Функции

DRY- Don't Repeat Yourself (Не повторяйся)
Если фрагмент кода повторяется неколько раз, то ему можно дать имя - получается функция

Функция - объект, который выполняет заданный фрагмент программы.
Имя функции - ссылка на этот объект
`func()`
Круглые скобки () - оператор вызова функции
Имя функции часто определяется через глагол: go, show, get, set

Если имя функции не уникально, то вызываться будет последняя объявленная одноименная функция.

(*print - это всего лишь ссылка на объект-функцию, а круглые скобки - оператор вызова этой функции*)

```
f1 = print
f1('Hello') # print и f1 - две ссылки на один объект-функцию.
```

```
def <name_function>([список параметров]):
    оператор1
    оператор2
    return res # явно указываем что должна вернуть функция.
    # функция возвращает ссылку на объект

# тут две пустые строки по PEP8
name_function([список аргументов])
# При вызове функции должно быть ровно столько аргументов, сколько параметров было задано при её определении.
```

```
def get_array(value1 : int, value2 : str) -> list:
""" комментарий к функции в тройных кавычках,
  будет показан в подсказке при наведении курсора на функцию"""
```

```
def get_sqrt(x):
    res = None if x < 0 else x ** 0.5
    return res, x

a, b = get_sqrt(49)
```

- x ссылается на число 49
*- вычисляем 49 ** 0.5*
- переменной res присваивается 7.0 (res ссылается на объект 7.0)
- функция возвращает ссылку на объект, содержащий 7.0
Как только программа встречает return, функция завершает свою работу

```
def get_max2(a, b):
    return a if a > b else b

def get_max3(a, b, c):
    return get_max2(a, get_max2(b, c))
```

```
perim = True
if perim:
    def get_rect(a, b):
        return 2 * (a + b)
else:
    def get_rect(a, b):
        return a * b
```

```
def even(x):    # True, False
    return x % 2 == 0

for i in range(1, 20):
    if even(i):
        print(i)
```

```
[print(i) for i in iter(input, '1') if even(int(i))]
```

```
# Сложная функция
def func1(n):
    def func2(x):
        return x + n
    return func2
new = func1(100)
new(200)  >>> 300
```

Именованные аргументы.

```
def get_v(a, b, c)
    print(f"a = {a}, b = {b}, c = {c}")
    return a * b * c

v = get_v(1, 2, 3) #Позиционная запись аргументов
# т.е. значения определяются порядком записи аргументов:
v = get_v(b=1, a=2, c=3) # именованные аргументы
v = get_v(1, c=2, b=3)
```

Нельзя ставить позиционные аргументы после именованных:

```
v=get_v(a=1, 2, 3)
```

Параметры со значениями по умолчанию (формальные параметры).

```
def get_v(a, b, c, verbose=True):
    if verbose:
        print(. . .)
    return . . .

# a, b, c - фактические параметры
# verbose - формальный параметр
v = get_v(1, 2, 3) # При вызове функции формальный параметр указывать необязательно
v = get_v(1, 2, 3, False)
```

Изменяемый объект в параметрах по умолчанию

```
def add_value(value, lst=[]):
    lst.append(value)
    return lst

l = add_value(1)
l = add_value(2) # lst ссылается на тот же смый список!!
print(lst) >>> [1, 2]
```

lst=[] - это изменяемый тип данных

Значения параметров по умолчанию создаются один раз - при определении функции, а не при вызове!

То есть для вызова используется уже определённое значение.

У функций параметр __defaults__ хранит значения по умолчанию.

Если внутри функции изменится изменяемый объект, то в func.__defaults__ он тоже изменится, что отразится при следующем вызове функции.

Изменить такое поведение:

```
l = add_value(2, [])
```

Или указать в качестве параметров неизменяемые типы данных:

```
def add_value(value, lst=None):
    if lst is None:
        lst = []
    lst.append(value)
    return lst
# При этом:
ls = add_value(1)
ls = add_value(2) >> [2]
ls = add_value(3, ls) >> [2, 3]
```

Функции с произвольным числом аргументов

Функция с произвольным числом фактических параметров:

```
def os_path(*args): # *args - имя переменной, кот.ссылается на список фактических параметров
    path = "\\".join(args)
    return path

p = os_path(str1, str2, str3)
# Но вызовет ошибку:
p = os_path(str1, str2, str3, sep='/')
# sep - именованный аргумент, но переменная sep не определена.
# Надо прописать формальный параметр sep при определении функции:
def os_path(*args, sep='\\'):
    path = sep.join(args)
    return path
```

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

my_func()
my_func(1, 2, 3)
my_func('a', 'b')
```

```
>>>
<class 'tuple'>
()
<class 'tuple'>
(1, 2, 3)
<class 'tuple'>
('a', 'b')
```

Звездочка в определении функции означает, что переменная (параметр) args получит в виде кортежа все аргументы, переданные в функцию при ее вызове от текущей позиции и до конца.

При описании функции можно использовать **только один** параметр помеченный звездочкой, причем располагаться он должен в конце списка параметров, иначе последующим параметрам не достанется значений.

```
def my_func(num, *args):
    print(args)
    print(num)

my_func(17, 'Python', 2, 'C#')
связывает с переменной num значение 17, а с переменной args значение кортежа ('Python', 2, 'C#'):
('Python', 2, 'C#')
17
```

параметр *args нормально переживает и отсутствие аргументов,
позиционные параметры всегда обязательны.

```
my_func(17)
>>>
()
17
```

Передача аргументов в форме списка и кортежа

Встроенная функция sum() принимает коллекцию чисел (список, кортеж и т.д.)

```
sum1 = sum([1, 2, 3, 4])       >>> 10
sum2 = sum((10, 20, 30, 40))   >>> 100
Однако функция sum() не может принимать переменное количество аргументов.
sum1 = sum(1, 2, 3, 4)      >>> TypeError: sum expected at most 2 arguments, got 4

def my_sum(*args):
    return sum(args)    # args – это кортеж

print(my_sum())   >>> 0
print(my_sum(1))   >>> 1
print(my_sum(1, 2, 3))   >>> 6
```

Можно вызывать my_sum(), передавая ей списки или кортежи, предварительно распаковав их.

```
print(my_sum(*[1, 2, 3, 4, 5]))   #  распаковка списка >>> 15
print(my_sum(1, 2, *[3, 4, 5], *(7, 8, 9), 10))   >>> 49
```

Произвольное число именованных аргументов.

Именованные аргументы получаются в виде словаря, что позволяет сохранить имена аргументов в ключах.

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

my_func()  >>> <class 'dict'>   >>> {}
my_func(name='Timur', job='Teacher')   >>> <class 'dict'>   >>> {'name': 'Timur', 'job': 'Teacher'}
```

```
def os_path(*args, ** kwargs):
    print(kwargs)

p = os_path(str1, str2, str3, sep="\\", tim=True)
```

kwargs будет предствалять собой словарь: {'sep': '\\', 'trim': True}

Параметр **kwargs пишется в самом конце, после последнего аргумента со значением по умолчанию.

```
def my_func(a, b, *args, name='Gvido', age=17, **kwargs):
    print(a, b)
    print(args)
    print(name, age)
    print(kwargs)

my_func(1, 2, 3, 4, name='Timur', age=28, job='Teacher', language='Python')
>>>
1 2
(3, 4)
Timur 28
{'job': 'Teacher', 'language': 'Python'}
```

Прописанные фактические параметры - перед *args
Прописанные именованные параметры - перед **kwargs

```
def os_path(disk, *args, sep='//', ** kwargs):
```

Применять именованный аргумент из коллекции **kwargs при вызове функции можно, только если он прописан:

```
if 'trim' in kwargs and kwargs['trim']:
    args = [x.strip() for x in args]
```

(учет в функци параметра trim)

Передача именованных аргументов в форме словаря

именованные аргументы можно передавать в функцию "пачкой" в виде словаря, попутно распаковав его:

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

info = {'name':'Timur', 'age':'28', 'job':'teacher'}
my_func(**info)
>>>
<class 'dict'>
{'name': 'Timur', 'age': '28', 'job': 'teacher'}
```

```
def print_info(name, surname, age, city, *children, **additional_info):
    print('Имя:', name)
    print('Фамилия:', surname)
    print('Возраст:', age)
    print('Город проживания:', city)
    if len(children) > 0:
        print('Дети:', ', '.join(children))
    if len(additional_info) > 0:
        print(additional_info)

children = ['Бодхи Рансом Грин', 'Ноа Шэннон Грин', 'Джорни Ривер Грин']
additional_info = {'height':163, 'job':'actress'}
print_info('Меган', 'Фокс', 34, 'Ок-Ридж', *children, **additional_info)
>>>
Имя: Меган
Фамилия: Фокс
Возраст: 34
Город проживания: Ок-Ридж
Дети: Бодхи Рансом Грин, Ноа Шэннон Грин, Джорни Ривер Грин
{'height': 163, 'job': 'actress'}
```

При подстановке аргументов "разворачивающиеся" наборы аргументов вроде *positional и **named можно указывать вперемешку с аргументами соответствующего типа: *positional с позиционными, а **named – с именованными.
Но все именованные аргументы должны идти после всех позиционных!

Keyword-only аргументы (С Python 3)
Аргументы, которые нельзя передать в функцию в виде позиционных.
т.е. вызвать функцию можно, только передав эти аргументы по именам.

```
def make_circle(x, y, radius, *, line_width=1, fill=True):
Здесь * отделяет обычные аргументы от строго именованных.

make_circle(10, 20, 5)              # x=10, y=20, radius=5,  line_width=1, fill=True
make_circle(x=10, y=20, radius=17, line_width=3)   # x=10, y=20, radius=17, line_width=3, fill=True
```

Аргументы x, y и radius могут быть переданы в качестве как позиционных, так и именованных аргументов.
Аргументы line_width и fill могут быть переданы **только** как именованные аргументы.

Приведенный ниже код:

```
make_circle(10, 20, 15, 20)
make_circle(x=10, y=20, 15, True)
make_circle(10, 20, 10, 2, False)
```

приводит к возникновению ошибок.

Объявить функцию, у которой только строго именованные аргументы
(поставить звёздочку в самом начале перечня аргументов)

```
def make_circle(*, x, y, radius, line_width=1, fill=True):
```

Теперь для вызова функции make_circle() нам нужно передать значения всех аргументов явно через их имя:

```
make_circle(x=10, y=20, radius=15)                              # line_width=1, fill=True
make_circle(x=10, y=20, radius=15, line_width=4, fill=False)
```

Такой разделитель можно использовать только один раз в определении функции. Его нельзя применять в функциях с неограниченным количеством позиционных аргументов *args.

Можно использовать одновременно *args и **kwargs в одной строке для вызова функции.
Но **порядок имеет значение**. *args должны предшествовать аргументам по умолчанию и **kwargs

Области видимости переменных. Глобальные и локальные переменные.

1. Пусть мы создали программу и сохранили в файле myprog.py
2. В Python автомтически создаётся пространство имен с именем myprog (изначально оно пустое)
3. Как только в программе создаём переменные, они появляются в пространстве имен файла. (myprog: N, M, WIDTH)
4. Если мы прописываем функцию my_func, то она создаёт свое пространство имен

```
def my_func(lst):
    for x in lst:
        n = x + 1
        print(n)
```

my_func: lst, n, x (цикл for не образует своего пространства имен)

Переменные внутри функции - локальные переменные. Они существуют только внутри функции.
За пределами функции мы не можем к ним обращаться.

Внутри функции при обращении к переменной:
1) Сначала ищется эта переменная внутри текущей локальной области видимости. Если она находится, то используется локальная переменная.
2) Если переменная не найдена в текущей локальной области, то поиск переходит на более высокий уровень.

Глобальные переменные.
- Объявлены вне функций. Доступны в любом месте программы после их объявления.
- Чтобы функция могла изменить значение глобальной переменной, необходимо внутри функции объявить эту переменную как глобальную с помощью ключ.слова global

```
N = 100
def my_func(lst):
    global N
    # global можно записывать только для тех переменных, которые отсутствуют в текущей локальной области
    # то есть нельзя обьявить N=20 до global
    N = 20 # создаётся новая глобальная переменная с именем N
    for x in lst:
        n = x + 1 + N
        print(n)
print(N) >>> 20
```

Локальные переменные.
Созданы внутри функции и доступны только внутри неё.
К локальным переменным обратиться вне тела функции нельзя.

```
x = 0
def outher():
    x = 1
    def inner():
        x = 2
        print("inner: ", x)
    inner()
    print("outher: ", x)
outher()
print("global: ", x)

>>> inner: 2
outher: 1
global: 0
```

Nonlocal. Если хотим в inner использовать переменную из outher:

```
def inner():
    nonlocal x
    x = 2 # в локальной области inner переменная x создаваться не будет
    # а будет браться из outher
    print("inner: ", x)

>>> inner: 2
>>> outher: 2
>>> global: 0
```

nonlocal x можно использовать только в том пространстве имен, которое ссылается на другое локальное пространство имен.

Константа - глобальная переменная, которая за весь код ни разу не меняет своего значения.
Задаётся в верхнем регистре.

Медленный Алгоритм Евклида для поиска НОД.

```
a = 18, b = 24
b = b - a = 24 - 18 = 6
a = 18, b = 6
a = a - b = 18-6 = 12
a = 12, b = 6
a = a - b = 6
a = b = 6
НОД = 6
т. е. пока a != b, находим max(a, b) и большее заменяем разностью a и b.
```

Быстрый Алгоритм Евклида для поиска НОД.

```
a = 18, b = 24
b = 24 % 18 = 6
a = 18 % 6 = 0
НОД(18, 24) = 6

import time
def get_nod(a, b):
    """
    Вычисляем НОД для натуральных чисел a и b
    по быстрому алгоритму Евклида
    param a, b: натуральные числа
    return: НОД(a, b)
    """
    if a < b:
        a, b = b, a
    while b != 0:
        a, b = b, a % b # (a % b) заведомо меньше чем b
        return a

# создадим вспомогательную тестирующую функцию.
def test_nod(func):
    # ---- test 1 ----
    a, b = 28, 35
    res = func(a, b)
    print('#test1 - ok') if res == 7 else print('#test1 - fail')

    # ---- test 2 ----
    a, b = 100, 1
    res = func(a, b)
    print('#test2 - ok') if res == 1 else print('#test2 - fail')

    # ---- test 3 ----
    # с таймером
    a, b = 2, 10000000
    st = time.time() # засечем время начала вычисления
    res = func(a, b)
    et = time.time() # время окончания вычисления
    dt = et - st
    print('#test3 - ok') if res == 2 and dt < 1 else print('#test3 - fail')

test_nod(get_nod)
```

С рекурсией:

```
def get_nod(a, b):
    return get_nod(b, a % b) if b else a
```

*Операторы * и ** упаковки и распаковки*

*args - позволяет упаковывать обычные позиционные аргументы в кортеж
**kwargs - позволяет упаковывать именованные аргументы в словарь

Упаковка данных
Эти же самые операторы (*, **) можно использовать и при работе с разными коллекциями

```
x, *y = (1, 2, 3, 4) # x == 1, y == [2, 3, 4] - упаковали вторую часть списка в список y
*x, y = (1, 2, 3, 4) # x = [1, 2, 3], y = 4
```

Упаковывать так можно не только из кортежей, но и из любых итерируемых объектов

Распаковка.

```
a = [1, 2, 3]
(a, )  >> ([1, 2, 3], )
(*a, ) >> (1, 2, 3)
```

```
d = (-5, 5)
range(*d)  # идентично range(-5, 5)
list(range(*d))  >> [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
```

Можно распаковать в список произвольный итерируемый объект:

```
[*range(*d)]  >> [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
[*range(*d), *(True, False), *a]  >> [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, True, False, 1, 2, 3]
```

Словари

```
d = {0: "безнадежно", 1: 'убого', 2: 'неуд', 3: 'удовл', 4: 'хорошо', 5: 'отл'}
{*d}  >> {0, 1, 2, 3, 4} множество ключей словаря (распаковал d как итерируемый объект)
{*d.values()}  >> {'безнадежно', 'убого', 'неуд', 'удовл', 'хорошо', 'отл'} - множество из значений словаря
{*d.items()}  >> {(0, "безнадежно"), (1, 'убого'), (2, 'неуд'), (3, 'удовл'), (4, 'хорошо'), (5, 'отл')}
```

```
{**d} >> {0: "безнадежно", 1: 'убого', 2: 'неуд', 3: 'удовл', 4: 'хорошо', 5: 'отл'}
# распакует словарь в словарь.
```

Но

```
d2 = {6: 'превосходно', 7: 'элитарно', 8: 'божественно'}
```

d и d2 можно объединить в один словарь таким способом:

```
{**d, **d2}
>> {0: "безнадежно", 1: 'убого', 2: 'неуд', 3: 'удовл', 4: 'хорошо', 5: 'отл', 6: 'превосходно', 7: 'элитарно', 8:
'божественно'}
```

Анонимные функции (λ-функции)

Анонимные функции (λ-функции).

Содержат только одно выражение.

```
func = lambda param1, param2: команда
func = lamda x, y: x+y
func(1, 2)  >>> 3

(lambda x, y: x+y)(1, 2)  >>> 3
```

- Только одна комнда
- только одна строчка
- нельзя использовать оператор присваивания (и вообще операторы) (но можно: lambda a: a + 1)
- может быть вызвана немедленно

```
f1 = lambda x, y, z: x + y + z
print(f1(5, 10, 30))  >>> 45
```

```
a = [4, 5, lambda: print('lambda'), 7, 8]
a[2] >>> ссылка на лямбда-функцию
a[2]() >>> lambda # вызвали функцию.
```

Однократное использование функции

```
points = [(1, -1), (2, 3), (-10, 15), (10, 9), (7, 18), (1, 5), (2, -4)]

print(sorted(points, key=lambda point: point[1]))              # сортируем по второму значению кортежа
print(sorted(points, key=lambda point: point[0] + point[1]))   # сортируем по сумме элементов кортежа
```

Передача анонимных функций в качестве аргументов другим функциям

```
numbers = [1, 2, 3, 4, 5, 6]
new_numbers1 = list(map(lambda x: x+1, numbers))      #  увеличиваем на 1
new_numbers2 = list(map(lambda x: x*2, numbers))      #  удваиваем
new_numbers3 = list(map(lambda x: x**2, numbers))     #  возводим в квадрат
```

```
strings = ['a', 'b', 'c', 'd', 'e']
numbers = [3, 2, 1, 4, 5]

new_strings = list(map(lambda x, y: x*y, strings, numbers))
print(new_strings)
>>>  ['aaa', 'bb', 'c', 'dddd', 'eeeee']
```

```
numbers = [-1, 2, -3, 4, 0, -20, 10, 30, -40, 50, 100, 90]

positive_numbers = list(filter(lambda x: x > 0, numbers))      #  положительные числа
large_numbers = list(filter(lambda x: x > 50, numbers))        #  числа, большие 50
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))     #  четные числа
```

```
words = ['python', 'stepik', 'beegeek', 'iq-option']

new_words1 = list(filter(lambda w: len(w) > 6, words))    #  слова длиною больше 6 символов
new_words2 = list(filter(lambda w: 'e' in w, words))      #  слова содержащие букву e
```

```
from functools import reduce

words = ['python', 'stepik', 'beegeek', 'iq-option']
numbers = [1, 2, 3, 4, 5, 6]

summa = reduce(lambda x, y: x + y, numbers, 0)
product = reduce(lambda x, y: x * y, numbers, 1)
sentence = reduce(lambda x, y: x + ' loves ' + y, words, 'Everyone')

print(summa)      >>> 21
print(product)    >>> 720
print(sentence)   >>> Everyone loves python loves stepik loves beegeek loves iq-option
```

```
lst = [5, 3, 0, -6, 8, 10, 1]
def get_filter(a, filter=None): # выбирает значения из списка a по определенному критерию
    if filter is None:
        return a
    res = []
    for x in a:
        if filter(x): # Если функция filter для текущего значения списка возвращает True
            res.append(x)
    return res

r = get_filter(lst)  # получим lst без изменений
r = get filter(lst, lambda x: x % 2 == 0)  #останутся только четные значения в списке.
```

Возвращение функции в качестве результата другой функции

Приведенный ниже код по значениям a,b,c строит и возвращает квадратный трехчлен:

```
def generator_square_polynom(a, b, c):
    def square_polynom(x):
        return a*x**2 + b*x + c
    return square_polynom
```

Такой код можно переписать так:

```
def generator_square_polynom(a, b, c):
    return lambda x: a*x**2 + b*x + c
```

Анонимные функции являются **замыканиями**: возвращаемая функция запоминает значения переменных a, b, c из внешнего окружения.

Условный оператор в теле анонимной функции

Общий вид тернарного условного оператора в теле анонимной функции:

```
значение1 if условие else значение2
```

Если условие истинно, возвращается значение1 , если нет – значение2.

```
numbers = [-2, 0, 1, 2, 17, 4, 5, 6]
result = list(map(lambda x: 'even' if x % 2 == 0 else 'odd', numbers))
print(result)
>>> ['even', 'even', 'odd', 'even', 'odd', 'even', 'odd', 'even']
```

Передача аргументов в анонимную функцию

анонимные функции поддерживают все способы передачи аргументов:

- позиционные аргументы;
- именованные аргументы;
- переменный список позиционных аргументов (*args);
- переменный список именованных аргументов (**kwargs);
- обязательные аргументы (*).

```
f1 = lambda x, y, z: x + y + z
f2 = lambda x, y, z=3: x + y + z

print(f1(1, 2, 3))                >>> 6
print(f2(1, 2))                   >>> 6
print(f2(1, y=2))                 >>> 6
```

```
f3 = lambda *args: sum(args)
f4 = lambda **kwargs: sum(kwargs.values())
f5 = lambda x, *, y=0, z=0: x + y + z

print(f3(1, 2, 3, 4, 5))          >>> 15
print(f4(one=1, two=2, three=3))  >>> 6
print(f5(1))                      >>> 1
print(f5(1, y=2, z=3))            >>> 6
```

Лямбда-функции являются выражениями. После определения лямбда-функции ее можно сразу же вызвать.

```
print((lambda x, y: x + y)(5, 10))     # 5 + 10      >>> 15
print(1 + (lambda x: x*5)(10) + 2)     # 1 + 50 + 2  >>> 53
```

В лямбда исчислении, часто применяемом в разработке языков программирования, все функции – анонимные, поэтому анонимные функции во многих языках тоже называют "лямбдами" или "лямбда-функциями".

В Python анонимные функции – лишь сокращенная запись функции

```
f = lambda x: x + 1
print(type(f))   >>> <class 'function'>
```

То есть, анонимные функции имеют такой же тип, как и обычные функции.

Анонимные функции **очень часто** используются вместе со встроенными функциями map(), filter(), reduce(), sorted(), max(), min() и т.д.

Рекурсивные функции

Функции, которые вызывают сами себя.

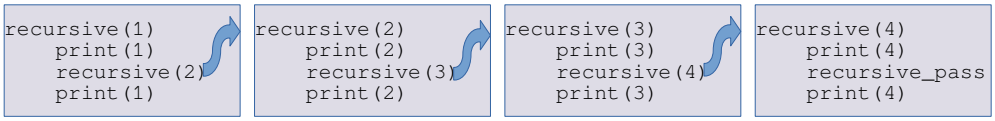
```
def recurcive(value):
    print(value)
    recurcive(value + 1)

recurcive(1)
```

Функция выполнится до 996, потом остановится с ошибкой:
RecursionError ()достигнута максимальная глубина рекурсии.

Когда вызывается какая-либо функция, то она автоматически помещается в стек вызова функций
Стек вызова функций конечен. То есть должно быть условие останова

```
def recurcive(value):
    print(value, end=' ')
    if value < 4:
        recurcive(value + 1)
    print(value, end=' ')
>>> 1 2 3 4 4 3 2 1
```



Классика рекурсии.
n! = 1 * 2 * 3 * ... * n
n! = (n-1)! * n
fact(n) = n * fact(n-1)

```
def fact(n):
    if n <= 0:
        return 1
    else:
        return n * fact(n-1)
```

Обход каталогов и файлов

```
# Создадим словарь с каталогами и файлами:
F = {
    'c:': {
          'Python39': ['python.exe', 'python.ini'],
          'Program files': {
                           'Java' : ['readme.txt', 'welcome.html' ...]
                           }
          }
    }
```

Для обхода словаря сделаем функцию.

```
def get_files(path, depth=0):
    for f in path:
        print(' ' * depth, f)
        if type(path[f]) == dict:
            get_files(path[f], depth + 1)
        else:
            print(' ' * (depth + 1), ' '.join(path[f]))
```

Метод слияния списков.
a = [1, 4, 10, 11]
b = [2, 3, 3, 4, 8]
Соединить с сортировкой два отсортированных списка: [1, 2, 3, 3, 4, 4, 8, 10, 11]

res = []
1. Если a[0] < b[0]: res.append(a[0]), указатель в списке a перемещаем на след.элемент
2. Сравниваем a[1] и b[0]
...
если один из списков завершился, то просто добавляем хвост второго.

Сортировка слияния.
1. [9, 5, -3, 4, 7, 8, -8]
2. разбиваем на две части: [9, 5, -3], [4, 7, 8, -8]
3. Каждую часть снова разбиваем: [9], [5, -3], [4, 7], [8, -8]
4. Собираем массив обратно методом слияния:
    4.1. [9], [5, -3], [4, 7], [8, -8]
    4.2. [9], [-3, 5], [4, 7], [-8, 8]
    4.3. [-3, 5, 9], [-8, 4, 7, 8]
    4.4. [-8, -3, 4, 5, 7, 8, 9]
$O(N) = N * \log_2(N)$ --> $7 * \log_2 7 < 21$
(Простейшие алгоритмы: $O(N^2)$ --> 7 ** 2 = 49)

Замыкания в Python.

Внутри функции можно объявить еще одну функцию:

```
def say_name(name):
    def say_goodbye():
        print(f'...{name}...')
    say_goodbye()

say_name('Sergey')  >>> Don't say me goodbye, Sergey!
```

Замыкание: Внутри внешней функции можно создать вложенную функцию, которую можно вызвать из глобального пространства.
Но чтобы это сделать, нужно чтобы внешняя функция возвращала ссылку на вложенную (ее имя).
Тогда мы сможем ее вызвать по этой ссылке, просто дописав оператор вызова функций '()'

```
def say_name(name):
    def say_goodbye():
        print(f'...{name}...')
    return say_goodbye # возвращается ссылка на внутреннюю функцию.

say_name('Sergey') # При таком запуске внутренняя функция не вызывается
# только возвращается ссылку на внутреннюю функцию)
f = say_name('Sergey') # Сохранили ссылку на внутреннюю функцию
f() # вызвали внутреннюю функцию

say_name('Sergey')() # тоже вызов внутренней функции
```

Внутреннее окружение вместе с именами переменных say_goodbye продолжает существовать, пока существует ссылка f на него.
А вместе с ним продолжают существовать все внешние окружения, которые с ним связаны (в т.ч. переменная name == 'Sergey')

Замыкание:
Т.е. переменная f - принадлежит глобальной области видимости программы my_prog.
Она ссылается на локальное окружение функции say_goodbye
Окружение say_goodbye ссылается на внешнее окружение функции say_name
say_name - на глобальное окружение my_prog

При каждом вызове say_name() будет создаваться свое независимое локальное окружение.

```
f2 = say_name('Python') # переменная name будет ссылаться на новую строчку 'Python'
f2()
```

Функция-счетчик, которая при каждом новом запуске увеличивает свое значение на 1.

```
def counter(start=0):
    def step():
        nonlocal start # чтобы использовать переменную start из внешнего окружения,
                       # а не создавать каждый раз новую
        start += 1
        return start
    return step

c1 = counter(10)
c2 = counter() # можем создавать много разных счетчиков и использовать их независимо друг от друга
print(c1(), c2()) # 11 1
print(c1(), c2()) # 12 2
print(c1(), c2()) # 13 3
```

Удаление ненужных символов в начале и в конце строки.

```
def strip_string(strip_chars=' '):
    def do_strip(string):
        return string.strip(strip_chars)
    return do_strip

strip1 = strip_string() # задаём внешний параметр – удаляемые символы
strip1 = strip_string(' !&,.;')
print(strip1(' hello python!.. ')) >>> hello python!.. # задали внутренний аргумент – строку
print(strip2(' hello python!.. ')) # hello python
```

Если есть глобальная ссылка f на внутреннее локальное окружение, то это окружение продолжает существовать. Оно не удаляется автоматическим сборщиком мусора.
А вместе с этим локльным окружением продолжают существовать и все остальные внешние окружения, которые с ним связаны (у каждого локального окружения есть скрытая ссылка на внешнее окружение).

Декораторы функций

```
def func_decorator(func): # func - ссылка на некоторую функцию
    def wrapper(title):
        [некие команды] # расширяем функционал сторонней функции func
        func(title)
        [некие команды]
    return wrapper

def some_func(title)  # Объявляем функцию, которую хотим декорировать.
    print(f'title = {title}')

some_func() # просто напечатает текст

f = func_decorator(some_func)
f() # не просто выполнили функцию some_func, а дополнили её командами из wrapper
```

Но,чтобы не плодить переменные, при декорировании обычно создают ссылку на то же самое имя:

```
some_func = func_decorator(some_func)
```

В итоге функция some_func меняет свою работу.

```
some_func('Hello Python') # не только выполнит изначальную функцию some_func, но и дополнит ее командами из wrapper()
```

Но если изменить аргументы some_func, то у wrapper тоже нужно будет изменить аргументы, чтобы эта конструкция не перестала работать.

Чтобы избежать зависимости от аргументов декорируемой функции:

```
def func_decorator(func): # func - ссылка на некоторую функцию
    def wrapper(*args, **kwargs):
        print('что-то перед функцией')
        res = func(*args, **kwargs)
        print('что-то после функции')
        return res
    return wrapper

def some_func(title, tag): # теперь смело добавляем еще один аргумент.
    print(f:'title = {title}, tag = {tag}')
    return f'<{tag}>{title}</{tag}>'

some_func = func_decorator(some_func)
some_func("Python навсегда", "h1")
```

```
# Декоратор, который будет измерять скорость работы любой функции:
import time
def test_time(func): # func - ссылка на некоторую функцию
    def wrapper(*args, **kwargs):
        st = time.time()
        res = func(*args, **kwargs)
        dt = time.time() - st
        print(f'время работы: {dt} сек')
        return res
    return wrapper

def get_nod(a, b):
    . . .

get_nod = test_time(get_nod)
get_fast_nod = test_time(get_fast_nod) # декорируем еще одну функцию тем же декоратором
res1 = get_nod(2, 10000)
res1 = get_fast_nod(2, 10000)
print(res1, res2)
```

```
# Упрощенная запись декорирования функции:
@test_time
def get_nod(a, b):
   . . .
```

Передача аргументов декораторам.

```
import math
# Создаем декоратор для вычисления производной функции
def df_decorator(dx=0.01): # задаём внешнюю функцию для того,
                           # чтобы декоратор умел принимать  дополнительные параметры
    def func_decorator(func):
        def wrapper(x, *args, **kwargs):
            dx = 0.0001
            res = (func(x + dx, *args, **kwargs) - func(x, *args, **kwargs)) / dx
            return res
        return wrapper
    return func_decorator

@df_decorator(dx=0.01) # теперь декоратор умеет принимать доп.аргумент
def sin_df(x):
    return math.sin(x)

df = sin_df(math.pi/3)
print(df)
```

```
# развернуто:
# sin_df = df_decorator(dx=0.001)(sin_df)
```

Потеря имени и описания декорируемой функции.

```
# у каждой функции есть свое имя, доступное с помощью спец.переменной __name__
def sin_df(x):
    """Функция для вычисления производной синуса"""
    return math.sin(x)

print(sin_df.__name__) # sin_df
print(sin_df.__doc__) # print(sin_df.__doc__) #  Функция для вычисления производной синуса
```

```
# Если мы декорируем функцию, то она теряет свое имя и описание.
@df_decorator(dx=0.01)
def sin_df(x):
    """Функция для вычисления производной синуса"""
    return math.sin(x)

print(sin_df.__name__) # wrapper
print(sin_df.__doc__) # None
```

```
Если мы хотим сохранить имя функции:
def df_decorator(dx=0.01):
    def func_decorator(func):
        def wrapper(x, *args, **kwargs):
            dx = 0.0001
            res = (func(x + dx, *args, **kwargs) - func(x, *args, **kwargs)) / dx
            return res
        wrapper.__name__ = func.__name__ # явно прописываем сохранение имени функции
        wrapper.__doc__ = func.__doc__ # явно прописываем сохранение описания функции
        return wrapper
    return func_decorator
```

```
Это стандартная процедура, и ее можно оформить проще:
from functools import wraps  # встроенный декоратор
def df_decorator(dx=0.01):
    def func_decorator(func):
        @wraps(func)
        def wrapper(x, *args, **kwargs):
            dx = 0.0001
            res = (func(x + dx, *args, **kwargs) - func(x, *args, **kwargs)) / dx
            return res
        return wrapper
    return func_decorator
```

Импорт стандартных модулей

```
import math as mt # можно использовать псевдонимы
import time
import pprint

math.ceil(1.8)
print(math.pi)
```

```
# в pycharm Ctrl + левой кнопкой мыши на модуль, то откроется содержимое импортируемого модуля.
Модуль - это текстовый файл, обычная программа.
```

Что делает import?

```
print(locals()) # отобразит словарь, который содержит все локальные переменные
>>> {'__name__': '__main__', '__doc__': None, '__pakage__': None . . . }
```

В пределах нашей программы эти переменные можно воспринимать как глобальные

```
pprint.pprint(locals())
# Отобразит переменные в удобном виде:
   {'__annotations__': {},
   '__builtins__': <module 'builtins' (built-in)>,
   '__cached__': None,
   '__doc__': None,
   '__file__': 'D:\\Python\\Projects\\stepik\\ex1.py',
   '__loader__': <_frozen_importlib_external.SourceFile>,
   '__name__': '__main__',
   '__package__': None,
   '__spec__': None,
   'math': <module 'math' (built-in)>,
   'pprint': <module 'print' from 'C:\\Python39\\lib\\. . .'>,
   'time': <module 'time' (built-in)>}
```

последние три строчки - **ссылки** на пространство имен, которые содержатся в этих модулях

все переменные, функции модуля доступны через переменную с именем этого модуля

т.е. import создает пространство имен с соответствующими именами через которое мы можем обращаться к глобальным определениям в соответствующих модулях

Если мы не хотим импортировать модуль целиком:

```
from math import ceil, pi
# ceil и pi импортировались в наше пространство имен.  поэтому обращаться к ним можно напрямую:
a = ceil(1.8) # 2 - будет также верно как и a = math.ceil(1.8)
```

```
from math import ceil as cl, pi # можно присвоить алиас для отдельной импортируемой функци или перем.
```

```
from math import *
# импортировали напрямую все содержимое библиотеки math (в наше глобальное окружение попало все содержимое math, которое можно вызывать напрямую)
Это может привести к конфликту имен.
(В разных импортируемых библиотеках могут оказаться объекты с одинаковыми именами)
```

Таких импортов лучше избегать

```
import pprint, time, math # можно, но не рекомендуется по стандарту PEP8
```

Некоторые стандартные модули Python	
audioop	библиотека для работы со звуком. Содержит функции для обработки записей 8, 16, 24 или 32 бит. Поддерживает различные кодировки, конвертацию форматов, настройки битов, и пр.
base64	позволяет шифровать бинарные данные в читаемые ASCII символы, а затем обратно раскодировать в бинарные данные. Можно кодировать последовательности байт и обычные строки.
calendar	позволяет работать с календарем. Можно выводить календарь, так же как это делает утилита cal в Linux. По умолчанию, началом недели считается понедельник, но можно поменять настройки на воскресение. Вы можете просматривать информацию о днях недели, месяца, года, выводить списки и многое другое.
cgi	позволяет интерпретатору обрабатывать скрипты по запросу веб-сервера и возвращать ему же результат обработки. Можно получать переменные, переданные браузером с помощью GET или POST, а также влиять на отправляемые данные.

configparser	позволяет разбирать содержимое простейших конфигурационных файлов формата ini. Такие файлы очень часто используются в Windows. Вы можете не только читать содержимое файлов, но и изменять его.
csv	- модуль, позволяющий работать с форматом файлов csv (Comma Separated Values). Модуль имеет два класса, reader и writer, которые позволяют читать и записывать данные в формат csv.
curses	движок псевдографического интерфейса. Позволяет довольно просто реализовать простой графический интерфейс в терминале.
datetime	реализует набор методов для получения информации, преобразования, изменения даты и времени. Можно преобразовать дату в строку или прочитать ее из строк различных форматов. Также можно выполнять арифметические операции с датами и временем.
decimal	содержит функции для быстрого преобразования чисел с плавающей точкой. Также содержит несколько дополнительных возможностей для встроенного типа float.
difflib	содержит набор функций для сравнения различных последовательностей. Можно сравнивать файлы, строки, различную информацию в HTML и многое другое.
email	Содержит функции для разбора структуры email сообщений, проверки списка почты, преобразования и много другого.
gettext	реализует функции локализации и интернационализации L10N для ваших программ на Python. Поддерживается стандартное API GNU gettext так и свое собственное API на основе классов. Все модули пишутся на вашем нативном языке, а затем к программе прикрепляется каталог для перевода на другие языки.
gzip, zlib	для работы со сжатыми данными. Можно распаковывать и упаковывать файлы, работать со строками, использовать пароли.
hashlib	предоставляет интерфейс для получения различных хэшей для данных. Поддерживаются алгоритмы: SHA1, SHA224, SHA256, SHA384, и SHA512, MD5.
html, http	Модуль http позволяет работать с интернет ресурсами по протоколу HTTP, отправлять запросы GET/POST, принимать запросы, обрабатывать Cookie и фактически реализовать свой клиент или сервер на Python. Библиотека html, в свою очередь, позволяет выполнять разбор html страниц.
io	содержит основные функции для работы с потоками ввода/вывода. Поддерживаются различные виды потоков, текстовые, бинарные и RAW потоки.
itertools	средства для организации итераций, похожих на Haskell, APL и SML. Модуль использует эффективные методы работы с памятью.
json	работа с форматом передачи данных json. Есть функции для разбора формата и для создания объектов для отправки.
logging	- модуль для логирования в программах Python. Библиотека реализует удобную систему логирования, которая используется в стандартных модулях. Ее преимущество в том, что вы можете отключить ведение лога в любой момент одной строчкой или изменить его подробность.
math	стандартные функции для работы с математикой. Вычисление корня, синусов, косинусов и др.
os	предназначена для взаимодействия с операционной системой. можно работать с файлами, получать информацию об интерфейсах операционной системы и другое.
pathlib	позволяет работать с путями в файловой системе. Можно преобразовывать пути из одного типа в другой, выполнять с ними различные операции.
random	реализует генератор псевдо-случайных чисел.
re	содержит базовый набор функций для работы с регулярными выражениями синтаксиса perl. Есть методы для поиска, замены, редактирования, удаления и др.
socket	- python поддерживает работу с сокетами напрямую. Вы можете без модуля http или url подключатся к любому системному или сетевому сокету и использовать его.
sqlite	позволяет использовать высокопроизводительную базу данных, которая полностью хранится в одном файле, в папке с программой.
ssl	позволяет работать с сертификатами ssl, используется для получения html страниц по протоколу https.
string	для работы со строками. (слияние строк, удаление лишних символов, замена, поиск и так далее.)
threading	реализует поддержку многопоточности для python. Здесь содержатся методы для управления потоками и получения информации о них;
time	- по возможностях и назначению эта библиотека похожа на datetime. Только ее методы рассчитаны на работу с датой и временем. Здесь реализовано множество функций стандартной библиотеки Си.
tkinter	для реализации графического интерфейса программ с помощью инструментария Tk GUI. Этот интерфейс будет работать как в Windows, так и в Linux системах.
urllib и **urllib2**	позволяют реализовать простой парсер или браузер на python. Можно получать и разбирать содержимое веб-страниц (работу с куки, заголовками и другими вещами библиотека берет на себя).
xml	помогает анализировать структуры XML, добавлять в структуру новые теги, менять значения существующих.

Импорт собственных модулей.

Импорт собственных модулей.

в текущем проекте создадим новый файл mymodule.py

```
NAME = "mymodule"
def floor(x):
    print('функция floor из модуля mymodule')
    return x
```

В глобальном пространстве этого файла существуют переменные:
NAME, floor

Импортируем этот модуль в ex1.py

```
import mymodule
mymodule.floor(-5.4) # функция floor из модуля mymodule  -5.4
```

Или так:

```
from mymodule import floor
print(floor(-5.4))
```

```
# в mymodule.py:
import math
NAME = "mymodule"

def floor(x):
    print('функция floor из mymodule')
    return x

#ex1.py
import mymodule
import pprint
pprint(dir(mymodule))
a = mymodule.math.floor(-5.6) # -6
```

Пространство имен в ex1.py содержит импортированные имена NAME, floor и имя библиотеки math:

```
['NAME',
'__builtins__',
'__cached__',
'__doc__',
'__file__',
'__loader__',
'__name__',
'__package__',
'__spec__',
'floor'
'math']
```

Не стоит импортировать через from math import *!!!!

```
#mymodule.py:
from math import *
NAME = "mymodule"
```

```
#ex1.py
import mymodule
import pprint
pprint(dir(mymodule))
# В пространстве имен ex1.py у нас окажется ВСЕ! имена из math!
```

```
#mymodule.py:
from math import pi, ceil
# эти имена будут находиться непосредственно в пространстве имен  mymodule.py
NAME = "mymodule"

def floor(x):
    print('функция floor из mymodule')
    return x
```

```
#ex1.py
import mymodule
import pprint
a = mymodule.ceil(-5.6) # -5
```

Порядок импорта прописан в специальной переменной

```
import pprint
import sys
pprint.pprint(sys.path) # path ссылается на коллекцию путей, где и происходит поиск модулей
```

Если модуль находится в подкаталоге нашего рабочего каталога:

```
import folder.mymodule
```

(или добавить в path новый мршрут)

```
import sys
sys.path.append(r"D:\Python\Projects\stepik\folder")
# добавлять маршрут нужно до импорта нашего модуля
import module
```

если все наоборот:
исполняемый файл в подкаталоге, а импортируемый в рабочем каталоге, то ничего делать ненужно:
в path автомтически будут прописаны и путь до исполняемого файла и путь до рабочего каталога проекта

В момент импорта Python компилирует импортируемый модуль в байт-код и исполняет его один раз

если в импортируемом файле был print, то он исполнится в исполняемом файле, в который импортировали этот модуль
подключаемые модули должны содержать переменные, но не их вызовы.

Переменная __name__

- принимает имя импортированного модуля если к ней обращаются из исполняемого файла.
- принимает значение __main__, если к ней обращаются внутри импортируемого файла

#mymodule.py:
```
print(__name__)
```

#ex1.py
```
import mymodule
import pprint
```

При исполнении импортированного mymodule в ex1.py:
```
 >>> mymodule
```

При исполнении mymodule.py:
```
>>> __main__
```

```
if __name__ == "__main__":
    print("самостоятельный запуск")
else:
    print("запуски при импорте")
```

#mymodule.py:
```
NAME = "mymodule"
if __name__ == "__main__":
# исполнит блок, если запуск происходит непосредственно из mymodule.py
    for i in range(5):
        print(NAME)
```

#ex1.py
```
import mymodule
import pprint
```

Замкнём импорт:

#mymodule.py:
```
import ex1
NAME = "mymodule"
print(NAME)
```

#ex1.py
```
import mymodule
print(ex1)
```

1) Если запустим ex1:
```
>>> ex1
>>> mymodule
>>> ex1
```
- сначала выполнился import ex1 в импортированном mymodule.py
- потом был скомпилирован и один раз выполнен импортируемый mymodule
- затем выполнен остальной код ex1.py

2) Если запустим mymodule:
```
mymodule
ex1
mymodule
```

Бесконечного замыкания не происходит, так как **импортируемый модуль выполняется только один раз**!

#mymodule.py:
```
NAME = "mymodule"
print(NAME)
```

#ex1.py
```
import mymodule
import mymodule
print("ex1")
```

При запуске ex1 (модуль запускается один раз, повторный импорт не срабатывает):
```
mymodule # import mymodule будет выполнен только один раз
ex1
```

Если нужно сделать повторный импорт (с python 3.4):

#mymodule.py:
```
NAME = "mymodule"
print(NAME)
```

#ex1.py
```
import mymodule
import importlib
importlib.reload(mymodule)
# вписываем модуль, который мы хотим обновить (импортировать заново)
print("ex1")
```

Установка внешних модулей

установка модулей осуществляется с привязкой к выбранной версии языка Python!

NumPy	работа с многомерными массивами
Matplotlib	отображение графиков
Pygame	реализация простой 2D-графики
Flask	простой фреймворк (часто для информационных сайтов)
Django	продвинутый фреймворк для самых сложных сайтов
и др.	
в командной строке	
pip list	получим список сторонних пакетов для текущего интерпретатора Python
pip install pygame	установка пакета pygame
pip install flask==1.1.2	установка конкретной версии пакета flask

pypi.org - репозиторий сторонних пакетов

Пакеты можно устанавливать в PyCharm.

Settings->ProjectInterpreter.
Увидим окно с текущим интерпретатором языка Python и набором внешних установленных пакетов, и их версий.
Чтобы добавить новый пакет, выбираем «+» и в открывшемся окне через поиск находим нужный пакет и нажимаем кнопку «Install Package».

Расположение установленных модулей.
Наведем курсор мыши на название любого пакета в PyCharm и увидим путь его расположения.
Этот путь также присутствует в специальной коллекции:

```
import sys
print(sys.path)
```

(все установленные внешние пакеты могут быть импортированы без доп.указаний каталогов и подкаталогов)

пакетная установка внешних модулей.

```
pip install –r <текстовый файл>
```

В текстовом файле нужно прописать имена устанавливаемых модулей, обычно, с указанием требуемых версий, но можно и без них (тогда будет установлена последняя версия пакета):

```
numpy==1.19.0
Pillow==7.2.0
six==1.15.0
wxPython==4.1.0
```

Этот файл можно сформировать вручную, но часто такая пакетная установка используется для переноса проекта с одного компьютера на другой. Поэтому, в файле нужно прописать все установленные внешние пакеты.
Чтобы не делать этого вручную, можно выполнить команду (в командной строке Pycharm):

```
pip freeze > requirements.txt
```

и в рабочем каталоге проекта появится текстовый файл requirements.txt с набором всех установленных модулей для текущего интерпретатора.

Для установки этих пакетов:

```
pip install –r requirements.txt
```

Пакеты

Модули - отдельные файлы с текстами программ, которые можно импортировать в другие программы.
Пакет - спец. образом организованный подкаталог с набором модулей, как правило, решающих сходные задачи.

В рабочем каталоге создадим новый py-package 'courses':

```
|- ex1.py
|- courses
    |- __init__.py
    |- добавим сюда несколько py-files:
        python.py, php.py, html.py, java.py...
```

Для корректной обработки модулей в пакете все вложенные файлы должны быть в кодировке utf8.
(в PyCharm кодировка отображается в правом нижнем углу)

__init__.py *# инициализатор пакета: в нем указываем, что импортировать при импорте всего пакета.*
```
NAME = "package courses"
```

html.py:
```
def get_html():
    print("курс по HTML")
```

php.py:
```
def get_php():
    pass
def get_mysql():
    pass
```

java.py:
```
def get_java():
    print("курс по Java")
```

python.py:
```
def get_python():
    print("курс по Python")
```

ex1.py: # в основном модуле
```
import courses
print(dir(courses)) # посмотрим, что мы импортировали
```

При импорте пакета автоматически выполняется __init__.py и все, что в нем определено.
В данном случае импортировалась только переменная NAME

__init__.py
```
import courses.python
# импортировали модуль python, теперь он импортируется при импорте пакета courses
NAME = "package courses"
```

ex1.py:
```
import courses
print(dir(courses))
# добавилось пространство имен python в котором содержится get_python()
courses.python.get_python()
# можем обратиться к функции из пакета
```

__init__.py
```
from courses.python import get_python
# абсолютный путь
NAME = "package courses"
```

в ex1.py:
```
import courses
courses.get_python()
# можно уже не прописывать пространство имен python
```

Чтобы не зависеть от имени пакета:
```
from .python import get_python  # относительный путь. обращаемся к текущему пакету.
from . import html, java, php, python # импорт на уровне модулей
```

__init__.py
```
from .php import * # внутри пакетов так можно делать
NAME = "package courses"
```

Конфликта имен можно избежать, контроллируя импортируемые переменные:

php.py:
```
__all__ = ['get_php', 'get_sql']
# разрешенные для импорта элементы из этого модуля, если происходит импорт с помощью *.
def get_php():
    pass
def get_mysql():
    pass
```

ex1.py:
```
courses.get_php()
```

Можно создавать вложенные пакеты
Внутри courses:
```
|- doc
    |- __init__.py
    |- java_doc.py, python_doc.py
```

courses.doc.__init__.py:
```
from . import python_doc, java_doc
```

courses.__init__.py:
```
from .doc import *
```

ex1.py:
```
import courses
print(courses.python_doc.doc)
# обращение к переменной doc в модуле python_doc
```

Обращение к родительской папке.
В модулях вложенного пакета можно обращаться к модулям внешнего пакета.
В **python_doc** (импорт модуля python):
```
from ..python import get_python
doc = """Документация по языку Python: """ + get_python()
```

Функция open. Чтение данных из файла.

главная особенность файлов - сохранение информации после отключения устройства от питания

open(file[, mode='r', encoding=None, ...])
Открывает указанный файл на чтение или запись.
file - путь к файлу (вместе с его именем)
mode - режим доступа к файлу (чтение, запись)
encoding - кодировка файла

Относительные пути.
"my_file.txt" # если файл находится в том же каталоге, что и исполняемый файл
"images/img.txt"
"../out.txt" # обратиться к родительскому каталогу (на один уровень наружу)
"../parent/prt.dat"

Абсолютные пути.
"d:\\app\\my_file.txt" # абсолютный путь (будет работать только под win)
"d:/app/my_file.txt" # с такими разделителями предпочтительней - будет работать и под win и под linux
"d:/app/images/img.txt"

file = open("my_file.txt", encoding='utf-8')
Файл будет открыт на чтение при запуске программы.
При закрытии программы файл автоматически закроется.

print(file.read())	# выведет все, что прочитает из файла
print(file.read(4))	# выведет первые 4 символа из файла (в кодировке utf-8 считается первый невидимый символ с кодом #FEFF)
print(file.read(4))	При повторном вызове file.read(4) будут прочитаны следующие 4 символа
Файловая позиция	место, с которого начнется чтение при следующем вызове .read() В конце файла находится невидимый символ EOF (End of file)
file.seek(offset[, from_what])	**Управление положением файловой позиции.**
file.seek(0)	# переместит файловую позицию в начало.
pos = file.tell()	# возвращает текущую файловую позицию Каждая буква русского алфавита - 2 байта в utf-8. Файловая позиция - не номер символа, а номер байта (на символе №5 файловая позиция будет 9)
print(file.readline())	# чтение первой строки из файла Будет считан и символ конца строки \n (либо символ конца файла)
print(file.readline())	# при повторном вызове будет прочитана вторая строка (и print добавит еще один символ \n, то есть между двумя строкми будет пустая строка)
for line in file: print(line, end="")	**файл - итерируемый объект и при итерации он возвращает новую строку**
s = file.readlines()	# считает все строки из файла. Вернёт список из строк ['\ufeff 1 строка\n', '2 строка\n', '3 строка\n'] Но для больших файлов может возникнуть ошибка нехватки памяти для хранения полученного списка.
file.close()	При завершении работы с файлом его обязательно надо закрыть - освобождаем память, связанную с этим файлом - И не будет проблем с потерей данных при записи их в файл.

Обработка исключения FileNotFoundError и менеджер контекста

FileNotFoundError
Для обработки подобных ошибок существует специальная группа операторов

```
try:
    Блок операторов критического кода
except:
    Блок операторов обработки исключения
finally:
    Блок операторов всегда исполняемых, вне зависимости от возникновения исключения.
```

Как только где-либо в блоке try возникает ошибка, мы сразу переходим в блок except.
Все остальные строки в try после возникновения исключения уже не выполняются

```
try:
    file = open("my_file.txt", encoding='utf-8')
    # предусмотрим ситуацию, когда файл открылся, но строчки из него не прочитались
    # необходимо, чтобы открытый файл был в конце закрыт независимо от дальнейших ошибок
    try:
        s = file.readlines()
        int(s) # это невозможная операция. программа перейдет на  блок  finally
        # после выполнения finally попадаем на второй блок exept
        print(s)
    finally:
        file.close()
except FileNotFoundError:
    print("Невозможно открыть файл")
except: # отловим все остальные исключения, кроме  FileNotFoundError
    print("Ошибка при работе с файлом")
```

Вместо внутреннего блока try-finally можно воспользоваться **файловым менеджером контекста**:

```
try:
    with open("my_file.txt", encoding='utf-8') as file:
    # откроет файл под именем file и независимо от ошибок в конце операций закроет файл.
        s = file.readlines()
        print(s)
except FileNotFoundError:
    print("Невозможно открыть файл")
except: # отловим все остальные исключения, кроме  FileNotFoundError
    print("Ошибка при работе с файлом")
finally:
    print(file.closed) # выведем флаг закрытия файла
```

Запись данных в файл.

```
file = open("out.txt", "w") # открыли файл на запись. Если в нем было содержимое, то оно уничтожается
file.write("Hello World!") # запишет строку в файл, предварительно обнулив файл
file.close()
```

```
try:
    with open("out.txt", "w") as file:
        file.write("Hello1")
        file.write("Hello2\n") # при записи тоже используется файловая позиция.
        # эти строчки будут записаны друг за другом без разделителей
except:
    print("Ошибка при работе с файлом")
```

Чтобы дозаписать информацию в уже существующий файл:
(Если файл не существует, то он будет создан)

```
try:
    with open("out.txt", "a") as file:  # append
        file.write("Hello1\n")
except:
    print("Ошибка при работе с файлом")
```

Если файл открыт на запись или дозапись, то чтение из него невозможно

Чтобы и записывать и считывать информациюв файле:

```
try:
    with open("out.txt", "a+") as file:  # при режиме "a+" файловая позиция будет в конце файла
        file.seek(0) # файловая позиция для чтения
        file.write("Hello4\n") # файловая позиция на запись при этом останется в конце файла
        # Для записи используется своя отдельная файловая позиция
        s = file.readlines()
        print(s)
except:
    print("Ошибка при работе с файлом")
```

```
file.writelines(["string_1\n", "string_2\n", "string_3\n"]) # запись в файл списка строк
```

Бинарный режим работы.

данные из файла считываются один в один без какой-либо обработки
(используется для сохранения и считывания объектов целиком)

```
import pickle
books = [. . .]
file = open("out.bin", "wb") #    бинарный режим доступа
pickle.dump(books, file)
file.close()
```

Прочитаем файл в бинарном режиме

```
file = open("out.bin", "rb")
bs = pickle.load(file)
file.close()
print(bs)
```

Сохраним в бинарный файл несколько коллекций.

```
import pickle
book1 = [. . .]
book2 = [. . .]
book3 = [. . .]
book4 = [. . .]

try:
    with open("out.bin", "wb") as file:
        pickle.dump(book1, file)
        pickle.dump(book2, file)
        pickle.dump(book3, file)
        pickle.dump(book4, file)
except:
    print("ошибка при работе с файлом")
```

```
try:
    with open("out.bin", "rb") as file:
        b1 = pickle.load(file)
        b2 = pickle.load(file)
        b3 = pickle.load(file)
        b4 = pickle.load(file)
except:
    print("ошибка при работе с файлом")
print(b1, b2, b3, b4, sep="\n")
```

Генераторы.

Генераторы

Можно определить генератор без привязки к коллекции (к списку или к словарю)
Используется такой же синтаксис, как и для генераторов коллекций (списков, множеств, словарей)

```
(<формирование значения> for <переменная> in <итерируемый объект>)
# Круглые скобки не означают кортеж - генераторов кортежей не существует.
# На выходе получаем чистый генератор
a = (x ** 2 for x in range(6))
a >>> <generator object . . .>
```

Генератор является итератором:

```
next(a) >>> 1
next(a) >>> 2
. . .
next(a) >>> StopIteration
```

т.к. генератор является итератором, перебрать его мы можем только один раз

```
gen = (x ** 2 for x in range(6))
for x in gen:
    print(x) # Получим вывод чисел из генератора

for x in gen:
    print(x) # Ничего не получится. Второй раз мы перебрать этот генератор не сможем.
```

Некоторые функции позволяют работать с итератором:
list(), tuple(), set(), sum(), min(), max() и др.

```
a = (x ** 2 for x in range(6))
list(a)
```

Но второй раз воспользоваться функцией для того же генератора уже не сможем

```
s = sum((x ** 2 for x in range(6))) # должны заново задать генератор
```

В отличие от списков, генераторы не хранят в памяти сразу все значения, а генерируют их по мере необходимости:

```
lst = list(range(1000000000)) >>> MemoryError
```

Но можно:

```
lst = (x for x in range(1000000000))
for i in lst:
    print(i, end=" ")
    if i > 50:
        break
```

Такое работать будет, причем достаточно быстро

Для генераторов не будет работать len() т.к. len не знает, сколько будет значений в этом объекте.
Также нельзя обращаться к генератору по индексу

```
[(x ** 2 for x in range(6))] # так не получим список значений.
>>> [<generator object . . .>] - получим список, в котором хранится ссылка на генератор.
```

Для создания списка из генератора нужно использовать list()

Функция-генератор. Оператор yield.

```
def get_list():
    for x in [1, 2, 3, 4]:
        return x # такая функция вернет только первое значение и завершит работу.
```

```
Превращаем функцию в функцию-генератор.
def get_list():
    for x in [1, 2, 3, 4]:
        yield x # замораживает состояние до следующего обращения к функции
```

```
a = get_list()
print(a)
>>> <generator object . . .> # получили функцию-генератор
print(next(a))  >>> 1
print(next(a))  >>> 2
print(next(a))  >>> 3
```

```
Вычислим средние арифметические для последовательностей вида [1,2,...10], [2,3,...10] и т.д.
def get_list():
    for i in range(1, 10):
        a = range(i, 11)
        yield sum(a) / len(a)

a = get_list()
print(list(a))
>>> [5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5]
```

```
a = gen_func() # это какая-нить функция-генератор
print(next(a))
print(next(a))
Выдаст 2  последовательных значения функции генератора (тут - первое и второе)
```

```
print(next(gen_func()))
print(next(gen_func()))
Выдаст 2  первых значения двух разных генераторов
```

```
т.е. ссылку на объект генератор сначала надо присвоить какой-то переменной (например a = gen_func() )
и только потом по этой а итерировать.
в случае next(gen_func()) каждый раз будет создаваться первая итерация для новосозданного объекта
```

```
yield в отличии от return не останавливает функцию.
В функции может быть выполнено несколько yield (например цикл в цикле).
```

```
def find_word(f, word): # функция поиска индекса слова в текстовом файле.
    g_ingx = 0
    for line in f: # читаем файл построчно
        indx = 0 # индекс вхождения слова в текущую строку
        while(indx != -1):
            indx = line.find(word, indx)
            if indx > -1:
                yield g_indx + indx
                indx += 1
        g_indx += len(line)

# Откроем файл на чтение:
try:
    with open("lesson.txt", encoding="utf-8") as file:
        a = find_world(file, "генератор") # обрабатываем файл с помощью функции-генератора
        print(list(a))
except FileNotFoundError:
    print("Файл не найден")
except:
    print("Ошибка обработки файла!")
```

```
yield - возвращает текущее значение и ПРЕРЫВАЕТ текущую итерацию, замораживая ее состояние,
т.е. в текущей итерации после yield не произойдет никаких дальнейших изменений.
Изменения после yield из первой итерации произойдут только когда программа вновь придет к этому циклу,
в таком случае будет происходить НЕ СЛЕДУЮЩАЯ итерация, а окончание ПРЕДЫДУЩЕЙ
```

Функции как объекты

Функции - тоже объекты.
Их можно записывать в переменные, передавать в качестве аргументов, возвращать из функций и т.д.

```
 def hello():
     print('Hello from function')

 print(type(hello))   >>>   <class 'function'>
 func = hello     #  присваиваем переменной func функцию hello
 func()           #  вызываем функцию
```

можно создать словарь, где ключом служит название команды, а значением — соответствующая функция.

```
def start():
    # тело функции start
    pass

def stop():
    # тело функции stop
    pass

commands = {'start': start, 'stop': stop}  # словарь соответствия команда → функция
command = input()  # считываем название команды
commands[command]()  # вызываем нужную функцию через словарь по ключу
```

Математическая функция может быть аргументом для функции построения графика.

```
 # f – функция, для которой хотим построить график, и a, b – границы диапазона построения графика.
 def plot(f, a, b):
    ...

 def cube_add_square(x):
     return x**3 + x**2

 plot(cube_add_square, -10, 10)  # построит график функции y=x3+x2 в диапазоне [-10;10]
```

функции высшего порядка - функции, которые принимают или/и возвращают другие функции

Встроенные функции min(), max(), sorted() – функции высшего порядка, так как принимают в качестве аргумента функцию сравнения элементов (key).

функции могут быть результатом работы других функций, что позволяет писать генераторы функций, возвращающие функции.

```
 # функция generator() возвращает функцию hello() в качестве результата своей работы:
 def generator():
     def hello():
         print('Hello from function!')
     return hello

 # Результат функции generator() можно записать в переменную, и использовать эту переменную как функцию.
 func = generator()
 func()
 >>>  Hello from function!
```

генератор функций, который по параметрам a,b,c вернет нам конкретный квадратный трехчлен:
$f(x)=x^2+2x+1$,
$g(x)=2x^2-3$,

```
def generator_square_polynom(a, b, c):
    def square_polynom(x):
        return a * x**2 + b * x + c
    return square_polynom

f = generator_square_polynom(a=1, b=2, c=1)
g = generator_square_polynom(a=2, b=0, c=-3)
print(f(1))   >>> 4
print(g(2))   >>> 5
```

внутренняя функция square_polynom() использует параметры внешней функции generator_square_polynom().
Замыкания – вложенные функции, ссылающиеся на переменные, объявленные вне определения этой функции, и не являющиеся её параметрами.

```
 def high_order_function(func):     # функция высшего порядка, так как принимает функцию
     return func(3)                 # возвращает результат её вызова с аргументом, равным 3.

 def double(x):                     # обычная функция = функция первого порядка
     return 2*x

 print(high_order_function(double))   >>> 6
```

Сортировка с помощью sort и sorted

list.sort()	метод для сортировки элементов списка.
	меняет сам список, ничего не возвращает!
	`a=[1,-45,3,2,100,-4]`
	`a.sort()`
	`print(a)  >>>  [-45, -4, 1, 2, 3, 100]`

sorted(iter)	функция для сортировки любых итерируемых объектов.
	На выходе - всегда упорядоченный список, независимо от того, какая коллекция на входе.
	не меняет исходные коллекции,
	возвращает новый список с отсортированными данными.
	`b=("ab", "bc", "wd", "gf")`
	`c = "hello world"`
	`b.sort()  >>> приведет к ошибке.`
	
	`res1 = sorted(b)  >>> ['ab', 'bc', 'gf', 'wd']`
	`res2 = sorted(c)  >>> [' ', 'd', 'e', 'h', 'l', 'l', 'l', 'o', 'o', 'r', 'w']`

сортировка возможна для однотипных элементов: или чисел, или строк, или кортежей, но не их комбинаций.

```
a=[1,-45,3,2,100,-4, "b"]
a.sort()  >>> возникнет ошибка, что строку нельзя сравнивать с числом.
```

необязательный параметр reverse = True/False
По умолчанию reverse=False
определяет порядок сортировки: по возрастанию (False) или по убыванию (True).

```
a = sorted(a, reverse=True)  >>> [100, 3, 2, 1, -4, -45]
a.sort(reverse=True)
```

сортировка для словаря:

```
d = {'river': "река", 'house': "дом", 'tree': "дерево", 'road': "дорога"}
k = sorted(d)  # сортировка ключей по возрастанию (на выходе – список ключей)
v = sorted(d.values())

d1 = sorted(d.items()) # отсортированный список кортежей (ключ, значение)
dict(d1) # словарь, отсортированный по ключам
```

Аргумент key для сортировки коллекций по ключу

необязательный аргумент key – функция, определяющая условия сравнения элементов.
(принимает один аргумент и возвращает на его основе ключ для сравнения.
Функция, определяющая условия сравнения элементов, называется **компаратор** (compare – сравнивать).

Функция должна принимать один аргумент и обязательно что-нибудь возвращать

```
numbers = [10, -7, 8, -100, -50, 32, 87, 117, -210]
print(max(numbers, key=abs))        >>> -210    # максимальный по модулю элемент
print(sorted(numbers, key=abs))     >>> [-7, 8, 10, 32, -50, 87, -100, 117, -210]
```

```
points = [(1, -1), (2, 3), (-10, 15), (10, 9), (7, 18), (1, 5), (2, -4)]
points.sort()    #  сортируем список точек на месте

#  сортировка пройдет по первым элементам кортежей, а в случае их совпадения – по вторым
print(points)
>>>   [(-10, 15), (1, -1), (1, 5), (2, -4), (2, 3), (7, 18), (10, 9)]
```

```
def compare_by_second(point):
    return point[1]

def compare_by_sum(point):
    return point[0] + point[1]

print(sorted(points, key=compare_by_second))   # сортируем по второму значению кортежа
print(sorted(points, key=compare_by_sum))      # сортируем по сумме кортежа
>>>
[(2, -4), (1, -1), (2, 3), (1, 5), (10, 9), (-10, 15), (7, 18)]
[(2, -4), (1, -1), (2, 3), (-10, 15), (1, 5), (10, 9), (7, 18)]
```

Пусть key - показатель четности значений. (0 - четные значения, 1 – нечетные)
Тогда последовательность будет выстроена по возрастанию этих ключей .

```
lst = [4,  3, -10, 1, 7, 12]
keys: [0,  1,   0, 1, 1, 0]
>>>   [4, -10, 12, 3, 1, 7]
```

```
def is_odd(x):    # аргумент x – это текущее значение элемента коллекции
    return x % 2  # значение ключа:  для четных значений 0, а для нечетных – 1
```

аргументу key присваиваем ссылку на функцию, которая будет формировать значение ключа:

```
b = sorted(a, key=is_odd)
```

для простых функций, обычно, в аргументе key записывают лямбда-функцию:

```
b = sorted(a, key=lambda x: x % 2)
a.sort(key=lambda x: x % 2)
```

Разделение на четные и нечетные, с сортировкой каждой группы по возрастанию.
Значения ключей увеличиваем на значения элементов. Нечетным величинам изначально присваивать число 100, чтобы гарантированно разделить четные и нечетные значения в нашей коллекции.

```
lst = [ 4,  3,   -10, 1,   7,   12]
keys: [ 0,  100, 0,   100, 100, 0]
sum:  [+4,  +3,  -10, +1,  +7,  +12]
>>>   [-10,  4,  12,  1,   3,   7]
```

```
def key_sort(x):
    return x if x % 2 == 0 else 100 + x

b = sorted(a, key=key_sort)
```

Сортировка по длине названий (в аргументе key укажем стандартную функцию len):

```
lst = ["Москва", "Тверь", "Смоленск", "Псков", "Рязань"]
print( sorted(lst, key=len) )
>>> ['Тверь', 'Псков', 'Москва', 'Рязань', 'Смоленск']
```

Сортировка по последнему символу слова:

```
print( sorted(lst, key=lambda x: x[-1]) )
>>> ['Москва', 'Псков', 'Смоленск', 'Тверь', 'Рязань']
```

отсортировать кортеж кортежей по цене (последнее значение)

```
books = (
    ("Евгений Онегин", "Пушкин А.С.", 200),
    ("Муму", "Тургенев И.С.", 250),
    ("Мастер и Маргарита", "Булгаков М.А.", 500),
    ("Мертвые души", "Гоголь Н.В.", 190)
)
print( sorted(books, key=lambda x: x[2]) )
>>>
[('Мертвые души', 'Гоголь Н.В.', 190), ('Евгений Онегин', 'Пушкин А.С.', 200), ('Муму', 'Тургенев И.С.',
250), ('Мастер и Маргарита', 'Булгаков М.А.', 500)]
```

Сортировка при помощи функции sorted() и списочного метода sort() стабильна, то есть гарантирует неизменность взаиморасположения равных между собой элементов.

```
def comparator(item):
    return item[0]

data = [('red', 1), ('blue', 2), ('green', 5), ('blue', 1)]
data.sort(key=comparator)   # сортируем по первому полю
print(data)   >>> [('blue', 2), ('blue', 1), ('green', 5), ('red', 1)]
# две записи с 'blue' сохранили начальный порядок.
```

Функции max() и min() возвращают **первый** максимальный или минимальный элемент, если таковых несколько.

Сортировка по двум (и более) параметрам.
Сортировка списка по сумме цифр. Если сумма цифр совпадает, то сортировка по значению элементов.

```
s = '111 14 12 79 7 4 123 45 90'.split()
s = [int(i) for i in s]

def comparator(item):
    return sum(map(int, tuple(str(item)))), item
# функция возвращает кортеж (сумма, само_число). Сначала сортировка по сумме, потом по числу.

print(*sorted(s, key=comparator))
>>> 12 111 4 14 123 7 45 90 79
```

Сортировка списка по другому списку.

```
lst_in = ['Атос=лейтенант', 'Портос=прапорщик', "д'Артаньян=капитан",
          "Арамис=лейтенант", "Балакирев=рядовой"]
lst = [i.split('=') for i in lst_in]
ranks = 'рядовой сержант старшина прапорщик лейтенант капитан майор подполковник полковник'.split()
t_sorted = sorted(lst, key=lambda x: ranks.index(x[1]))
>>>
[['Балакирев', 'рядовой'], ['Портос', 'прапорщик'], ['Атос', 'лейтенант'], ['Арамис', 'лейтенант'],
["д'Артаньян", 'капитан']]
```

Сортировка сначала по длине слова, потом в лексикографическом порядке:

```
print(*sorted(data, key=lambda x: (len(x), x)))
```

max(iter, key), если в списке смешанные значения (int, str)

```
mixed_list = ['a', 'ab', 3, 5, 1, 8, 0, 'c', 'ac', 'aab']
print(max(mixed_list, key=lambda x: x if type(x) == int else 0))
```

Функция map()

Применяет одно и то же преобразование к каждому элементу списка.

Преобразование списка чисел в список их квадратов

```
def f(x):
    return x**2     # тело функции, которая преобразует аргумент x

old_list = [1, 2, 4, 9, 10, 25]
new_list = []
for item in old_list:
    new_item = f(item)
    new_list.append(new_item)
print(new_list)   >>> [1, 4, 16, 81, 100, 625]
```

Цикл будет выглядеть одинаково практически во всех случаях. Меняться будет только применяемая функция f(). Можно обобщить код, чтобы функция была параметром:

```
def map(function, items):
    result = []
    for item in items:
        new_item = function(item)
        result.append(new_item)
    return result
```

Теперь мы можем совершать преобразования, используя функцию высшего порядка map().

```
def square(x):
    return x**2

numbers = [1, 2, -3, 4, -5, 6, -9, 0]
strings = map(str, numbers)         # используем в качестве преобразователя – функцию str
abs_numbers = map(abs, numbers)     # используем в качестве преобразователя – функцию abs
squares = map(square, numbers)      # используем в качестве преобразователя – функцию square

print(strings)   >>> ['1', '2', '-3', '4', '-5', '6', '-9', '0']
print(abs_numbers)   >>> [1, 2, 3, 4, 5, 6, 9, 0]
print(squares)   >>> [1, 4, 9, 16, 25, 36, 81, 0]
```

Функция называется "map" то есть "отобразить". Название пришло из математики, где так называется функция, отображающая одно множество значений в другое путём преобразования всех элементов с помощью некой трансформации.

Цепочки преобразований

```
new_numbers = map(abs, map(int, numbers))
# map(int, numbers) – сначала преобразуем список строк в список чисел
# Далее вычисляем модули у всех элементов полученного списка
```

Встроенная функция map()

a = map(func, *iter)

func - ссылка (!) на функцию, которая будет последовательно применяться к каждому элементу списка,
iter – любой итерируемый объект.
На выходе функция map() возвращает итератор:

```
print(a)   >>> <map object at 0x...>
```

```
b = map(int, ['1', '2', '3', '5', '7'])  # b – это итератор
print(next(b))
print(next(b))
```

Или переберем все их с помощью цикла for:

```
for x in b:
    print(x, end=" ")
```

функция map() последовательно применила функцию int() к каждому элементу списка.

Можно сохранить результат преобразования в новом списке, используя функцию list():

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

функция list() автоматически перебрала итератор, неявно вызывая функцию next().

Этот же результат можно получить, используя генератор списка.

```
numbers1 = [int(c) for c in strings]  # используем списочное выражение для преобразования
numbers2 = map(int, strings)          # используем функцию map() для преобразования
```

В списочном выражении все значения храним в памяти.
Функция map() возвращает итератор и значения формируются по одному в процессе вызова функции next().

эквивалентный map() генератор в классическом виде:

```
a = (int(x) for x in ['1', '2', '3', '5', '7'])
```

функции list(), sum(), max(), min() - в качестве аргумента принимают итерированный объект

```
print(sum(b))
```

Но, если следом попытаться перебрать итератор еще раз:

```
print(sum(b))
```

то увидим значение 0, так как итератор можно перебирать только один раз.

```
cities = ["Москва", "Астрахань", "Самара", "Уфа", "Смоленск", "Тверь"]
b = map(len, cities)
```

Строковые методы в map.

```
b = map(str.upper, cities)
```

Использование своих функций в map().
функция обязательно должна принимать один аргумент и возвращать некоторое значение

```
def symbols(s):
    return list(s.lower()) # (формируем список из отдельных символов в нижнем регистре)

b = map(symbols, cities)
```

(без круглых скобок, то есть, передаем ссылку на функцию, а не вызываем).
>>> наборы вложенных списков из отдельных символов исходных строк.

если функция выполняет какую-либо простую операцию, то часто используют лямбда-функции:

```
b = map(lambda s: list(s.lower()), cities)
```

преобразовать строки, записав их символы в обратном порядке:

```
b = map(lambda s: s[::-1], cities)
```

```
s = map(int, input().split())
print(list(s))
```

input().split() возвращает список из строк введенных чисел,
к каждой строке применяется функция int()
с помощью функции list() генератор s превращается в список из чисел.

Несколько последовательностей в map()

map(func, *iterables).
может принимать сразу несколько последовательностей, переменное количество аргументов.
func - функция, которой будет передаваться текущий элемент последовательности.

Функции map() можно передать несколько последовательностей.
В func будут передаваться сразу несколько элементов, расположенных в последовательностях на одинаковых позициях.
Если в последовательностях разное количество элементов, то последовательность с минимальным количеством элементов становится ограничителем.

суммируем элементы трех списков:

```
def func(elem1, elem2, elem3):
    return elem1 + elem2 + elem3

numbers1 = [1, 2, 3, 4, 5]
numbers2 = [10, 20, 30, 40, 50]
numbers3 = [100, 200, 300, 400, 500]

new_numbers = list(map(func, numbers1, numbers2, numbers3))
print(new_numbers)
>>> [111, 222, 333, 444, 555]
```

```
numbers1 = [1, 2, 3, 4]
numbers2 = [10, 20]
numbers3 = [100, 200, 300, 400, 500]

new_numbers = list(map(func, numbers1, numbers2, numbers3))  #  преобразуем итератор в список
print(new_numbers)
>>> [111, 222]
```

```
circle_areas = [3.56773, 5.57668, 4.31914, 6.20241, 91.01344, 32.01213]
result1 = list(map(round, circle_areas, [1]*6))         # округляем числа до 1 знака после запятой
result2 = list(map(round, circle_areas, range(1, 7)))   # округляем до 1,2,...,6 знаков после запятой

print(result1)   >>> [3.6, 5.6, 4.3, 6.2, 91.0, 32.0]
print(result2)   >>> [3.6, 5.58, 4.319, 6.2024, 91.01344, 32.01213]
```

Встроенная функция round(x, n=0) принимает два числовых аргумента x и n и округляет переданное число x до n цифр после десятичной запятой. По умолчанию n = 0.

Функция filter()

Отбирает элементы списка по определенному критерию.
Функция-критерий, которая возвращает значение True или False, называется предикатом.
Реализация такой функции может выглядеть так:

```
def filter(function, items):
    result = []
    for item in items:
        if function(item):
            result.append(item)  # добавляем элемент item если функция function вернула значение True
    return result
```

Наша функция filter() применяет предикат function к каждому элементу и добавляет в итоговый список только те элементы, для которых предикат вернул True.

Отобрать только элементы больше 10:

```
def is_greater10(num):  # функция возвращает True если num > 10 и False в противном случае
    return num > 10

numbers = [12, 2, -30, 48, 51, -60, 19, 10, 13]
large_numbers = filter(is_greater10, numbers)  #  список large_numbers содержит элементы, большие 10
print(large_numbers)   >>> [12, 48, 51, 19, 13]
```

```
def is_odd(num):
    return num % 2

def is_word_long(word):
    return len(word) > 6

numbers = list(range(15))
words = ['В', 'новом', 'списке', 'останутся', 'только', 'длинные', 'слова']
odd_numbers = filter(is_odd, numbers)      >>> [1, 3, 5, 7, 9, 11, 13]
large_words = filter(is_word_long, words)  >>> ['останутся', 'длинные']
```

Встроенная функция filter()

служит для фильтрации элементов итерированного объекта.

filter(func, iterable)
iterable - любой итерируемый объект (список, строка, кортеж, и т.д.).
func - ссылка на функцию, которой будет передаваться текущий элемент последовательности.

Внутри функции func необходимо вернуть значение True или False.
Если func возвращает для текущего элемента True, то он будет возвращен, если False – отброшен.
filter() возвращает не список, а итератор.

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  #выберем только четные значения.
b = filter(lambda x: x % 2 == 0, a)
print(b)
>>> <filter object at...>
```

Переменная b ссылается на специальный объект filter.
Это итератор, который можно перебрать с помощью функции next():

```
print(next(b))
print(next(b))
```

Или с помощью цикла for:

```
for x in b:
    print(x, end=" ")
```

Или с помощью функции list(), или tuple():

```
c = list(b)
c = tuple(b)
```

удалим все отрицательные значения из списка.

```
def func(elem):
    return elem >= 0

numbers = [-1, 2, -3, 4, 0, -20, 10]
positive_numbers = list(filter(func, numbers))  #  преобразуем итератор в список
print(positive_numbers)   >>> [2, 4, 0, 10]
```

filter можно применять с любыми типами данных, например, строками.

```
lst = ("Москва", "Рязань1", "Смоленск", "Тверь2", "Томск")
b = filter(str.isalpha, lst)  # True, если в строке только буквенные символы.
print(list(b))
>>> Москва Смоленск Тверь Томск
```

Проверим, является ли число простым (натуральное число, которое делится только на себя и на единицу).

```
def is_prost(x):
    d = x-1
    if d < 0:
        return False
    while d > 1:
        if x % d == 0:
            return False
        d -= 1
    return True

b = filter(is_prost, a) # указываем только ссылку на функцию is_post, а не вызов is_post()
lst = tuple(b)  # На выходе получим только простые числа из списка.
```

Можно в качестве первого параметра func передать значение None.
Тогда каждый элемент последовательности будет проверен на соответствие значению True.
Если элемент возвращает значение False, то он не будет добавлен в возвращаемый результат.

```
true_values = filter(None, [1, 0, 10, '', None, [], [1, 2, 3], ()])
print(*true_value)
>>> 1 10 [1, 2, 3]
Значения 0, '', None, [], () позиционируется как False,
значения 1, 10, [1, 2, 3] как True.
```

Вложенные вызовы функции filter()

Функция filter() возвращает итератор
В качестве второго аргумента также можно указывать любой итерируемый объект,
--> мы можем одну функцию filter() вложить в другую:

```
a = filter(func, *iterables)
b = filter(func, a)
```

```
b = filter(is_prost, a)  # оставляем только простые числа
b2 = filter(lambda x: x % 2 != 0, b)   # из списка простых чисел выбираем четные
c = tuple(b2)
print(c)
```

Или так:

```
b2 = filter(lambda x: x % 2 != 0, filter(is_prost, a))
```

вместо этой вложенности можно немного модифицировать саму функцию is_prost(), следующим образом:

```
def is_prost(x):
    d = x-1
    if d < 0 or x % 2 == 0:
        return False
    while d > 1:
        if x % d == 0:
            return False
        d -= 1
    return True
```

```
sm = sorted(filter(lambda x: x in b and x % 2 == 0, a))
```

```
a, b = map(str.split, (input(), input()))
print(*filter(lambda a: int(a) % 2 == 0, set(a)&set(b)))
```

Функция zip

zip(iter1 [, iter2 [, iter3] ...])
перебирает соответствующие элементы заданных последовательностей и формирует из них кортежи:
кортеж из всех нулевых элементов, кортеж из первых элементов и т.д.
продолжает работу до тех пор, пока не дойдет до конца самой короткой коллекции.

zip(*iterables)
iterable - любой итерируемый объект (список; кортеж; строка; множество; словарь и т.д.)

Функция zip() возвращает итератор

```
numbers = [1, 2, 3]
words = ['one', 'two', 'three']

result = zip(numbers, words)
print(result)
print(list(result))
>>>
<zip object at 0x...>
[(1, 'one'), (2, 'two'), (3, 'three')]
```

Можно передавать функции zip() сколько угодно итерируемых объектов.

```
numbers = [1, 2, 3]
words = ['one', 'two', 'three']
romans = ['I', 'II', 'III']

result = zip(numbers, words, romans)
print(list(result))
>>>  [(1, 'one', 'I'), (2, 'two', 'II'), (3, 'three', 'III')]
```

Можно передать функции zip() даже один итерируемый объект.

```
numbers = [1, 2, 3]
result = zip(numbers)
print(list(result))
>>>  [(1,), (2,), (3,)]
```

Последовательности разной длины.
объект с наименьшим количеством элементов определяет итоговую длину.

```
numbers = [1, 2, 3, 4]
words = ['one', 'two']
romans = ['I', 'II', 'III']

result = zip(numbers, words, romans)
print(list(result))
>>>  [(1, 'one', 'I'), (2, 'two', 'II')]
```

```
a = [1, 2, 3, 4]
b = [5, 6, 7, 8, 9, 10]
z = zip(a, b)
print(*z)    >>> (1, 5) (2, 6) (3, 7) (4, 8)
```

переменная z ссылается на объект zip - это итератор, элементы которого перебираются функцией next():

```
print(next(z))
print(next(z))
```

Или через цикл for:

```
for x in z:
    print(x)  # x – кортеж из соответствующих значений списков a и b
```

функция zip() возвращает итератор. То есть перебрать элементы можно только один раз.
Повторный вызов оператора for для этой же функции ничего не возвратит.

Если нужно несколько раз обходить коллекцию, то её следует преобразовать, например, в кортеж:

```
z = tuple(zip(a, b))
```

(Но в этом случае увеличивается расход памяти)

В качестве перебираемых коллекций могут быть любые итерируемые объекты, например, строки.

```
c = "python"
z = zip(a, b, c)
>>> итератор из кортежей: (1, 5, 'p') (2, 6, 'y') (3, 7, 't') (4, 8, 'h')
```

оператор for можно записать с тремя переменными, если перебираемые значения являются кортежами:

```
for v1, v2, v3 in z:
    print(v1, v2, v3)
```

все кортежи из функции zip() мы можем получить с помощью распаковки:

```
z1, z2, z3, z4 = zip(a, b, c) # тут важно не ошибиться в количестве переменных для распаковки
print(z1, z2, z3, z4, sep="\n")
```

Здесь неявно вызывается функция next() для извлечения всех элементов.

Или, можно записать так:

```
z1, *z2 = zip(a, b, c)
print(z1, z2, sep="\n")
>>> первое значение будет помещено в переменную z1, а все остальные – в список z2.
```

Если преобразовать итератор в список:

```
z = zip(a, b, c)
lz = list(z)
```

Можно передать распакованный список на вход функции zip():

```
t1, t2, t3 = zip(*lz)
print(t1, t2, t3)
>>> (1, 2, 3, 4) (5, 6, 7, 8) ('p', 'y', 't', 'h')
```

на выходе будут сформированы три кортежа из соответствующих элементов четырех входящих кортежей.

Можно распаковать непосредственно итератор z:

```
t1, t2, t3 = zip(*z)
```

```
zip(*[iter(s)]*n)
```

в zip() N раз передается list из итератора нашего исходного списка s.
Каждый раз при вызове итератора происходит вызов следующего объекта в s.

```
x = iter([1, 2, 3, 4, 5, 6, 7, 8, 9])
print(*zip(x, x, x))

zip(*[iter(input().split())]*3) - поможет с делением на 3 столбца
```

Частые сценарии использования функции zip()

Сценарий 1. Создание словарей, когда ключи и значения находятся в разных списках.

```
keys = ['name', 'age', 'gender']
values = ['Timur', 28, 'male']

info = dict(zip(keys, values))
print(info)
>>>  {'name': 'Timur', 'age': 28, 'gender': 'male'}
```

Сценарий 2. Одновременное (параллельное) итерирование сразу по нескольким коллекциям.

```
name = ['Timur', 'Ruslan', 'Rustam']
age = [28, 21, 19]

for x, y in zip(name, age):
    print(x, y)
>>>
Timur 28
Ruslan 21
Rustam 19
```

Примечание 1.
итераторы можно обойти циклом for;
итератор можно преобразовать в список или кортеж, с помощью функций list() и tuple();
итератор можно распаковать с помощью *

Примечание 2. Можно использовать одновременно функции zip() и enumerate():

```
list1 = ['a1', 'a2', 'a3']
list2 = ['b1', 'b2', 'b3']

for index, (item1, item2) in enumerate(zip(list1, list2)):
    print(index, item1, item2)
>>>
0 a1 b1
1 a2 b2
2 a3 b3
```

Функция reduce()

Функция reduce()

Задача агрегации результата — формирование одного результирующего значения при комбинации элементов с использованием аргумента-аккумулятора.
Типичные примеры агрегации — сумма всех элементов списка или их произведение.

```
numbers = [1, 2, 3, 4, 5]
total = 0
product = 1
for num in numbers:
    total += num
    product *= num

print(total)   >>> 15
print(product)   >>> 120
```

С точки зрения математики сумма 1+2+3+4+5 может быть выражена как:
(((((0+1)+2)+3)+4)+5).
Нуль здесь - аккумулятор, точнее его начальное значение. Он не добавляет к сумме ничего, поэтому может служить отправной точкой.
А еще нуль будет результатом, если входной список пуст.
Этот цикл будет выглядеть одинаково практически во всех случаях.
Меняться будет только начальное значение аккумулятора (0 для суммы, 1 для произведения и т.д.) и операция, которая комбинирует элемент и аккумулятор.

```
def reduce(operation, items, initial_value):
    acc = initial_value
    for item in items:
        acc = operation(acc, item)
    return acc
```

```
def add(x, y):
    return x+y

def mult(x, y):
    return x*y

numbers = [1, 2, 3, 4, 5]
total = reduce(add, numbers, 0)
product = reduce(mult, numbers, 1)
print(total)   >>> 15
print(product)   >>> 120
```

В математике определенная нами функция reduce() называется **левая свёртка** (left fold), по сути, мы сворачиваем список в одно значение, начиная слева.
Существует ещё и правая свёртка (right fold).
В большинстве случаев обе свёртки дают одинаковый результат, если применяемая операция ассоциативна.

встроенная функция reduce()

```
from functools import reduce
reduce(func, iterable, initializer=None)
# Если начальное значение не установлено, то в его качестве используется первое значение из последовательности iterable.
```

Функция reduce() как и функции map() и filter() может принимать любой итерируемый объект.

```
from functools import reduce

def func(a, b):
    return a + b

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
total = reduce(func, numbers, 0)   # в качестве начального значения 0
print(total)   >>> 55
```

```
total = reduce(func, numbers)   # в качестве начального значения первый элемент списка numbers
```

Функция reduce() во второй версии языка Python была встроенной, но в Python 3 ее решили перенести в модуль functools.

Модуль operator

Модуль operator

Для стандартных математических операций можно использовать уже реализованные функции из модуля operator.

Операция	**Синтаксис**	**Функция**
Addition	a + b	add(a, b)
Containment Test	obj in seq	contains(seq, obj)
Division	a / b	truediv(a, b)
Division	a // b	floordiv(a, b)
Exponentiation	a ** b	pow(a, b)
Modulo	a % b	mod(a, b)
Multiplication	a * b	mul(a, b)
Negation (Arithmetic)	-a	neg(a)
Subtraction	a - b	sub(a, b)
Ordering	a < b	lt(a, b)
Ordering	a <= b	le(a, b)
Equality	a == b	eq(a, b)
Difference	a != b	ne(a, b)
Ordering	a >= b	ge(a, b)
Ordering	a > b	gt(a, b)

```
from operator import *     #  импортируем все функции

print(add(10, 20))       >>> 30     #  сумма
print(floordiv(20, 3))   >>> 6      #  целочисленное деление
print(neg(9))            >>> -9     #  смена знака
print(lt(2, 3))          >>> True   #  проверка на неравенство <
print(lt(10, 8))         >>> False  #  проверка на неравенство <
print(eq(5, 5))          >>> True   #  проверка на равенство ==
print(eq(5, 9))          >>> False  #  проверка на равенство ==
```

```
from functools import reduce
import operator

words = ['Testing ', 'shows ', 'the ', 'presence', ', ', 'not ', 'the ', 'absence ', 'of ', 'bugs']
numbers = [1, 2, -6, -4, 3, 9, 0, -6, -1]

opposite_numbers = list(map(operator.neg, numbers))    #  смена знаков элементов списка
concat_words = reduce(operator.add, words)             #  конкатенация элементов списка

print(opposite_numbers)   >>> [-1, -2, 6, 4, -3, -9, 0, 6, 1]
print(concat_words)       >>> Testing shows the presence, not the absence of bugs
```

Модуль operator реализован на языке C, поэтому функции этого модуля работают в разы быстрее, чем самописные функции в Python.

строковые методы в виде функций можно получить через название типа str.

```
pets = ['alfred', 'tabitha', 'william', 'arla']
chars = ['x', 'y', '2', '3', 'a']

uppered_pets = list(map(str.upper, pets))
capitalized_pets = list(map(str.capitalize, pets))
only_letters = list(filter(str.isalpha, chars))

print(uppered_pets)   >>> ['ALFRED', 'TABITHA', 'WILLIAM', 'ARLA']
print(capitalized_pets)   >>> ['Alfred', 'Tabitha', 'William', 'Arla']
print(only_letters)   >>> ['x', 'y', 'a']
```

Функции isinstance и type для проверки типов данных

функция isinstance() - выполняет проверку на принадлежность объекта определенным типам данных

```
a = 5
isinstance(a, int)
>>> True

isinstance(a, float)
>>> False
```

Но есть один нюанс, связанный с булевым типом данных.

```
b = True
isinstance(b, bool)
>>> True
isinstance(b, int)
>>> True.
```

Это связано с особенностью реализацией типа bool.

Если нужна строгая проверка на типы, то лучше использовать функцию type():

```
type(b) == bool
type(b) is bool
type(b) is int
type(b) in (bool, float, str)  # проверяем переменную b на три типа данных.
```

isinstance() в отличие от type() делает проверку с учетом иерархии наследования объектов и была разработана для проверки принадлежности объекта тому или иному классу:

Например, тип bool наследуется от int, поэтому isinstance() выдает True для обоих типов, когда b – булева переменная.
А функция type() даст True только для типа bool.
То есть, здесь проверка происходит без учета иерархии.

Предположим, что у нас есть кортеж с произвольными данными:

```
data = (4.5, 8.7, True, "книга", 8, 10, -11, [True, False])
```

Нужно вычислить сумму всех вещественных чисел этой коллекции.

```
s = 0
for x in data:
    if isinstance(x, float):
        s += x
print(s)
```

Или с использованием функции filter():

```
s = sum(filter(lambda x: isinstance(x, float), data))
```

(работает быстрее, так как используются встроенные функции вместо цикла for)

попробуем вычислить сумму целочисленных значений, просто изменив тип данных:

```
s = sum(filter(lambda x: isinstance(x, int), data))
>>> 8
```

так как в коллекции data присутствует булево значение True, которое интерпретируется как целое число 1.

Здесь лучше применять строгую проверку с использованием функции type():

```
s = sum(filter(lambda x: type(x) is int, data))
>>> 7.
```

С помощью функции isinstance() можно делать и множественные проверки.

```
a = 5.5
isinstance(a, (int, float))
```

Это эквивалентно записи:

```
isinstance(a, int) or isinstance(a, float)
```

Но первый вариант короче и потому чаще используется на практике.

Функции all и any

функция all()

на вход принимает итерируемый объект и все его значения приводит к булевым величинам:
Возвращает True, Если **все значения** входящей последовательности равны True.
Возвращает False, если же, хотя бы одно значение принимает False, то на выходе будет False

all(iterable)
iterable - любой итерируемый объект: (список; кортеж; строка; множество; словарь и т.д.)

```
print(all([True, True, True]))     #  возвращает True, так как все значения списка равны True
print(all([True, True, False]))    #  возвращает False, так как не все значения списка равны True
```

в Python все следующие значения приводятся к значению False:

- константы None и False;
- нули всех числовых типов данных: 0, 0.0, 0j, Decimal(0), Fraction(0, 1);
- пустые коллекции: '', (), [], {}, set(), range(0).

```
print(all([1, 2, 3]))     >>> True
print(all([1, 2, 3, 0, 5]))  >>> False
print(all([True, 0, 1]))    >>> False
print(all(('', 'red', 'green')))   >>> False
print(all({0j, 3+4j}))   >>> False
```

Работу этой функции можно повторить с помощью обычных булевых операций.

```
all_res = True
for x in a:
    all_res = all_res and bool(x)
print(all_res)
```

переменная all_res сохранит начальное значение True только в том случае, если не встретится ни один False. (True and False = False)
(В Python, лучше использовать all() – это и удобнее и быстрее.)

При работе со словарями функция all() проверяет на соответствие True ключи словаря.

```
dict1 = {0: 'Zero', 1: 'One', 2: 'Two'}
dict2 = {'Zero': 0, 'One': 1, 'Two': 2}
print(all(dict1))   >>> False
print(all(dict2))   >>> True
```

если переданный итерируемый объект пустой, то функция all() возвращает True.

```
print(all([]))        >>> True   #  передаем пустой список
print(all(()))        >>> True   #  передаем пустой кортеж
print(all(''))        >>> True   #  передаем пустую строку
print(all([[], []]))  >>> False  #  передаем список, содержащий пустые списки
```

функция any()

возвращает True, если **хотя бы один элемент** переданной ей последовательности приводится к значению True, и False в противном случае.
any(iterable)
iterable - любой итерируемый объект (список; кортеж; строка; множество; словарь и т.д.)

```
print(any([False, True, False]))    #  возвращает True, так как есть хотя бы один элемент, равный True
print(any([False, False, False]))   #  возвращает False, так как нет элементов, равных True
```

```
print(any([0, 0, 0]))      >>> False
print(any([0, 1, 0]))      >>> True
print(any([False, 0, 1]))      >>> True
print(any(['', [], 'green']))  >>> True
print(any({0j, 3+4j, 0.0}))    >>> True
```

При работе со словарями функция any() проверяет на соответствие True ключи словаря.

```
dict1 = {0: 'Zero'}
dict2 = {'Zero': 0, 'One': 1}

print(any(dict1))   >>> False
print(any(dict2))   >>> True
```

Повторим также работу этой функции через булевы операции.

```
any_res = False
for x in a:
    any_res = any_res or bool(x)
print(any_res)
```

Если встретится хотя бы один True, то оператор or вернет True и оно сохранится в переменной any_resю

если переданный объект пуст, то функция any() возвращает значение False.

```
print(any([]))        >>> False  #  передаем пустой список
print(any(()))        >>> False  #  передаем пустой кортеж
print(any(''))        >>> False  #  передаем пустую строку
print(any([[], []]))  >>> False  #  передаем список, содержащий пустые списки
```

Пусть мы делаем игру «Крестики-нолики». Определим, есть ли выигрышная позиция, например, у крестиков.
Все поле из девяти клеток представим одномерным списком :

```
P = ['x', 'x', 'o', 'o', 'x', 'o', 'x', 'x', 'x']

def true_x(a):
    return a == 'x'

row_1 = all(map(true_x, P[:3]))  # результат для первой строки
row_2 = all(map(true_x, P[3:6]))
row_3 = all(map(true_x, P[6:]))
print(row_1, row_2, row_3)   >>> False False True
# map() преобразует крестики в True, а нолики – в False.
# Далее для каждой строки: True – есть выигрышная комбинация; False – нет выигрышной комбинации.
```

По аналогии можно сделать проверку выигрыша по столбцам:

```
col_1 = all(map(true_x, P[::3]))
col_2 = all(map(true_x, P[1::3]))
col_3 = all(map(true_x, P[2::3]))
```

Функции all() и any() в связке с функцией map()

Функции all() и any() могут быть полезны в комбинации с функцией map(), которая может преобразовывать элементы последовательности (итерируемого объекта) к значению True/False в соответствии с неким условием.

все ли элементы списка numbers больше 10:

```
numbers = [17, 90, 78, 56, 231, 45, 5, 89, 91, 11, 19]
result = all(map(lambda x: x > 10, numbers))
>>> False (т.к. список numbers содержит число 5, которое не больше чем 10).
```

проверяем, что **хотя бы один элемент** списка четное число:

```
numbers = [17, 91, 78, 55, 231, 45, 5, 89, 99, 11, 19]
result = any(map(lambda x: x % 2 == 0, numbers))
if result:
    print('Хотя бы одно число четное')
else:
    print('Все числа нечетные')
>>> Хотя бы одно число четное  # (так как список numbers содержит четное число 78.)
```

Реализация встроенных функций all() и any() выглядит *примерно* так:

```
def all(iterable):
    for item in iterable:
       if not item:
           return False
    return True
```

```
def any(iterable):
    for item in iterable:
        if item:
            return True
    return False
```

Функция enumerate()

возвращает последовательность кортежей из индекса элемента и самого элемента переданной ей последовательности.
enumerate(iterable, start=0)
iterable - любой итерируемый объект (список; кортеж; строка; множество; словарь и т.д.)
start - начальное значение индекса. (По умолчанию start = 0)

функция enumerate() возвращает итератор

```
colors = ['red', 'green', 'blue']
pairs = enumerate(colors)

print(pairs)   >>> <enumerate object at 0x...>
print(list(pairs))   >>> [(0, 'red'), (1, 'green'), (2, 'blue')]
```

```
for pair in enumerate(colors, 100):
    print(pair)
>>>
(100, 'red')
(101, 'green')
(102, 'blue')
```

распаковка кортежей при итерировании с помощью цикла for.

```
colors = ['red', 'green', 'blue']
for index, item in enumerate(colors):
    print(index, item)
```

```
>>>
0 red
1 green
2 blue
```

Основные парадигмы программирования

Основные парадигмы программирования:
императивное программирование;
структурное программирование;
объектно-ориентированное программирование;
функциональное программирование;
логическое программирование.

Императивное программирование (ИП) характеризуется тем, что:

- в исходном коде программы записаны инструкции (команды);
- инструкции должны выполняться последовательно;
- данные, полученные при выполнении инструкции, могут записываться в память;
- данные, получаемые при выполнении предыдущих инструкций, могут читаться из памяти последующими инструкциями.

Императивная программа похожа на приказы (англ. imperative — приказ, повелительное наклонение), выражаемые повелительным наклонением в естественных языках. Это последовательность команд, выполняемых процессором.

При императивном подходе к составлению кода широко используется присваивание. Наличие операторов присваивания увеличивает сложность модели вычислений и создает условия для специфических ошибок императивных программ.

Основные механизмы управления:

- последовательное исполнение команд;
- использование именованных переменных;
- использование оператора присваивания;
- использование ветвления (оператор if);
- использование безусловного перехода (оператор goto).

Ключевой идеей императивного программирования является работа с переменными, как с временным хранением данных в оперативной памяти.

Структурная парадигма программирования нацелена на сокращение времени разработки и упрощение поддержки программ за счёт использования блочных операторов и подпрограмм. Отличительная черта структурных программ — отказ от оператора безусловного перехода (goto), который широко использовался в 1970-х годах.

Основные механизмы управления:

- последовательное исполнение команд;
- использование именованных переменных;
- использование оператора присваивания;
- использование ветвления (оператор if);
- использование циклов;
- использование подпрограмм (функций).

В структурном программировании программа по возможности разбивается на маленькие подпрограммы (функции) с изолированным контекстом.

Парадигму структурного программирования предложил нидерландский ученый Эдсгер Дейкстра.

В объектно-ориентированной парадигме программа разбивается на объекты – структуры данных, состоящие из полей, описывающих состояние, и методов – функций, применяемых к объектам для изменения или запроса их состояния.

Объектно-ориентированную парадигму программирования поддерживают:

- Python;
- C#;
- Java;
- C++;
- JavaScript;
- и другие.

Основные механизмы управления:

- абстракция;
- класс;
- объект;
- полиморфизм;
- инкапсуляция;
- наследование.

При использовании логического программирования программа содержит описание проблемы в терминах фактов и логических формул, а решение проблемы система находит с помощью механизмов логического вывода.

В конце 60-х годов XX века Корделл Грин предложил использовать резолюцию как основу логического программирования. Алан Колмеро создал язык логического программирования Prolog в 1971 году. Логическое программирование пережило пик популярности в середине 80-х годов XX века, когда было положено в основу проекта разработки программного и аппаратного обеспечения вычислительных систем пятого поколения.

Важное его преимущество — достаточно высокий уровень машинной независимости, а также возможность

откатов, возвращения к предыдущей подцели при отрицательном результате анализа одного из вариантов в процессе поиска решения.

Один из концептуальных недостатков логического подхода — специфичность класса решаемых задач.

Недостаток практического характера — сложность эффективной реализации для принятия решений в реальном времени, скажем, для систем жизнеобеспечения.

Основной инструмент **функционального программирования** (ФП) — математические функции.

Математические функции выражают связь между исходными данными и итогом процесса. Процесс вычисления также имеет вход и выход, поэтому функция — вполне подходящее и адекватное средство описания вычислений. Именно этот простой принцип положен в основу функциональной парадигмы программирования.

Функциональное программирование (ФП) — декларативная парадигма программирования.

Функциональная программа — набор определений функций. Функции определяются через другие функции или рекурсивно через самих себя. При выполнении программы функции получают аргументы, вычисляют и возвращают результат, при необходимости вычисляя значения других функций.

Как преимущества, так и недостатки данной парадигмы определяет модель вычислений без состояний. Если императивная программа на любом этапе исполнения имеет состояние, то есть совокупность значений всех переменных, и производит побочные эффекты, то чисто функциональная программа ни целиком, ни частями состояния не имеет и побочных эффектов не производит. То, что в императивных языках делается путем присваивания значений переменным, в функциональных достигается передачей выражений в параметры функций. В результате чисто функциональная программа не может изменять имеющиеся данные, а может лишь порождать новые копированием и/или расширением старых. Следствие того же — отказ от циклов в пользу рекурсии.

Сильные стороны функционального программирования:

- повышение надёжности кода;
- удобство организации модульного тестирования;
- возможности оптимизации при компиляции;
- возможности параллелизма.

Недостатки: отсутствие присваиваний и замена их на порождение новых данных приводят к необходимости постоянного выделения и автоматического освобождения памяти, поэтому в системе исполнения функциональной программы обязательным компонентом становится высокоэффективный сборщик мусора.

Основные идеи функционального программирования:

- **неизменяемые переменные** — в функциональном программировании можно определить переменную, но изменить ее значение нельзя;
- **чистая функция** — это функция, результат работы которой предсказуем. При вызове с одними и теми же аргументами, такая функция всегда вернет одно и то же значение. Про такие функции говорят, что они не вызывают побочных эффектов;
- **функции высшего порядка** — могут принимать другие функции в качестве аргумента или возвращать их;
- **рекурсия** — поддерживается многими языками программирования, а для функционального программирования обязательна. Дело в том, что в языках ФП отсутствуют циклы, поэтому для повторения операций служит рекурсия. Использование рекурсии в языках ФП оптимизировано, и происходит быстрее, чем в языках императивного программирования;
- **лямбда-выражения** — способ определения анонимных функциональных объектов.

Термин «парадигма программирования» впервые применил в 1978 году Роберт Флойд.

В основе императивных, структурных, объектно-ориентированных языков программирования лежит машина Тьюринга, разработанная Аланом Тьюрингом.

В основе функциональных языков программирования лежит модель лямбда-исчислений, разработанная Алонзо Чёрчем.