

Основные понятия модуля

Python — это высокоуровневый интерпретируемый кроссплатформенный язык программирования.

Актуальная версия Python — **Python 3**

[Документация по Python](#) 

Переменная — именованная область памяти компьютера, адрес которой позволяет получить доступ к данным.

Алгоритм — набор последовательных действий, направленных на достижение поставленной цели или решение конкретной задачи.

Функция — фрагмент кода, к которому можно обратиться из любого другого места.

Функции, как правило, возвращают некоторое значение в качестве результата работы.

Аргументы — данные, которые необходимы функции для её работы.

Python — язык с **неявной сильной динамической типизацией**.

Динамическая типизация

Тип переменной определяется во время выполнения программы

Сильная типизация

Нельзя совершать операции над объектами разного типа без приведения их к одному типу

Неявная типизация

Не надо указывать тип переменной при её объявлении

Строки — неизменяемый тип данных, предназначенный для хранения текстовой информации.

Ввод и вывод информации

<code>print(аргумент)</code>	Печатает на экране данные, которые мы передали. Это может быть переменная или выражение.
<code>print(аргумент_1, аргумент_2, ... , аргумент_n)</code>	Печатает переданные значения через пробел.
<code>input(подсказка_для_пользователя)</code>	<p>Функция для ввода информации от пользователя.</p> <p>В круглых скобках можем написать строку-подсказку для пользователя о том, какую именно информацию мы ожидаем от него получить.</p>

Codeboard

RUN	Запускает выполнение кода. Так вы можете проверить, как работает ваш код до отправки решения.
TEST	Запускает тесты для вашего решения. Таким образом проверяется логика решения.
SUBMIT	Отправляет код на проверку и запускает дополнительные тесты => результат и начисление баллов (при верном решении).

Присваивание

<code>a = 5</code>	Переменной <code>a</code> присвоили значение 5.
<code>b = a</code>	Переменной <code>b</code> присвоили значение переменной <code>a</code> .
<code>a, b = 5, 6</code>	Множественное присваивание: переменной <code>a</code> присвоили значение 5, переменной <code>b</code> присвоили значение 6.

Правила именования переменных

- Название переменной должно состоять только из букв, цифр и знаков подчёркивания `_`.
- Название переменной не может начинаться с цифры.

Типы данных

Тип данных	Изменяемость	Класс	Пример
Целые числа	-	int	73 0
Числа с плавающей точкой	Нет	float	3.14 -2.79
Строки	Нет	str	"Hello, world!" "5"
Логические переменные	Нет	bool	True False
Списки	Да	list	[1,2,3,4]
Кортежи	Нет	tuple	('a','b','c')
Словари	Да	dict	{ 'a' : 1, 'b' : 2 }
Множества	Да	set	{ 'a', 1, 'b', 2 }

Определение типа переменной и идентификатора объекта

- `type(n)` — тип переменной `n`.
- `id(n)` — уникальный идентификатор объекта, который хранится в переменной `n`.

Операции с целыми и вещественными числами

Сложение	+	$7+5 = 12$ $3.14+1 = 4.14$
Вычитание	-	$7-5 = 2$ $3.14-1 = 2.14$
Умножение	*	$7*5 = 35$ $3.14*2 = 6.28$
Возведение в степень	**	$7**5 = 16807$ $3.14**2 = 9.8596$
Деление	/	$5/2 = 2.5$ $3.14/2 = 1.57$
Целочисленное деление	//	$7 // 5 = 1$ $3.14 // 2 = 1.0$
Остаток от деления	%	$7 \% 5 = 2$ $3.14 \% 2 = 1.14$

Округление чисел

- `round(значение, количество_знаков_после_запятой)` — округляет число к заданной точности.

Значения логического типа данных

- `True` — Истина
- `False` — Ложь

Строки

<code>s = "Hello!"</code>	Задаём строку
<code>s[начало:конец:шаг]</code>	Срез строки
<code>s = "Hel" + "lo!"</code>	Сложение строк
<code>s = "Hello!"*n</code>	Дублирование значения строки n раз
<code>len(s)</code>	Длина строки
<code>find(substr)</code>	Метод для поиска подстроки в строке <i>Пример вызова: <code>s.find('e')</code> возвращает индекс символа 'e' в строке s</i>
<code>isdigit()</code>	Метод возвращает <code>True</code> , если строка состоит только из цифр
<code>isalpha()</code>	Метод возвращает <code>True</code> , если строка состоит только из букв
<code>isalnum()</code>	Метод возвращает <code>True</code> , если строка состоит только из букв и цифр
<code>upper()</code>	Метод возвращает новую строку в верхнем регистре
<code>lower()</code>	Метод возвращает новую строку в нижнем регистре
<code>split(разделитель)</code>	Метод разбивает строку на части по разделителю (по умолчанию — пробел) и возвращает результат в виде списка
<code>'строка-разделитель'.join(список)</code>	Метод объединяет элементы списка в строку, вставляя между ними строку-разделитель

Форматирование строк

Форматирование строк используется, когда нам необходимо вставлять в шаблон строки разные данные.

Это можно сделать и с помощью соединения частей строк и данных, но с помощью приёмов форматирования делать это можно гораздо удобнее.

Способ создания форматированной строки	Пример задания шаблона
Метод <code>format()</code>	'The {} currency rate on the date {} is {:.3f}'.format(currency, cur_date, rate)
f-строки	f'The {currency} currency rate on the date {cur_date} is {rate:.3f}'

Список

Создание списка

```
a = []
a = list()
a = [1,2,3]
a = ["h", "hello", "world", "a"]
a = range(0,10)
a = range(0,10, 2)
a = range(10)
list1 = ["hello", 4, 3.5, (1,2), [1,2,3], {"name": "Bob", "age": 33}]
```

Индексы и срезы

```
a[2]
a[-1]
a[:2]
a[1:3]
a[0:3:2]
a[::-1]
```

Работа со списком

Вставка	<code>a.append("new_elem")</code>
Подсчёт	<code>a.count("new_elem")</code>
Копирование	<code>b = a.copy()</code>
Расширение	<code>a.extend(["another", "list", "with", "some", "elements"])</code>
Реверс	<code>a.reverse(), a[::-1]</code>
Сортировка	<code>a.sort()</code>
Очистка	<code>a.clear()</code>

Строка — список

```
s = "hello"  
s[0] # 'h'  
s[:2] # 'he'
```

Кортеж

Создание кортежа

```
tpl1 = ()  
tpl1 = tuple()  
tpl1 = (1,2,3)  
tpl1 = (1,)   
tpl1 = ("hello", 4, 3.5, (1,2), [1,2,3], {"name": "Bob", "age": 33})
```

Отличия от списка

Кортежи — неизменяемые братья для списков. Можно использовать в ключах для словаря.

Словарь

Создание словаря

```
dict1 = dict()  
dict1 = {}  
dict1 = {"name": "Bob", (1,2): 3, 3.5: [1,2,3]}
```

Работа со словарём

Взятие ключей	dict1.keys()
Взятие значений	dict1.values()

Обращение по ключу	<code>dict1["name"]</code>
Обращение по ключу с дефолтным значением	<code>dict1.get("name", "empty")</code>
Обновление/дополнение словаря	<code>dict1.update({"name": "foo", 4: 3})</code>
Удаление элемента с возвратом	<code>dict1.pop("name")</code>
Взятие/добавление через <code>setdefault</code>	<code>dict1.setdefault("surname", "Bobrov")</code>

Множество

Создание множества

```
set1 = set()
set1 = set([1,2,2,2,2,1])
set1 = {33}
set1 = {33, 22, 33, 1}
set1 = {33, 2.3, "hello", (1,2,3)}
```

Работа со множеством

Добавление элемента	<code>set1.add("new_element")</code>
Удаление элемента	<code>set1.remove("new_element")</code> или <code>set1.discard("new_element")</code>
Объединение множеств	<code>set1.union(set2)</code>
Пересечение множеств	<code>set1.intersection(set2)</code>
Вычитание множеств	<code>set1.difference(set2)</code>
Проверка на входжение во множество	<code>set2.issubset(set1)</code>

Преобразование типов

Вещественное число в целое число	<code>float_to_int = int(3.4)</code>
Целое число в вещественное число	<code>int_to_float = float(3)</code>
Вещественное число в строку	<code>float_to_string = str(3.4)</code>
Целое число в строку	<code>int_to_string = str(3)</code>
Строка в вещественное число	<code>string_to_float = float("3.4")</code>
Строка в целое число	<code>string_to_int = int("3")</code>
Список в кортеж	<code>list_to_tuple = tuple([1,2,3])</code>
Кортеж в список	<code>tuple_to_list = list((1,2,3))</code>

Операторы сравнения

<code><</code>	Меньше
<code>></code>	Больше
<code><=</code>	Меньше или равно
<code>>=</code>	Больше или равно
<code>==</code>	Равенство
<code>!=</code>	Неравенство

Побитовые логические операторы

<code>~ x</code>	Побитовое "НЕ"
<code> </code>	Побитовое "ИЛИ"
<code>^</code>	Побитовое "ИСКЛЮЧИТЕЛЬНОЕ ИЛИ"
<code>&</code>	Побитовое "И"

Логические операторы

<code>not</code>	Логическое НЕ
<code>or</code>	Логическое ИЛИ
<code>and</code>	Логическое И
<code>in, not in</code>	Проверка принадлежности
<code>is, is not</code>	Проверка тождественности

Логическое НЕ

x	not x
True	False
False	True

Логическое И

x	y	x and y
False (0)	False (0)	False (0)
False (0)	True (1)	False (0)
True (1)	False (0)	False (0)
True (1)	True (1)	True (1)

Логическое ИЛИ

x	y	x or y
False (0)	False (0)	False (0)
False (0)	True (1)	True (1)
True (1)	False (0)	True (1)
True (1)	True (1)	True (1)

Условный оператор

```
if Условие:  
    Блок инструкций 1  
else:  
    Блок инструкций 2
```

if-elif-else

```
if Условие 1:  
    Блок инструкций 1  
elif Условие 2:  
    Блок инструкций 2  
else:  
    Блок инструкций 3
```

Инлайновый if

```
A = 5 if X else -5
```

Общая терминология циклов

Тело цикла — это набор команд, находящихся на одном (четыре пробела) и более отступе от отступа самого цикла.

Другими словами, **тело цикла** — это те команды, которые находятся внутри него и будут повторяться.

Итерация — это один шаг цикла, повторное применение операции, прописанной в теле цикла.

В Python предусмотрены две встроенные конструкции, которые организуют цикл: **for** и **while**.

Цикл for

Для работы с циклом **for** используется следующая конструкция:

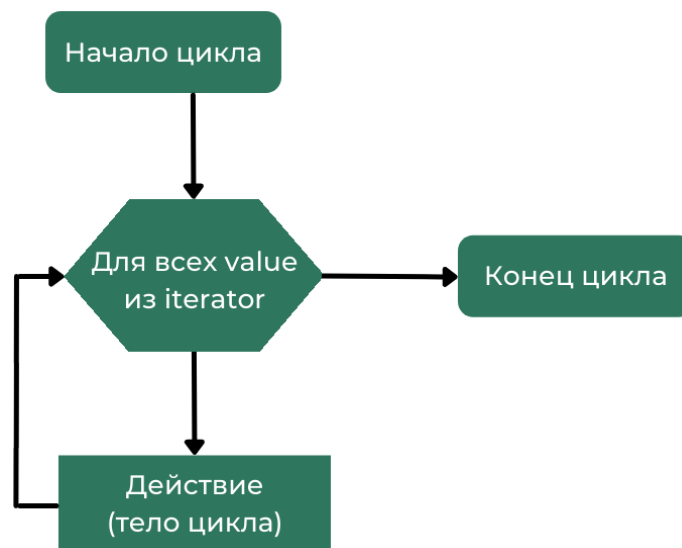
```
for value in iterator:
    # Начало блока кода с телом цикла
    ...
    ...
    ...
    # Конец блока кода с телом цикла
# Код, который будет выполняться после
цикла
```

Здесь:

- **for** — ключевое слово, с которого начинается цикл, отправная точка.
- **value** — переменная цикла (может иметь другое название), в которой на каждом шаге цикла (итерации) содержится текущее значение из итерируемого объекта **iterator**.
- **in** — оператор принадлежности, который указывает, откуда берутся значения для переменной **value**.

→ **iterator** — итерируемый объект, из которого на каждой итерации извлекаются элементы (например, список, словарь, кортеж, строка и т. д.).

Блок-схема цикла for



Алгоритм создания цикла for

1. Определяем итерируемый объект, по которому будем проходить циклом (это может быть список, строка, кортеж, словарь или любой другой объект-итератор).
2. Когда вы создаёте цикл, скорее всего, вам нужно совершить какое-то действие с каждым элементом итерируемого объекта. Можно заранее определить для себя это действие, а можно придумать его при отладке программы.
3. Оборачиваем данную конструкцию в код. Для этого берём шаблон для создания цикла. Придумываем имя для переменной цикла, в которую будем помещать значения из итерируемого объекта.

Пример — последовательный вывод элементов списка на экран:

```
# Определяем итерируемый объект
```

```
my_list= [5, 9, 19]
# Подставляем его в шаблон для цикла и записываем имя переменной цикла
for element in my_list:
    # Указываем необходимые действия в теле цикла
    print('Element', element)
```

Совмещённые операторы

Длинная запись	Сокращённая запись
value = value + a	value += a
value = value - a	value -= a
value = value / a	value /= a
value = value * a	value *= a
value = value ** a	value **= a

Функция range()

Функция `range()` позволяет создавать последовательность целых чисел.

Общий синтаксис:

```
range(START, STOP, STEP)
```

Здесь:

- **START** — число, с которого начинается последовательность. По умолчанию последовательность начинается с 0.
- **STOP** — верхняя граница последовательности. Обязательный аргумент.
Важно! **STOP** не включается в итоговый диапазон чисел.
- **STEP** — шаг последовательности. По умолчанию последовательность идёт с шагом 1.

Пример — список целых чисел от 0 до 5 (не включая 5):


```
list(range(5))
```

Пример — список целых чисел от -4 до 10 (не включая 10):

```
list(range(-4, 10))
```

Пример — список целых чисел от -4 до 10 (не включая 10) с шагом 3:

```
list(range(-4, 10, 3))
```

Способы обхода списка

1. Цикл по элементам списка.

```
weight_of_products = [10, 42.4, 240.1, 101.5, 98, 0.4, 0.3, 15]

max_weight = 100
num = 1

for weight in weight_of_products:
    if weight < max_weight:
        print('Product {}, weight: {} -passenger car'.format(num,
weight))
    else:
        print('Product {}, weight: {} -truck'.format(num, weight))
    num += 1
```

2. Цикл по индексам списка.

```
weight_of_products = [10, 42.4, 240.1, 101.5, 98, 0.4, 0.3, 15]
max_weight = 100
N = len(weight_of_products)
for i in range(N):
    if weight_of_products[i] < max_weight:
        print('Product {}, weight: {} -passenger car'.format(i+1,
weight_of_products[i]))
    else:
        print('Product {}, weight: {} -truck'.format(i+1,
```

```
weight_of_products[i]))
```

Цикл while

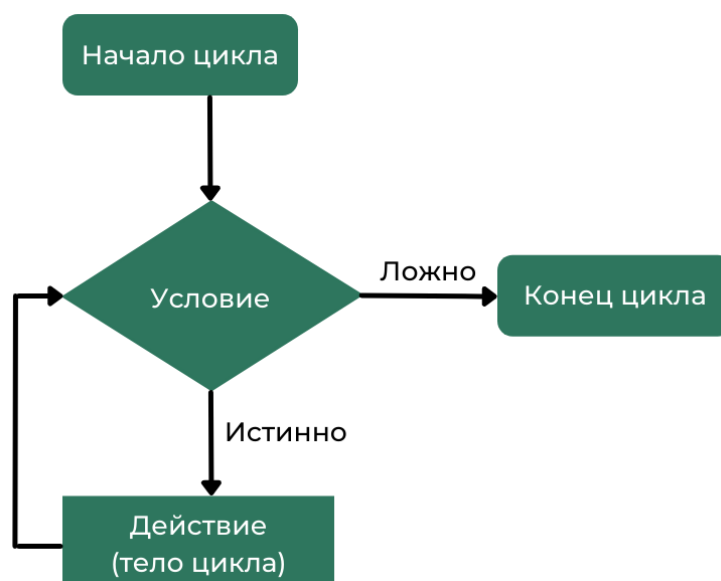
Цикл **while** выполняется до тех пор, пока истинно его условие. Как только оно становится ложным, цикл прерывается.

Условие — это логическое выражение, которое может принимать значение True и False.

Для работы с циклом **while** используется следующая конструкция:

```
while # условие:  
    # Начало блока кода с телом цикла  
    # Пока условие истинно, цикл выполняется  
    ...  
    ...  
    ...  
    # Конец блока кода с телом цикла  
# Код, который будет выполняться после цикла
```

Блок-схема цикла while



Алгоритм создания цикла while

1. Определяем, какое условие будет проверять цикл на каждой итерации. Это может быть любое выражение, которое возвращает булево значение — True или False.
2. Определяем действия, которые мы будем совершать внутри цикла.
3. Оборачиваем эту конструкцию в код. Для этого используем шаблон для создания цикла.

Пример — считаем, сколько раз к x нужно прибавить 2, чтобы x стал больше либо равен y :

```
x = 21
y = 55
# Задаём начальное значение количества итераций
count = 0
# Записываем условное выражение в цикл
while x < y:
    # Увеличиваем значение переменной x на 2
    # Равносильно x = x + 2
    x += 2
    # Увеличиваем количество итераций на 1
    # Равносильно count = count + 1
    count += 1
# Выводим результирующее количество итераций
print('Number of iterations', count)
```

Бесконечный цикл

При работе с циклом `while` нужно быть внимательными, ведь если условие остановки будет всегда True, то цикл никогда не завершится.

Пример:

```
# Плохо
n = 1
```

```
# В данной программе условие всегда True, цикл будет бесконечным
while n < 10:
    print("Hello World")
```

Чтобы остановить выполнение цикла, используется ключевое слово `break`:

```
# Хорошо
n = 1
# В данной программе условие всегда True, цикл будет бесконечным
while True:
    print("New client !")
    n += 1
    # Условие, при достижении которого цикл while будет принудительно
    # завершён
    if n > 10:
        break
```

Особенность использования такого цикла `while` с условием внутри заключается в том, что тело цикла точно выполнится один раз, в отличие от цикла с предусловием. Такой цикл ещё называют **циклом с постусловием**.

Вложенные циклы

Циклы могут быть вложены друг в друга. Такие конструкции могут пригодиться при работе с вложенными структурами.

Пример — вложенный цикл по элементам внешнего и внутренних списков:

```
matrix = [
    [1, 2],
    [3, 4],
    [5, 6]
]
for row in matrix:
    print('Current row', row)
    for elem in row:
        print('Current elem', elem)
    print()
```

Первый цикл называют **циклом по первому уровню вложенности**, второй цикл — **циклом по второму уровню вложенности**. Если уровней вложенности всего два, то удобнее называть первый цикл **внешним**, а второй — **внутренним** (вложенным).

Пример — вложенный цикл по индексам внешнего и внутренних списков:

```
matrix = [  
    [1, 2],  
    [3, 4],  
    [5, 6]  
]  
  
N = len(matrix)  
M = len(matrix[0])  
for i in range(N):  
    print('Current i', i)  
    print('Current row', matrix[i])  
    for j in range(M):  
        print('Current j', j)  
        print('Current elem', matrix[i][j])  
    print()
```

При работе с вложенными циклами стоит учитывать количество итераций.

Количество итераций во вложенном цикле = количество элементов в первой итерируемой последовательности * количество элементов во второй итерируемой последовательности

Чем больше итераций, тем больше времени необходимо на обработку кода.

Старайтесь избегать использования вложенных циклов в задачах, где в них нет необходимости.

Пример — подсчитать сумму элементов в первом столбце матрицы:

```
# Плохой способ:  
table = [  
    [1, 3, 6],
```

```
[4, 6, 8],
[10, 33, 53]
]
N = len(table)
M = len(table[0])
S = 0
count = 0
for i in range(N):
    for j in range(M):
        if j == 0:
            S += table[i][j]
            count += 1

# Лучше:
table = [
    [1, 3, 6],
    [4, 6, 8],
    [10, 33, 53]
]
N = len(table)
S = 0
count = 0
for i in range(N):
    S += table[i][0]
    count += 1
```

Функция `enumerate()`

Функция `enumerate()` часто используется в цикле `for` и позволяет выдавать одновременно индекс элемента последовательности (порядковый номер, начиная от 0), а также сам элемент итерируемого объекта.

Синтаксис такой конструкции будет следующим:

```
for index, value in enumerate(iterator):
    ...
```

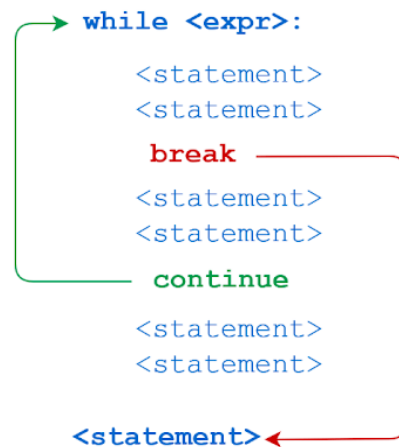
На каждой итерации в переменную `index` заносится индекс текущего элемента из `iterator`, а в `value` — значение текущего элемента.

Пример — одновременный проход по элементам списка и их индексам:

```
my_list = [5, 9, 13]
for index, value in enumerate(my_list):
    print(index, value)
```

Break

Любой цикл, встречая ключевое слово `break`, преждевременно заканчивает своё выполнение и переходит к основному коду программы. Ниже приведён пример для цикла `while`:



Python "while" Loops
(Indefinite Iteration) – Real
Python

Пример — добавление элементов в инвентарь, размер инвентаря — 3:

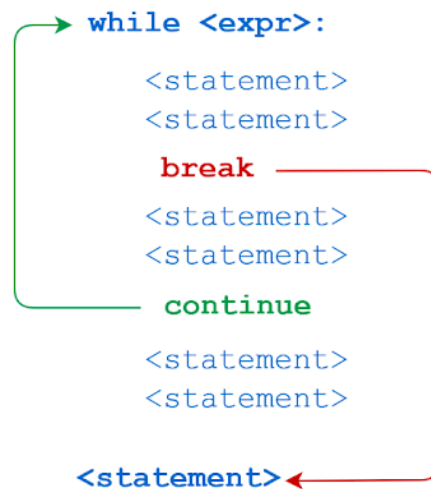
```
to_inventory = ['Blood Moon Sword', 'Sunset-colored sword', 'Bow of Stars', 'Gain Stone']
inventory = []

for item in to_inventory:
    if len(inventory) == 3:
        print('Inventory is full!')
        break
    else:
        inventory.append(item)
```

Continue

Если в теле цикла встречается ключевое слово `continue`, то цикл пропускает весь код до конца тела цикла и переходит на следующий шаг.

Ниже приведён пример для цикла `while`:



Python "while" Loops
(Indefinite Iteration) – Real
Python

Пример:

```
client_status = {
    103303: 'yes',
    103044: 'no',
    100423: 'yes',
    103032: 'no',
    103902: 'no'
}

for user_id in client_status:
    if client_status[user_id] == 'no':
        continue
    else:
        print('Send present user', user_id)
```


Примечание. Ключевые слова `break` и `continue` могут использоваться как в цикле `while`, так и в цикле `for`.

Pass

Помимо ключевых слов `break` и `continue`, существует ключевое слово `pass`. Заглушка `pass` означает «ничего не делать». Обычно мы используем её, так как Python не позволяет создавать класс, функцию, цикл или оператор `if` без кода внутри.

Пример — цикл, в котором ничего не происходит, если элемент списка больше 3:

```
lst = [1,2,3,4,5]
for elem in lst:
    if elem > 3:
        pass
    else:
        print(i)
```

Создание функции

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

Проверка аргументов

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

Аргументы по умолчанию

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

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

Получение переменного числа аргументов

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

...

Передача аргументов

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

Lambda-функции

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

Вложенные функции

Вложенной называется функция, которая объявлена в теле другой функции.

Пример:

```
def outer():
    print('Called outer function')
    def inner():
        print('Called inner function')
    inner()

outer()
## Будет напечатано:
## Called outer function
## Called inner function
```

Любая функция скрывает, защищает (**инкапсулирует**) доступ к переменным и функциям, объявленным внутри неё. Причина в том, что переменные и функции существуют только в момент вызова самой функции. Как только функция завершается, все переменные и функции, которые были объявлены внутри неё, уничтожаются — просто удаляются из памяти. Для этого в *Python* применяется механизм сборки мусора (*Garbage Collection*).

```
def print_root(value, n=2):
    def root(value, n=2):
        result = value ** (1/n)
        return result
    res = root(value, n)
    print(f'Root of power {n} from {value} equals {res}')

print_root(81, 4)
## Будет напечатано:
## Root of power 4 from 81 equals 3.0

print(root(81, 4))
## Возникнет ошибка:
## NameError: name 'root' is not defined
```

```
print(res)
## Возникнет ошибка
## NameError: name 'res' is not defined
```

Разрешение переменных. Области видимости

Разрешение — это процесс поиска интерпретатором объекта, который скрывается за названием переменной.

В *Python* существует **четыре типа переменных** в зависимости от их видимости. Порядок их разрешения будет идти от пункта 1 до пункта 4.

1. **Локальные переменные (local)** — это имена объектов, которые были объявлены в функции и используются непосредственно в ней.

В разряд локальных переменных также входят аргументы функции.

Важно! Эти объекты существуют только во время выполнения функции, в которой они были объявлены. После завершения работы функции они удаляются из оперативной памяти.

2. **Нелокальные переменные (nonlocal)** — это имена объектов, которые были объявлены во внешней функции относительно рассматриваемой функции.
3. **Глобальные переменные (global)** — это имена объектов, которые были объявлены в основном блоке программы (вне функций).
4. **Встроенные переменные (built-in)** — это имена объектов, которые встроены в функционал *Python* изначально. К ним относятся, например, функции `print`, `len`, структуры данных `list`, `dict`, `tuple` и другие.

Встроенные переменные (Built-in)

Имена объектов, которые встроены в функционал *Python* изначально. К ним относятся, например, функции *print*, *len*, структуры данных *list*, *dict*, *tuple* и другие.

Глобальные переменные (Global)

Имена объектов, которые были объявлены непосредственно в основном блоке программы (вне функций).

Нелокальные переменные (Nonlocal)

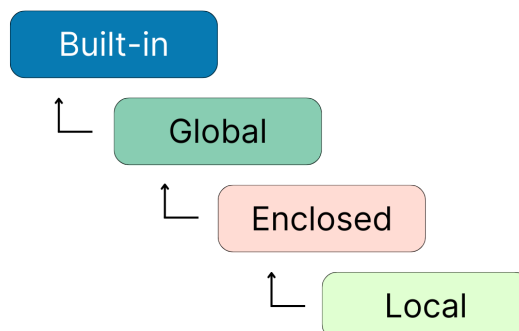
Имена объектов, которые были объявлены во внешней функции относительно рассматриваемой функции.

Локальные переменные (Local)

Имена объектов, которые были объявлены в функции и используются непосредственно в ней. В разряд локальных переменных также входят аргументы функции.

Когда интерпретатор встречает в коде неизвестное имя (имя переменной или функции), он начинает искать имя в локальной области видимости, затем — в нелокальной, затем — в глобальной и наконец — во встроенной.

Часто нелокальную область видимости называют объемлющей (*enclosed*). Поэтому правило, по которому происходит поиск имени объекта среди областей видимости (разрешение), часто именуют правилом *LEGB* (*Local* → *Enclosed* → *Global* → *Built-in*).



Функция **не может** изменить значение переменной, которая находится вне своей локальной области видимости. Вместо изменения значения глобальной переменной создаётся новая локальная переменная с тем же именем.

```
count = 10
def function():
    count = 100
function()
print(count)

## Будет выведено:
## 10
```

Однако функция может скорректировать объект изменяемого типа, находящийся за пределами её локальной области видимости, если изменит его содержимое.

```
words_list = ['foo', 'bar', 'baz']
def function():
    words_list[1] = 'quux'
function()
print(words_list)

## Будет выведено:
## ['foo', 'quux', 'baz']
```

Изменение значений глобальных переменных. Объявление `global`

Если в коде функции происходит переопределение глобальной переменной, то необходимо просто указать, что та или иная переменная является глобальной. Для этого используются ключевое слово `global`:

```
global_count = 0

def add_item():
    global global_count
    global_count = global_count + 1

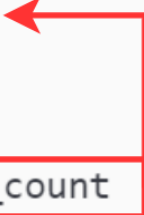
add_item()
print(global_count)
## Будет выведено:
```

```
## 1
```

Выражение `global` указывает на то, что, пока выполняется функция `add_item()`, ссылки на имя `global_count` будут вести к переменной `global_count`, объявленной в глобальной области видимости.

```
global_count = 0

def add_item():
    global global_count
    global_count = global_count + 1
```



Указываем, что `global_count` объявлена в глобальной области видимости

Изменение значений нелокальных переменных. Объявление `nonlocal`

Если в коде функции происходит **переопределение** нелокальной переменной, то необходимо просто указать, что та или иная переменная является нелокальной. Для этого используются ключевое слово `nonlocal`:

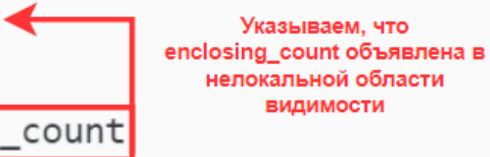
```
def outer():
    enclosing_count = 0
    def inner():
        nonlocal enclosing_count
        enclosing_count = enclosing_count + 1
    print(enclosing_count)
    inner()

outer()

## Будет напечатано:
## 1
```

После выражения `nonlocal enclosing_count`, когда `inner()` ссылается на `enclosing_count`, мы будем обращаться к `enclosing_count` в ближайшей объемлющей области, чье определение дано внутри `outer()`.


```
def outer():  
    enclosing_count = 0  
    def inner():  
        nonlocal enclosing_count  
        enclosing_count = enclosing_count + 1  
        print(enclosing_count)  
    inner()
```



Указываем, что
enclosing_count объявлена в
нелокальной области
видимости

Изменение значений встроенных переменных

Python позволяет создавать переменные с именами идентичными именам встроенных в Python объектов.

Однако, делать этого крайне не рекомендуется, так как это может привести к ошибкам при дальнейших попытках программы обратиться к встроенному имени.

Пример (весь код является частью одной программы):

```
my_list = [1,4,5,7]  
len = len(my_list)  
print(len)  
## Будет выведено:  
## 4  
  
new_list = ['Ivan', 'Sergej', 'Maria']  
length = len(new_list)  
print(length)  
  
## Возникнет ошибка:  
## TypeError: 'int' object is not callable
```

Определение рекурсии

Рекурсия в программировании — функция, которая вызывает саму себя в своем теле.

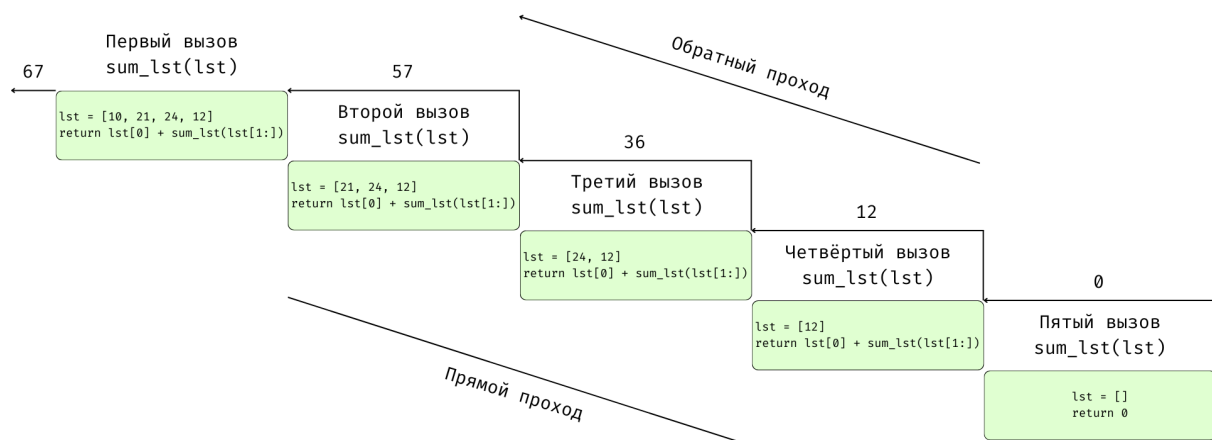
У рекурсивной функции должно быть прописано условие, при выполнении которого функция перестает вызывать сама себя. Такое условие будем называть **условием выхода из рекурсии** (его ещё называют базовым случаем).

Пример №1 (сумма элементов в списке):

```
def sum_lst(lst):
    print(lst)
    if len(lst) == 0:
        return 0
    return lst[0] + sum_lst(lst[1:])
```

```
my_lst = [10, 21, 24, 12]
print(sum_lst(my_lst))
```

```
## Будет выведено:
## [10, 21, 24, 12]
## [21, 24, 12]
## [24, 12]
## [12]
## []
## 67
```

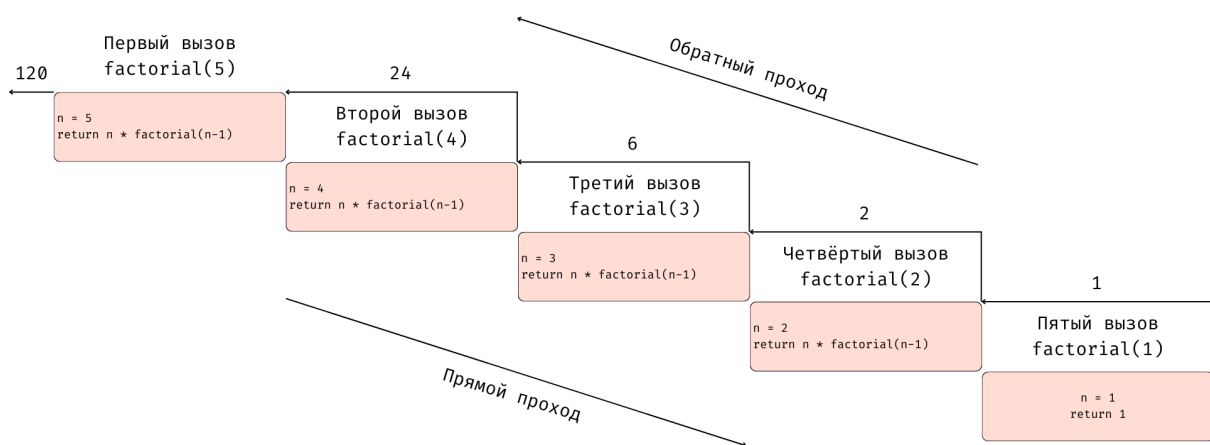


Пример №2 (факториал):

```
def factorial(n):
    if n==0: return 1
    if n==1: return 1
    return n * factorial(n-1)

print(factorial(0))
print(factorial(3))
print(factorial(5))

## Будет напечатано:
## 1
## 6
## 120
```



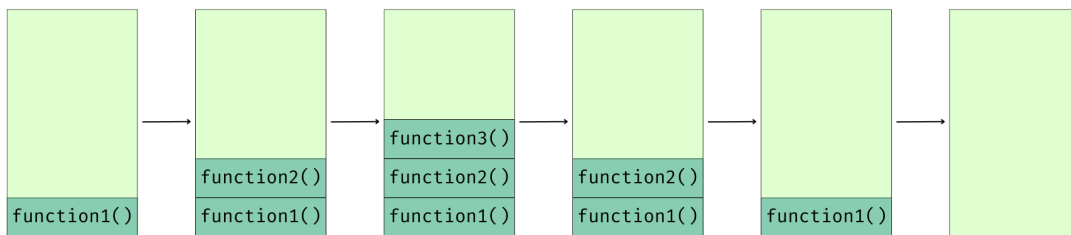
Стек вызова функций

Стек вызова функций — структура данных, хранящая информацию об вызванных в программе функциях в виде адресов возврата функций.

В стеке действует правило *LIFO* (англ. *Last In, First Out*), «последний вошёл, первый вышел».

Ниже представлена схема работы стека вызова трёх функций, которые последовательно вызывают друг друга.

Стек вызова функций



Глубина рекурсии

Число единовременно ожидающих выполнения вызовов рекурсивной функции в стеке называется **глубиной рекурсии**.

Проще говоря, глубина рекурсии — это длина стека вызова.

Когда глубина рекурсии оказывается слишком большой (превышает максимальный размер стека функций), возникает ошибка `RecursionError`.

К сожалению, интерпретатор не может понять, есть ли условие выхода в коде функции, поэтому, когда в его стеке собирается слишком много вызовов той же самой функции, он возвращает ошибку. Это защита от бесконечного выполнения программы.

Этот порог индивидуален и может зависеть от сложности самой функции, версии *Python* и других настроек.

```
print(len(str(factorial(969))))
```

```
## Будет выведено:
```

```
## 2475
```

```
print(len(str(factorial(970))))
```

```
## Возникнет ошибка:
```

```
## RecursionError: maximum recursion depth exceeded in comparison
```

Глубину рекурсии можно увеличить самостоятельно, если для решения задачи крайне необходимо использовать рекурсию большей глубины.

Для этого нам необходимо импортировать модуль `sys`, в котором содержатся функции для управления вашей системой. Из него нам понадобится функция `setrecursionlimit()`, в аргументы которой нужно передать желаемую максимальную глубину рекурсии.

```
import sys
sys.setrecursionlimit(1000000000)
print(len(str(factorial(970))))

# Будет напечатано:
# 2478
```

Цикл vs рекурсия

Важно учитывать, что рекурсии работают медленнее, чем аналогичные по сути циклы. К тому же существует ряд случаев, когда преобразование рекурсии в цикл затруднительно и неэффективно, например, реализация алгоритма дерева решений, быстрое преобразование Фурье или алгоритм *quicksort*. Однако, когда это оказывается возможным, цикл работает быстрее.

Функция как объект

Функция, как и всё в *Python* (кроме ключевых слов, таких как `if`, `def`, `while` и т. д.), — это объект.

→ Во-первых, точно так же, как и любой другой объект (числа, строки, списки и т. д.), мы можем положить функцию в переменную.

```
p = print
p('Hello world!')
```

```
## Будет выведено:  
## Hello world!
```

- Во-вторых, функцию можно передать в качестве аргумента для другой функции.

```
def apply_func(func, x):  
    return func(x)  
print(apply_func(max, [1, 10, 35, 20, -1]))  
  
## Будет выведено:  
## 35
```

- В-третьих, у функции точно так же, как и любого другого объекта, есть свой тип данных — тип `function`.

```
print(type(apply_func))  
  
## Будет выведено:  
## <class 'function'>
```

Важно! Когда мы обращаемся с функцией как с объектом, мы не пишем скобки после имени функции. Они нужны только тогда, когда мы вызываем функцию и передаем в неё аргументы.

Функция `map`

Функция `map()` позволяет преобразовать каждый элемент итерируемого объекта по заданной функции.

Аргументы функции `map()` следующие:

- первый — функция, которую необходимо применить к каждому элементу;
- второй — итерируемый объект (например, список).

Можно записать это в виде шаблона кода:

```
map(<имя_функции>, <итерируемый_объект>)
```

Пример №1:

```
number_list = [11, 12, 13, 14, 15, 16]
square_number_list = list(map(lambda x: x**3, number_list))
print(square_number_list)

## Будет выведено:
## [1331, 1728, 2197, 2744, 3375, 4096]
```

Пример №2:

```
def calculate_tax(salary):
    if salary < 1000:
        return salary * 0.05
    elif salary < 2000:
        return salary * 0.1
    else:
        return salary * 0.15

salaries = [1500, 2200, 3500, 1200]
taxes = list(map(calculate_tax, salaries))
print(taxes)

## Будет выведено:
## [150.0, 330.0, 525.0, 120.0]
```

Пример №3:

```
data = [('Amanda', 1.61, 51), ('Patricia', 1.65, 61), ('Marcos', 1.91, 101)]
map_func = lambda x: (*x, round(x[2] / (x[1]**2)))
updated_data = list(map(map_func, data))
```

```
print(updated_data)

## Будет выведено:
## [('Amanda', 1.61, 51, 19.7), ('Patricia', 1.65, 61, 22.4), ('Marcos',
1.91, 101, 27.7)]
```

Функция filter

Функция `filter()` позволяет отфильтровать переданный ей итерируемый объект и оставить в нём только те элементы, которые удовлетворяют условию, заданному в виде функции.

Аргументы функции `filter()` следующие:

- первый — функция, которая должна возвращать `True`, если условие выполнено, иначе — `False`;
- второй — итерируемый объект, с которым производится действие.

Можно записать это в виде шаблона кода:

```
filter(<имя_функции>, <итерируемый_объект>)
```

Пример №1:

```
words_list = ["We're", 'in', 'a', 'small', 'village', 'near', 'Chicago',
'My', "cousin's", 'getting', 'married.']
even_list = filter(lambda x: len(x) % 2 == 0, words_list)
print(list(even_list))

## Будет выведено:
## ['in', 'near', 'My', "cousin's", 'married.']
```

Пример №2:

```
data = [
    ("FPW-2.0_D", "Бонус: Тренажер по HTML", 10, 100, 10),
```



```
("FPW-2.0", "Бонус: Тренажер по JavaScript", 9.2, 70, 18),
("FPW-2.0_D", "Бонус: Тренажер по React", 8.5, 66.67, 68),
("FPW-2.0", "Бонусный: IT в современном мире", 8.64, 53.74, 856),
("FPW-2.0", "Бонусный: Введение", 8.73, 56.24, 745),
("FPW-2.0", "Бонус: D1. Знакомство с Django (NEW)", 9.76, 95.24, 21),
("FPW-2.0_D", "Бонус: D2. Модели (NEW)", 9.44, 77.78, 18)
]

def filter_module(module):
    code, name, avg_votes, nessa, count = module
    cond_1 = code == "FPW-2.0"
    cond_2 = nessa >= 70
    cond_3 = count > 50
    return cond_1 and cond_2 and cond_3

filtered_data = list(filter(filter_module, data))
print(filtered_data)

## Будет выведено:
## [('FPW-2.0', 'Бонус: Тренажер по JavaScript', 9.2, 70, 180),
('FPW-2.0', 'Бонусный: IT в современном мире', 8.64, 83.74, 856)]
```

Конвейеры из map и filter

В некоторых случаях необходимо выполнить сразу несколько действий с итерируемыми объектами. Например, вы хотите сначала преобразовать данные, а затем отфильтровать их. Такие преобразования называются **конвейерными**.

Для построения конвейерных преобразований необязательно каждый раз после применения `map()` или `filter()` получать список элементов. Объекты `map` и `filter` можно подставлять в эти же функции `map()` и `filter()`.

Пример №1:

```
words_list = ["We're", 'in', 'a', 'small', 'village', 'near', 'Chicago',
'My', "cousin's", 'getting', 'married.']
filtered_words = filter(lambda x: len(x) >= 5, words_list)
```

```
count_a = map(lambda x: (x, x.lower().count('a')), filtered_words)

print(list(count_a))

## Будет выведено:
## [("We're", 0), ('small', 1), ('village', 1), ('Chicago', 1),
('cousin's', 0), ('getting', 0), ('married.', 1)]
```

Пример №2

```
data = [
    ('Amanda', 1.61, 51),
    ('Patricia', 1.65, 61),
    ('Marcos', 1.91, 101),
    ('Andrey', 1.79, 61),
    ('Nikos', 1.57, 78),
    ('Felicia', 1.63, 56),
    ('Lubov', 1.53, 34)
]

map_func = lambda x: (*x, x[2]/(x[1]**2))
updated_data = map(map_func, data)

filter_func = lambda x: 18.5 <= x[3] <= 25
filtered_data = filter(filter_func, updated_data)

print(list(filtered_data))

## Будет выведено:
## [('Amanda', 1.61, 51, 19.7), ('Patricia', 1.65, 61, 22.4), ('Andrey',
1.79, 61, 19.0), ('Felicia', 1.63, 56, 21.1)]
```