## Создание функции

```
# В круглых скобках называем аргументы
def new_function(arg1, arg2):
    # Прописываем инструкции
    ...
    # Возвращаем результаты
    return result1, result2
```

## Проверка аргументов

```
def get_time(dist, speed):
    # Проверяем аргумент с помощью условия
    if speed <= 0:
        # С помощью raise возвращаем ValueError
        raise ValueError("Speed can't be less than 0")
    # Инструкции для корректных аргументов
    ...
```

## Аргументы по умолчанию

```
# С помощью '=' присваиваем
# значение по умолчанию
def root(value, n=2):
    return value ** (1/n)

# Изменяемые типы данных должны
# создаваться в самом теле функции!
def new_function(in_list=None):
    if in_list is None:
        in_list = list()
    ...
```

## Получение переменного числа аргументов

```
def new_function(*args, **kwargs):
    # В переменной args — кортеж из
    # порядковых аргументов
    # В kwargs — словарь из именованных
    ...
```

## Передача аргументов

```
# Сначала порядковые, затем — именованные аргументы
new_function("Arg1", 24, city="Moscow")
# Распаковка списков и словарей в функцию:
new_list = [1,3,4,5]
how = {'sep': ', ', 'end': '; '}
# С помощью операторов * и **
print(*new_list, **how)
# 1, 3, 4, 5;
```

## Разрешение переменных

### Порядок:

1. Локальные переменные (local)
2. Нелокальные переменные (nonlocal)
3. Глобальные переменные (global)
4. Встроенные переменные (built-in)

```
# Использовать глобальную переменную:
global variable
variable += 1

# Использовать нелокальную переменную:
nonlocal variable
variable += 1
```

## Декораторы

```
def decorator(func):
    # Декорирующая функция
    def decorated(*args, **kwargs):
        # Дополнительные действия "до"
        ...
        res = func(*args, **kwargs)
        # Дополнительные действия "после"
        ...
        return res
    return decorated
# Применяем декоратор через @
@decorator
def my_function(arg): ...
```

## Итераторы

```
my_list = [1, 4, 6]
# Получить итератор из списка:
iter_list = iter(my_list)
# Получить следующий объект:
elem = next(iter_list)
# Перебрать все элементы из итератора
for elem in iter_list:
    ...
# Записать все значения в список:
new_list = list(iter_list)
```

## Генераторы

```
def my_generator(num):
    for i in range(num):
    # Используем yield для выдачи результата
        yield i
    # Выполнение функции замораживается
# Получить экземпляр итератора:
new_gen = my_generator(5)
```

```
# Генератор – тоже итератор:
elem = next(new_gen)
# Сгенерировать всё в список:
l = list(new_gen)
```

## Списочные сокращения

```
# Получить генератор в одну строку:
gen = (x**2 for x in range(10))
# Сохранить результаты в список:
new_list = [x**2 for x in range(10)]
# Или во множество:
new_list = {x**2 for x in range(10)}
# Или в кортеж
new_list = tuple(x**2 for x in range(10))
# Воспользоваться условием:
gen = (x**2 for x in range(10)\
       if x%2 == 0)
```

## Lambda-функции

```
# От одного аргумента:
get_length = lambda x: len(x)
# От двух
mult = lambda x,y : x*y
# От неограниченного числа:
all_args = lambda *args, **kwargs:\
           (args, kwargs)
l = ['dd', 'bbb', 'aaa']
# Сортировка с lambda по длине слова,
# потом – по алфавиту
l.sort(key=lambda x: (len(x), x))
```

## Функции для итераторов

```
# Применить функцию к итератору
map(function_name, my_iterator)
# Отобрать элементы по условной функции:
filter(condition, my_iterator)

# Сгруппировать элементы из итератора:
list1 = [1,3,4,6]
list2 = [5,9,10,13]
for a, b in zip(list1, list2):
   print(a, b)
```