пример:

```
# max()
iter_obj = [20, 2, 5, 100]
print(max(iter_obj))
iter_obj = (20, 2, 5, 100)
print(max(iter_obj))
iter_obj = "abracadabra"
print(max(iter_obj))
```

Результат:

```
100
100
r
```

Пример:

```
# min()
iter_obj = [20, 2, 5, 100]
print(min(iter_obj))
iter_obj = (20, 2, 5, 100)
print(min(iter_obj))
iter_obj = "abracadabra"
print(min(iter_obj))
```

Результат:

```
2
2
a
```

Пример:

```
# sum()
iter_obj = [20, 2, 5, 100]
print(sum(iter_obj))
iter_obj = (20, 2, 5, 100)
print(sum(iter_obj))
```

Результат:

```
127
127
```

### ***Функция range() для многократно выполняемых действий***

Отвечает за генерацию набора чисел в пределах указанного диапазона. Для этого можно использовать ещё один параметр — шаг.

Пример:

```
print(list(range(7))) # целые числа в диапазоне [0, 7)
print(list(range(2, 8))) # целые числа в диапазоне [2, 8)
print(list(range(1, 9, 2))) # целые числа в диапазоне [1, 9) с шагом 2
print(list(range(1, -7, -2))) # целые числа в диапазоне [1, -7) с шагом -2
```

```
print(list(range(0))) # целые числа в диапазоне (0, 0)
print(list(range(1, 0))) # целые числа в диапазоне (1, 0)
```

Результат:

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

Функция `range()` может использоваться в циклах:

Пример:

```
for el in range(4, 20, 4):
    res = el / 2
    print(f"Результат деления {el} на 2: {int(res)}")
```

Результат:

```
Результат деления 4 на 2: 2
Результат деления 8 на 2: 4
Результат деления 12 на 2: 6
Результат деления 16 на 2: 8
```

## 1.2 Области видимости переменных в функциях

Понятие «Область видимости» определяет, когда и в каких точках программы разработчик может применять свои пользовательские объекты (переменные, функции). В Python доступны следующие области видимости: локальная, глобальная, нелокальная.

### *Локальная область видимости*

Переменная, объявленная в рамках функции, по умолчанию имеет локальную область видимости. Рассмотрим ещё раз пример, представленный ранее.

Пример:

```
def full_s_calc():
    r_val = float(input("Укажите радиус: "))
    h_val = float(input("Укажите высоту: "))
    s_side = 2 * 3.14 * r_val * h_val
    s_circle = 3.14 * r_val ** 2
```

```
s_full = s_side + 2 * s_circle
return s_full

s_val = full_s_calc()
print(s_val)
print(s_side)
```

#### Результат:

```
Укажите радиус: 4
Укажите высоту: 3
175.84
Traceback (most recent call last):
  File "my_file.py", line 11, in <module>
    print(s_side)
NameError: name 's_side' is not defined
```

В этом примере попытки обратиться к переменным **r\_val**, **h\_val**, **s\_side**, **s\_circle** приведут к аварийному завершению работы программы, так как они локальные и доступны только в пределах функции **full\_s\_calc()**. Для решения этой проблемы переведите нужные локальные переменные в глобальную область видимости.

#### *Глобальная область видимости*

Оператор **global** позволяет определить глобальную область видимости для переменной, объявленной в рамках функции.

#### Пример:

```
def full_s_calc():
    global r_val, h_val, s_side, s_circle
    r_val = float(input("Укажите радиус: "))
    h_val = float(input("Укажите высоту: "))
    s_side = 2 * 3.14 * r_val * h_val
    s_circle = 3.14 * r_val ** 2
    s_full = s_side + 2 * s_circle
    return s_full

s_val = full_s_calc()
print(s_val)
print(s_circle)
```

#### Результат:

```
Укажите радиус: 4
Укажите высоту: 3
```

```
175.84
50.24
```

### *Нелокальная область видимости*

Перевод переменной в область видимости объемлющей функции. Пример:

```
def ext_func():
    my_var = 0
    def int_func():
        my_var += 1
        return my_var
    return int_func

func_obj = ext_func()
print(func_obj)
print(func_obj())
print(func_obj())
print(func_obj())
```

### Результат:

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

Ошибка возникает из-за того, что Python пытается увеличить значение переменной **my\_var** на единицу. Но исходное значение этой переменной не определено. То есть оно как бы определено, но вне области видимости функции **int\_func()**, и потому по умолчанию недоступно.

Решение проблемы — перевод переменной **my\_var** в нелокальную зону видимости. Пример:

```
def ext_func():
    my_var = 0
    def int_func():
        nonlocal my_var
        my_var += 1
        return my_var
    return int_func

func_obj = ext_func()
print(func_obj)
print(func_obj())
print(func_obj())
print(func_obj())
```

Результат:

```
<function ext_func.<locals>.int_func at 0x0000009E70C658C8>
1
2
3
```

Ещё один интересный момент. В этом примере объемлющая функция **ext\_func()** возвращает нам объект-функцию:

```
<function ext_func.<locals>.int_func at 0x0000009E70C658C8>
```

Следовательно, переменная **func\_obj** начинает ссылаться на объект-функцию, и значит, допустим вызов этой функции:

```
func_obj()
```

### ***1.3 Алгоритм создания функции***

Благодаря функциям разработчик получает возможность многократного использования кода. Это повышает модульность проекта, упрощает его последующую модификацию. Для создания функции можно использовать алгоритм, который рассмотрим на примере определения площади прямоугольника:

1. Определить название функции. Оно должно быть информативным, чтобы было понятно назначение функции.
2. Определить в строках документации назначение функции, типы данных её параметров и тип данных результата. Можно указать пример вызова функции с параметром и возвращаемый результат.
3. Определить информативные имена параметров, передаваемых в функцию, написать тело функции с возвращаемым результатом (при необходимости).
- 4.

Пример:

```
def rectangle_area_calc(length, width):
    """
    Возвращает площадь прямоугольника по длине и ширине

    (number, number) -> number

    >>> rectangle_area_calc(10, 10)
    100
```



```
"""
```

```
return length * width
```

## 2 Полезные инструменты

### 2.1 Импортирование

#### *Импортирование в Python*

«Импорт» и «модуль» — понятия пока незнакомые. Но о них нужно поговорить, поскольку функция вызывается не только в том файле, где она написана. Она может быть импортирована из другого файла с Python-кодом, называемого модулем.

Итак, модуль в Python — это файл с кодом, то есть некая программа, которую можно связать с другой. Есть встроенные модули, которые можно импортировать из стандартной библиотеки, и реализованные самим разработчиком. Благодаря модульному принципу программ мы можем связывать модули друг с другом и импортировать из них функции и классы для последующего использования.

#### *Импорт модуля из стандартной библиотеки*

Для этого применяется оператор **import**, за которым следует название модуля. С помощью одной инструкции импорта можно подключить к программе сразу несколько модулей. Но это ухудшает читаемость кода и не соответствует соглашениям PEP-8. Поэтому импортировать следует каждый модуль отдельно.

Рассмотрим применение оператора `random` с применением модулей **random** и **time**.

Пример:

```
import time
import random
print(time.time())
print(random.random())
```

Результат:

```
1563440619.2266152
0.7303585873639512
```

После импорта модуля его имя можно использовать как переменную, через которую доступны параметры и функции модуля.

#### *Использование инструкции from*

В примере, рассмотренном выше, импортируются модули целиком. Можно импортировать только определённые объекты модуля:

Пример:

```
from time import time
from random import random
print(time())
print(random())
```

Результат:

```
1563441483.3917782
0.5331559021496495
```

### *Создание собственного модуля*

Отметим ещё раз, что, создавая файл с программным кодом на Python (с расширением **.py**), вы фактически воплощаете модуль, в котором можно определить переменные, функции и классы. Создадим файл-модуль **my\_functions.py** и определим в нём две функции.

Пример:

```
def show_msg():
    print("Приветствие!")

def simple_calc():
    x = int(input("Введите значение x: "))
    return x ** 2 - 1
```

Теперь в директории с файлом **my\_functions.py** создадим ещё один файл, например, **main.py** и выполним подключение созданного ранее модуля **my\_functions.py**.

Пример:

```
import my_functions

my_functions.show_msg()
print(my_functions.simple_calc())
```

Результат:

```
Приветствие!
Введите значение x: 4
15
```

Можно записать по-другому:

```
from my_functions import show_msg
```

```
from my_functions import simple_calc

show_msg()
print(simple_calc())
```

## 2.2 Запуск скрипта с параметрами

Выполняя запуск скриптов, пользователь зачастую должен передавать в программу некоторые данные, необходимые для выполнения скрипта. Запрашивать их у пользователя можно интерактивно, в процессе работы скрипта. Для этого применяется функция **input()**, которая отвечает за получение данных от пользователя и их сохранение в переменных.

Есть и другое решение, суть которого заключается в передаче данных в скрипт прямо в момент его запуска. Этот механизм называется запуском скрипта с параметрами.

Рассмотрим работу этого механизма на примере. Создадим простой файл-модуль, например, с именем **script\_params\_test.py**, и добавим в него несколько простых инструкций:

Пример:

```
from sys import argv

script_name, first_param, second_param, third_param = argv

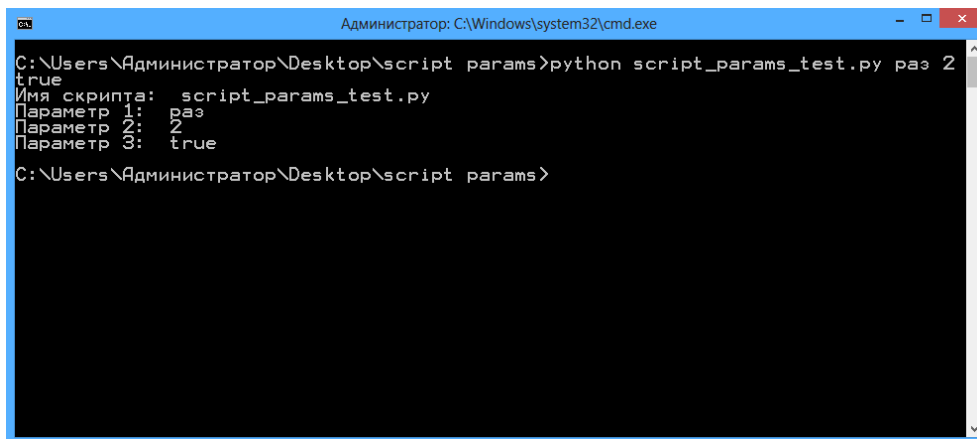
print("Имя скрипта: ", script_name)
print("Параметр 1: ", first_param)
print("Параметр 2: ", second_param)
print("Параметр 3: ", third_param)
```

Скрипт небольшой, но позволит отразить возможности передачи данных в программу. Для его запуска нужно вызвать командную строку — желательно из директории расположения скрипта — и запустить команду:

Пример:

```
python script_params_test.py paz 2 true
```

Результат:

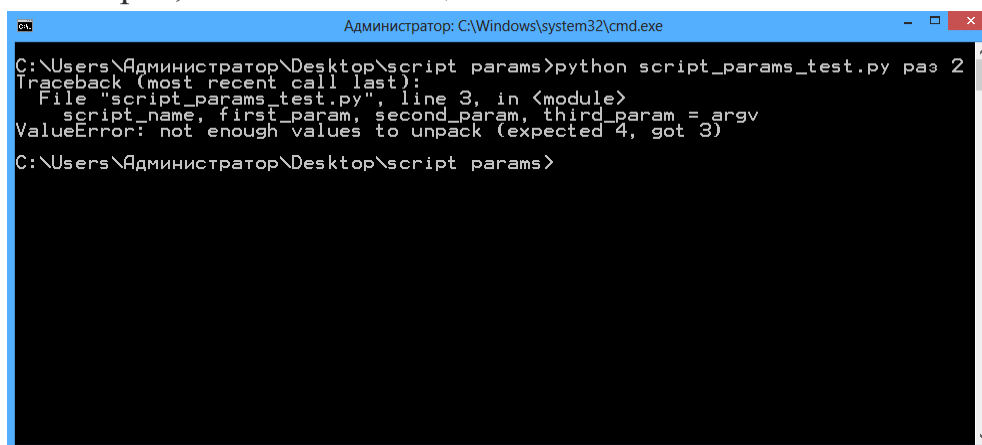


```
Администратор: C:\Windows\system32\cmd.exe
C:\Users\Администратор\Desktop\script params>python script_params_test.py паэ 2
true
Имя скрипта: script_params_test.py
Параметр 1: паэ
Параметр 2: 2
Параметр 3: true
C:\Users\Администратор\Desktop\script params>
```

В этом примере мы передали три параметра, отобразили их значения и сделали вывод имени скрипта. Теперь разберёмся с кодом подробнее.

Первая строка отвечает за импорт списка аргументов командной строки, переданных скрипту (**sys.argv**). В следующей — осуществляется распаковка содержимого списка **argv** в переменные. Мы как бы говорим интерпретатору Python, что он должен взять данные из списка **argv** и последовательно связать извлекаемые данные с каждой из переменных. Эти переменные указаны с левой стороны выражения. Далее мы можем выполнять нужные операции с представленными переменными.

Первый переданный параметр — имя скрипта. В качестве других параметров мы можем указать любые другие значения, отличные от примера. Но число этих значений должно совпадать с числом переменных в левой части выражения. Если, например, для этого скрипта попытаться передать два параметра вместо трёх, появится сообщение об ошибке:



```
Администратор: C:\Windows\system32\cmd.exe
C:\Users\Администратор\Desktop\script params>python script_params_test.py паэ 2
Traceback (most recent call last):
  File "script_params_test.py", line 3, in <module>
    script_name, first_param, second_param, third_param = argv
ValueError: not enough values to unpack (expected 4, got 3)
C:\Users\Администратор\Desktop\script params>
```

Сообщение об ошибке возникает из-за передачи в скрипт недостаточного числа параметров. Строка с **ValueError** сообщает, что, согласно логике скрипта, необходимо передать 4 значения вместо указанных трёх.

Используя подобный механизм, важно помнить, что передаваемые в скрипт параметры — строковые. Если параметры предполагается использовать дальше в программе, их нужно преобразовать в нужный тип данных.

## 2.3 Генераторы списков и словарей

Это механизм, миссия которого — быстрое создание и заполнение списков и словарей в Python. Генераторы предполагают использование итерируемого объекта. На его базе формируется новый список и выражение. Второе призвано выполнить с извлечёнными из итерируемого объекта элементами некоторые операции перед их включением в итоговый список.

### *Генераторы списков*

Генераторы — это пример так называемого синтаксического сахара в языке программирования Python. Это возможность использования таких инструкций кода, которые не меняют поведения программы. Они делают конструкции на Python более понятными.

Пример:

```
my_list = [2, 4, 6]
new_list = [el+10 for el in my_list]
print(f"Исходный список: {my_list}")
print(f"Новый список: {new_list}")
```

Результат:

```
Исходный список: [2, 4, 6]
Новый список: [12, 14, 16]
```

В приведённом примере функцию генератора выполняет выражение: **el+10 for el in my\_list**, где **my\_list** — итерируемый объект. Из него в цикле **for** поочерёдно извлекаются элементы. Перед инструкцией **for** указывается действие, которое выполняется над элементом перед добавлением его в новый список.

**Важно!** Генератор создаёт новый список, а не изменяет текущий.

Пример:

```
my_list = [2, 4, 6]
print(f"Исходный список: {my_list}")
new_list = []
for el in my_list:
    new_list.append(el + 10)
print(f"Новый список: {new_list}")
```

Результат:

```
Исходный список: [2, 4, 6]
Новый список: [12, 14, 16]
```

В цикле **for** возможен перебор не только элементов списка, но и строк файла. Пример:

```
lines = [line.strip() for line in open('text.txt')]
print(lines)
```

Результат:

```
['stroka_1', 'stroka_2', 'stroka_3']
```

В генератор допускается добавление условия. Пример:

```
my_list = [10, 25, 30, 45, 50]
print(my_list)
new_list = [el for el in my_list if el % 2 == 0]
print(new_list)
```

Результат:

```
[10, 25, 30, 45, 50]
[10, 30, 50]
```

Допустимо использовать вложенные циклы.

Пример:

```
str_1 = "abc"
str_2 = "d"
str_3 = "efg"
sets = [i+j+k for i in str_1 for j in str_2 for k in str_3]
print(sets)
```

Результат:

```
['ade', 'adf', 'adg', 'bde', 'bdf', 'bdg', 'cde', 'cdf', 'cdg']
```

Обратите внимание на следующий пример:

```
my_tuple = (2, 4, 6)
new_obj = (el+10 for el in my_tuple)

print(new_obj)
```

Результат:

```
<generator object <genexpr> at 0x0000008E23521138>
```

Здесь мы используем генераторное выражение для элементов кортежа, но в результате получаем объект-итератор. Такой результат связан с использованием круглых скобок в генераторном выражении. Если в этом примере заменить кортеж на список, результат будет идентичный (объект-итератор). Пример:

```
my_tuple = [2, 4, 6]
new_obj = (el+10 for el in my_tuple)

print(new_obj)
```

Результат:

```
<generator object <genexpr> at 0x0000003E13BB9620>
```

### ***Генераторы словарей и множеств***

Если в конструкции, определяющей генератор, вместо квадратных скобок указать фигурные, то результатом работы генератора будет словарь. Пример:

```
my_dict = {el: el*2 for el in range(10, 20)}
print(my_dict)
```

Результат:

```
{10: 20, 11: 22, 12: 24, 13: 26, 14: 28, 15: 30, 16: 32, 17: 34,
18: 36, 19: 38}
```

Генератор для множеств отличается незначительно. Пример:

```
my_set = {el**3 for el in range(5, 10)}
print(my_set)
```

Результат:

```
{512, 343, 216, 729, 125}
```

## ***2.4 Модуль random как генератор псевдослучайных чисел***

Модуль содержит специальные функции для генерации целых и дробных чисел. Рассмотрим использование этих функций на примерах.

### ***Генерация целых случайных чисел***

Применяются функции **randint()** и **randrange()**. Первая — самая простая в использовании. Она принимает два аргумента — нижняя и верхняя границы целочисленного диапазона, из которого выбирается число.



Пример:

```
import random
print(random.randint(0, 10))
```

Результат:

7

Для функции **randint()** значения и нижней, и верхней границы входят в диапазон, из которого определяется число.

Можно работать с функцией напрямую, импортируя из модуля.

Пример:

```
from random import randint
print(randint(0, 10))
```

Результат:

10

Левая граница всегда должна быть меньше правой. Допускается использование отрицательных чисел для определения границ диапазона.

Пример:

```
from random import randint
print(randint(-100, -10))
```

Результат:

-78

Функция **randrange()** устроена сложнее. Она может принимать от одного до трёх аргументов.

1. Один аргумент — возвращается случайное число от 0 до переданного аргумента. При этом сам аргумент в диапазон не включается.

2.

Пример:

```
from random import randrange
print(randrange(10))
```

Результат:

```
5
```

3. Два аргумента — возвращается случайное число в указанном диапазоне. При этом верхняя граница в диапазон не включается.

Пример:

```
from random import randrange
print(randrange(10, 20))
```

Результат:

```
17
```

4. Три аргумента. Первые два — нижняя и верхняя границы, третий — шаг. Например, для функции `randrange(20, 30, 3)` случайное число выбирается из чисел 20, 23, 26, 29.

Пример:

```
from random import randrange
print(randrange(20, 30, 3))
```

Результат:

```
26
```

### *Генерация дробных случайных чисел*

Такие числа называются вещественными, или числами с плавающей точкой. Самый простой способ получить вещественное число — применить функцию **`random()`** без параметров. Результат её работы — число с плавающей точкой от 0 до 1, не включая верхнюю границу диапазона. Пример:

```
from random import random
print(random())
```

Результат:

```
0.7745718967220968
```

Для генерации вещественного числа в других пределах можно воспользоваться следующим приёмом. Пример:

```
from random import random
print(random() * 10)
```

Результат:

6.369620932985977

При этом генерируется вещественное число от 0 до указанного целого. Само целое число в диапазон не входит.

Чтобы нижняя граница отличалась от нуля, нужно число, генерируемое функцией **random()**, умножить на разность верхней и нижней границ, и прибавить нижнюю. Пример:

```
from random import random
print(random() * (10 - 4) + 4)
```

Результат:

7.913607590966955

В этом примере результат выполнения функции **random()** умножается на 6. В результате выходит число от 0 до 6. Прибавляем 4 и получаем число от 4 до 10.

Основные функции модуля **random** представлены в таблице:

Функции	Назначение
<code>.random()</code>	Возвращает псевдослучайное число от 0.0 до 1.0
<code>uniform(&lt;Начало&gt;, &lt;Конец&gt;)</code>	Возвращает псевдослучайное вещественное число в указанных пределах
<code>randint(&lt;Начало&gt;, &lt;Конец&gt;)</code>	Возвращает псевдослучайное целое число в указанных пределах
<code>choice(&lt;Последовательность&gt;)</code>	Возвращает случайный элемент из любой последовательности (строки, списка, кортежа)
<code>randrange(&lt;Начало&gt;, &lt;Конец&gt;, &lt;Шаг&gt;)</code>	Возвращает случайно выбранное число из последовательности
<code>shuffle(&lt;Список&gt;)</code>	Перемешивает последовательность элементов

В таблице приводится только часть функций. С полным списком можно ознакомиться по [ссылке](#).

## 2.5 Конструкция *yield*

Использование конструкции **yield** тесно связано с понятием генератора. Это итерируемый объект, который можно использовать один раз, так как при использовании генератора значения не хранятся в памяти. Они формируются в процессе обращения к ним, по мере запроса. Пример:

```
generator = (param * param for param in range(5))

for el in generator:
    print(el)
```

Результат:

```
0
1
4
9
16
```

Важно, что пройти по генератору можно только один раз, данные в памяти не хранятся. При повторной попытке возникнет ошибка. Например, можно попытаться получить следующее значение с помощью функции **next()**. Пример:

```
generator = (param * param for param in range(5))

for el in generator:
    print(el)

print(next(generator))
```

Результат:

```
StopIteration
```

Оператор **yield** по назначению похож с оператором **return**, но возвращает генератор вместо значения. Пример:

```
def generator():
    for el in (10, 20, 30):
        yield el

g = generator()
print(g)
```

```
for el in g:
    print(el)
```

Результат:

```
<generator object generator at 0x000000C64E181138>
10
20
30
```

Такой механизм может быть полезен в том случае, когда функция возвращает большой объём данных. Но использовать их нужно только единожды. При вызове функции с оператором **yield** функция не выполняется. Она возвращает объект-генератор, с которым далее можно выполнять нужные действия.

## 2.6 Модуль *functools*

Это специализированный модуль высокого порядка. Его также называют функциями, которые взаимодействуют с другими функциями и возвращают их. Для начала изучим только часть функций модуля **functools**.

### Функция *reduce()*

Применяет указанную функцию к некоторому набору объектов и сводит его к единственному значению. Пример:

```
from functools import reduce

def my_func(prev_el, el):
    # prev_el - предыдущий элемент
    # el - текущий элемент
    return prev_el + el

print(reduce(my_func, [10, 20, 30]))
```

Результат:

```
60
```

### Функция *partial()*

Позволяет создать новую функцию с частичным указанием передаваемых аргументов. Пример:

```
from functools import partial
```

```
def my_func(param_1, param_2):  
    return param_1 ** param_2  
  
new_my_func = partial(my_func, 2)  
print(new_my_func)  
print(new_my_func(4))
```

Результат:

```
16
```

В этом примере создана простая функция, возвращающая результат выполнения операции с параметрами. Далее создаётся новый экземпляр функции **partial**, в которую передаётся экземпляр исходной функции и параметр.

### **2.7 Модуль *itertools***

Содержит итераторы, выполняющие бесконечный процесс итерирования. Это требует условия разрыва итераторов, чтобы избежать бесконечного цикла. Модуль включает широкие возможности, но мы пока рассмотрим только две его функции.

#### **Функция *count()***

Это итератор, возвращающий равномерно распределённые переменные с числа, переданного как стартовый параметр. Допускается указывать значения шага.

Пример:

```
from itertools import count  
  
for el in count(7):  
    if el > 15:  
        break  
    else:  
        print(el)
```

Результат:

```
7  
8  
9  
10  
11  
12  
13
```

```
14
15
```

В примере импортируется функция **count()** из модуля **itertools**, и создаётся цикл **for**. В скрипт добавляется условная проверка, разрывающая цикл при превышении итератором значения 15. Иначе выводится текущее значение итератора. Результат начинается со значения 7, так как оно определено в качестве стартового.

### **Функция cycle()**

Это функция, создающая итератор для формирования бесконечного цикла набора значения.

Пример:

```
from itertools import cycle

c = 0
for el in cycle("ABC"):
    if c > 10:
        break
    print(el)
    c += 1
```

Результат:

```
A
B
C
A
B
C
A
B
C
A
B
```

В этом примере создаётся цикл **for** для бесконечного заикливания букв А, В, С. Но создавать бесконечный цикл — плохая идея, поэтому дополнительно реализован счётчик для разрыва цикла.

Для выполнения операций перемещения по итератору применяется функция **next**.

Пример:

```

from itertools import cycle

progr_lang = ["python", "java", "perl", "javascript"]
iter = cycle(progr_lang)

print(next(iter))
print(next(iter))
print(next(iter))
print(next(iter))
print(next(iter))
print(next(iter))

```

Результат:

```

python
java
perl
javascript
python
java

```

Здесь создаётся список нескольких языков программирования, которые передаются по циклу. Далее новый итератор сохраняется в качестве переменной. Она передаётся следующей функции. При каждом вызове функции возвращается очередное значение в итераторе. Этот итератор бесконечный, поэтому ограничений на число вызовов **next()** нет.

Основные функции модуля **itertools** представлены в таблице:

Функции	Назначение
count (<Начало>, <Шаг>)	Возвращает равномерно распределённые переменные, начиная с числа — стартового параметра. Можно указать параметр шага.
cycle (<Итерируемый объект>)	Итератор, создающий бесконечный цикл поочерёдного вывода неких символов или чисел.
repeat (<Объект>, <Количество повторений>)	Итератор, осуществляющий повторение объекта, переданного в качестве первого параметра в функцию.
combinations (<Объект>, <Количество значений>)	Функция комбинирования элементов последовательности. Принимает два аргумента: объект и количество значений, которые должны присутствовать в каждой комбинации.



<code>combinations_with_replacement</code> (<Объект>, <Количество значений>)	Модифицированный вариант предыдущей функции. Предоставляет программе возможность делать выборку из отдельных элементов с учётом их порядка. Комбинации могут состоять из повторяющихся элементов.
<code>permutations</code> (<Объект>, <Количество значений>)	Сходна с предыдущей функцией, но в текущей не допускается размещение идентичных элементов в одной комбинации.
<code>product</code> (<Массив данных>)	Принимает в качестве параметра массив данных, объединяющий несколько групп значений. Позволяет получить из введённого набора чисел и символов новую совокупность групп во всех возможных вариациях.

В таблице приводится только часть функций. С полным списком можно ознакомиться по [ссылке](#).

## 2.8 Модуль *math*

Предоставляет многочисленные функции для работы с числами:

Функции	Назначение
<code>ceil (N)</code>	Округлить число N до ближайшего большего числа
<code>fabs (N)</code>	Определить модуль числа N
<code>factorial (N)</code>	Найти факториал числа N
<code>floor (N)</code>	Округлить число вниз
<code>fmod (a, b)</code>	Получить остаток от деления a на b
<code>isfinite (N)</code>	Является ли N числом
<code>modf (N)</code>	Определить дробную и целую часть числа N
<code>sqrt (N)</code>	Определить квадратный корень числа N
<code>sin (N)</code>	Определить синус для N-радианов
<code>cos (N)</code>	Определить косинус для N-радианов
<code>tan (N)</code>	Определить тангенс для N-радианов
<code>degrees (N)</code>	Перевести радианы в градусы

radians (N)	Перевести градусы в радианы
-------------	-----------------------------

В таблице приведена только часть функций. С полным списком можно ознакомиться по [ссылке](#). Пример:

```
from math import ceil, fabs, factorial, floor, \
    fmod, isfinite, modf, sqrt, sin, cos, tan, degrees, radians

print(f"ceil() -> {ceil(6.75)}")
print(f"fabs() -> {fabs(-4)}")
print(f"factorial() -> {factorial(5)}")
print(f"floor() -> {floor(4.34)}")
print(f"fmod() -> {fmod(9, 4)}")
print(f"isfinite() -> {isfinite(10)}")
print(f"modf() -> {modf(10.5)}")
print(f"sqrt() -> {sqrt(16)}")
print(f"sin() -> {sin(1.5708)}")
print(f"cos() -> {cos(1.5708)}")
print(f"tan() -> {tan(1.5708)}")
print(f"degrees() -> {degrees(1.5708)}")
print(f"radians() -> {radians(90)}")
```

Результат:

```
ceil() -> 7
fabs() -> 4.0
factorial() -> 120
floor() -> 4
fmod() -> 1.0
isfinite() -> True
modf() -> (0.5, 10.0)
sqrt() -> 4.0
sin() -> 0.9999999999932537
cos() -> -3.673205103346574e-06
tan() -> -272241.80840927624
degrees() -> 90.00021045914971
radians() -> 1.5707963267948966
```

## 2.9 Исключения в python

Исключения (exceptions) - ещё один тип данных в python. Исключения необходимы для того, чтобы сообщать программисту об ошибках. Конструкция try..except имеет следующее формальное определение:

Конструкция try..except имеет следующее формальное определение:

```
try:
    инструкции
except [Тип_исключения]:
    инструкции
```

### Рассмотрим пример деления на 0

```
try:
    a = 100
    b = 0
    c = a / b
except ZeroDivisionError as e:
    print(e)
```

### Результат

```
division by zero
```

Данная инструкция не полная. Полная форма представлена ниже.

```
try:
    выполняется какой-то код
except Exception as e:
    обработка исключения
else:
    код, который будет исполнен в случае, когда не возникает
    исключения
finally:
    код, который гарантированно будет исполнен последним (всегда
    выполняется)
```

Весь основной код, в котором потенциально может возникнуть исключение, помещается после ключевого слова `try`. Если в этом коде генерируется исключение, то работа кода в блоке `try` прерывается, и выполнение переходит в блок `except`.

После ключевого слова `except` опционально можно указать, какое исключение будет обрабатываться (например, `ValueError` или `KeyError`). После слова `except` на следующей строке идут инструкции блока `except`, выполняемые при возникновении исключения.

Рассмотрим обработку исключения на примере преобразовании строки в число:

```
try:
    number = int(input("Введите число: "))
```

```
print("Введенное число:", number)
except:
    print("Преобразование прошло неудачно")
print("Завершение программы")
```

### ***Блок finally***

При обработке исключений также можно использовать необязательный блок `finally`. Отличительной особенностью этого блока является то, что он выполняется вне зависимости, было ли сгенерировано исключение:

```
try:
    number = int(input("Введите число: "))
    print("Введенное число:", number)
except:
    print("Преобразование прошло неудачно")
finally:
    print("Блок try завершил выполнение")
print("Завершение программы")
```

Как правило, блок `finally` применяется для освобождения используемых ресурсов, например, для закрытия файлов.

Список основных исключений и их описания приведены в таблице ниже:

Исключение	Описание
Exception	Любое исключение, не являющееся системным
ZeroDivisionError	Попытка деления на ноль
IndexError	Индекс не входит в диапазон элементов
KeyError	Несуществующий ключ
FileExistsError	Попытка создания существующего файла или директории
FileNotFoundError	Файл или директория не существует
IndentationError	Неправильные отступы
TypeError	Несоответствие объекта и типа данных
ValueError	Некорректное значение аргумента функции



## Сводная таблица «Функции builtins»

Интерпретатор Python предоставляет разработчику ряд встроенных функций:

Функция	Описание
<code>abs()</code>	Возвращает абсолютное значение числа (целого или с плавающей точкой)
<code>all()</code>	Возвращает True, если все элементы итерируемого объекта — истинные
<code>any()</code>	Возвращает True, если какой-либо элемент итерируемого объекта равен True
<code>ascii()</code>	Возвращает строку, содержащую печатаемое представление объекта
<code>bin()</code>	Преобразует целое число в двоичную строку с префиксом 0b
<code>bool()</code>	Возвращает логическое значение (True или False)
<code>breakpoint()</code>	Перемещает в отладчик
<code>bytearray()</code>	Возвращает массив байтов
<code>bytes()</code>	Возвращает объект bytes, представляющий собой неизменяемый набор целых чисел в диапазоне от 0 до 256
<code>callable()</code>	Возвращает True, если аргумент функции поддерживает возможность вызова
<code>chr()</code>	Возвращает символ, соответствующий коду Unicode (целому числу)
<code>classmethod()</code>	Преобразует функцию в метод класса, а не только его экземпляра
<code>compile()</code>	Компилирует исходный код в объект кода, либо в объект абстрактного синтаксического дерева
<code>complex()</code>	Помогает преобразовать в комплексное число
<code>delattr()</code>	Удалить из объекта указанный атрибут
<code>dict()</code>	Вызов конструктора, создающего словарь
<code>dir()</code>	Возвращает список имён, определённых в модуле

<code>divmod()</code>	Возвращает частное-остаток от деления чисел
<code>enumerate()</code>	Возвращает генератор пар счётчик-элемент для элементов указанного набора
<code>eval()</code>	Выполняет разбор и запуск указанного выражения
<code>exec()</code>	Выполняет переданный в функцию код
<code>filter()</code>	Выполняет фильтрацию элементов объекта
<code>float()</code>	Преобразует объект к числу с плавающей точкой
<code>format()</code>	Форматирование переданного объекта
<code>frozenset()</code>	Создание неизменяемого множества
<code>getattr()</code>	Получить значение атрибута объекта
<code>globals()</code>	Получить словарь с глобальной таблицей символов модуля
<code>hasattr()</code>	Возвращает True, если объект содержит указанный атрибут
<code>hash()</code>	Получить хеш объекта
<code>help()</code>	Вызов встроенной справки
<code>hex()</code>	Возвращает целое число в виде строки в шестнадцатеричном формате
<code>id()</code>	Получить идентификатор объекта
<code>input()</code>	Запросить строковый пользовательский ввод
<code>int()</code>	Преобразовать объект в целочисленный формат
<code>isinstance()</code>	Возвращает True, если переданный объект является экземпляром указанного класса
<code>issubclass()</code>	Возвращает True, если указанный класс является подклассом другого класса
<code>iter()</code>	Получить объект итератора
<code>len()</code>	Получить количество элементов в объекте
<code>list()</code>	Создание объекта-списка
<code>locals()</code>	Получить текущую локальную таблицу символов в виде словаря

<code>map()</code>	Применить указанную функцию к каждому элементу коллекции
<code>max()</code>	Возвращает элемент с максимальным значением из набора
<code>memoryview()</code>	Возвращает объект — представление в памяти для переданного аргумента
<code>min()</code>	Возвращает элемент с наименьшим значением из набора
<code>next()</code>	Получить очередной элемент итератора
<code>object()</code>	Создать новый базовый класс
<code>oct()</code>	Возвращает заданное целое число в восьмеричном формате в виде строки
<code>open()</code>	Открыть файл и вернуть его объект
<code>ord()</code>	Вернуть числовой код символа
<code>pow()</code>	Возвести число в степень
<code>print()</code>	Отправить указанный объект текстовым потоком
<code>property()</code>	Вернуть свойство
<code>range()</code>	Определить диапазон с шагом (при необходимости)
<code>repr()</code>	Получить для переданного объекта формальное строковое представление
<code>reversed()</code>	Получить обратный итератор для переданного набора значений
<code>round()</code>	Получить число с плавающей точкой. Округлённое до нужного числа знаков после запятой
<code>set()</code>	Создать изменяемое множество
<code>setattr()</code>	Связать с объектом указанный атрибут
<code>slice()</code>	Выполнить срез в последовательности
<code>sorted()</code>	Вернуть список, состоящий из элементов объекта, поддерживающего итерирование
<code>staticmethod()</code>	Определить указанную функцию в качестве статического метода



<code>str()</code>	Преобразовать объект к строковому типу
<code>sum()</code>	Выполнить суммирование элементов объекта и вернуть результат
<code>super()</code>	Вернуть объект-посредник, перенаправляющий вызовы методов родителю
<code>tuple()</code>	Создать кортеж
<code>type()</code>	Определить тип объекта
<code>vars()</code>	Получить словарь из атрибута <code>__dict__</code> объекта
<code>zip()</code>	Вернуть итератор для кортежей, где каждый <i>i</i> -й кортеж содержит <i>i</i> -й элемент каждой из коллекций