Python

Язык программирования Python

Python – это свободный высокоуровневый объектно-ориентированный язык программирования, появившейся относительно давно (в 1991 году), но получивший огромную популярность в текущее время.

Основным преимуществом языка Python перед другими технологиями программирования является мощная база математических библиотек, в том числе, для решения задач Data Mining. Данный факт признан ведущими университетами и научными организациями мира.

Установка и запуск Python

Скачать python https://www.python.org/downloads/

При установке обязательно выбрать «Add python.exe to PATH».

Запускать в терминале VS Code или командной строке операционной системы находясь в той же папке, что и запускаемая программа

```
py prog_name.py
```

Для перехода между папками в терминале используется команда `cd`

```
cd dir1
```

```
cd ..
```

Версии Python

В данном курсе мы будем опираться на актуальную на 1 марта 2026 года версию 3.14.3 или более старшие версии семейства 3.14.

Python быстро развивается поэтому хорошей документации на русском языке найти практически нельзя. Есть разные варианты переводов официальной документации, но выполнены они преимущественно энтузиастами, а не официальным разработчиком.

Официальная документация для языка Python на английском языке доступна на его официальном сайте.

https://docs.python.org/3/ (на английском языке)

https://docs-python.ru/ (на русском языке)

Язык Python иногда называют «языком с двумерным синтаксисом». Это название происходит от того, что отделение основных конструкций языка от их содержимого осуществляется не за счет фигурных скобок (как, например в C++ и Java) иди конструкций do-end (как, например, в языке Ruby), **а за счет отступов.**

Функция Print() в Python

Функция print() в Python используется для вывода текстовой информации на экран или в консоль. Эта функция может принимать один или несколько аргументов. Одним из обязательных аргументов является строка или объект, который будет выведен.

```
print("Hello, World!")
```

Аргументы print() в Python

```
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

1. sep позволяет задать разделитель, который будет выводиться между элементами, переданными в функцию print(). По умолчанию разделителем является пробел, но с помощью sep пробел можно заменить на другой разделитель, к примеру, на запятую.

2. end позволяет определять символ, который будет добавлен в конец сообщения после вывода. По умолчанию это символ перевода строки. Поменять его можно, к примеру, на точку с запятой.

3. file позволяет перенаправить вывод текста в нужный вам файл. По умолчанию функция выводит текст в консоль, в которой вы и задаёте команды Python. Аргумент file полезен, если нужно сохранить вывод для дальнейшего использования.

4. flush позволяет управлять буферизацией вывода. Если этот аргумент установлен в True, то вывод не будет буферизоваться и будет выводиться немедленно.

Функция Print() в Python

```
print("Hello", "world")
print("Hello", "world", sep="|")
print("Hello", "world", sep="|", end="\n\n")


x = 10
y = 25
print(f"x = {x}, y = {y}")
# x = 10, y = 25
print(f"{x = }, {y = }")
# x = 10, y = 25

# Это комментарий!
```

Основные типы данных

В Python любое значение – это объект. Число, строка, кортеж, булева величина, функция – все это объект. У каждого объекта есть свой тип.
Тип объекта можно узнать при помощи функции `type(объект)`.

int – целые числа

str – строки в кодировке Unicode. Записываются в одинарных или двойных кавычках. Разницы между кавычками нет!

bool – логический тип True или False (на самом деле это целое число: 1 или 0).

float – действительное число.

Одноименные функции позволяют осуществлять преобразования типов.

```
print(type(1))
print(type(1.0))
print(type("1"))
print(type(1==1))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
```

Преобразования типов

Процесс преобразования значения одного типа данных (целые числа, строки, числа с плавающей точкой и т. д.) в другой называется преобразованием типа. В Python есть два вида преобразования – неявное преобразование и явное приведение типов.

Неявное преобразование

```
num_int = 123
num_float = 1.23
num_new = num_int + num_float
print(type(num_int), type(num_float), type(num_new))
```

```
<class 'int'> <class 'float'> <class 'float’>
```

Явное приведение типов

В явном преобразовании программист сам заменяет текущий тип данных объекта на требуемый. Для этого используются встроенные функции, такие как `int(), float(), str()` и т. д., чтобы выполнить явное преобразование типов.

Этот тип преобразования также называется приведением типов, поскольку пользователь приводит (изменяет) тип данных объекта.

Преобразования типов

```
a = int(15)      # a = 15
b = int(3.7)     # b = 3
c = int("4")     # c = 4
e = int(False)     # e = 0
f = int(True)      # f = 1
b = int("a1c")     # Ошибка
c = int("4.7")     # Ошибка
a = float(15)        # a = 15.0
b = float(3.7)       # b = 3.7
c = float("4.7")     # c = 4.7
d = float("5")       # d = 5.0
e = float(False)     # e = 0.0
f = float(True)      # f = 1.0
d = float("abc")   # Ошибка
a = str(False)       # a = "False"
b = str(True)        # b = "True"
c = str(5)          # c = "5"
d = str(5.7)        # d = "5.7"
a = bool("1")    # True
b = bool("")     # False
c = bool(0)      # False
d = bool(1)      # True
```

Простейший ввод

Наряду с выводом на консоль мы можем получать ввод пользователя с консоли, получать вводимые данные. Для этого в Python определена функция `input()`. В эту функцию передается приглашение к вводу. А результат ввода мы можем сохранить в переменную.

```
name = input("Your name: ")
age = input("Your age: ")
print(f"Name: {name}  Age: {age}")


a = float(input("a="))
b = int(input("b="))
print(a,b,  a+b)
```

Простейшие операции с числами

Оператор	Пример	Описание	Тип результата
Унарные + -	+2 -(3)	Не делает ничего! Инвертирует знак числа.	Такой же, как у самого числа
+ - *	2+4 5.2-6.8 2*22	Сложение Вычитание Умножение	Если оба аргумента int, то int. Если хотя бы один float, то float.
/	10/5	Вещественное деление	Всегда float
//	10//5	Целочисленное деление	Если оба аргумента int, то int. Если хотя бы один float, то float.
%	10%6	Остаток от деления	Если оба аргумента int, то int. Если хотя бы один float, то float.
**	2**10	Возведение в степень	Если оба аргумента int, то int. Если хотя бы один float, то float.

Встроенные функции для работы

Функция	Пример	Описание
abs	abs(-5)	Модуль числа
divmod	divmod(10, 3)	Частное и остаток от деления
round	round(123.4567, 2)	Округление (второй аргумент – число знаков после запятой)
sum	sum((1, 2.6, -3.1, 5, 0.4))	Сумма аргументов
pow	pow(5, 3)	Степень (альтернативный вариант)

Строки в Python

Строки в Python 3 – это последовательности символов, записанных в кодировке Unicode. Для задания строки можно использовать одинарные или двойные кавычки. Оба варианта эквивалентны. Единственное отличие, что в двойных кавычках нужно экранировать символ двойной кавычки, а в одинарных – символ одинарной.
Строки относятся к типу str.

При использовании обычных одинарных или двойных кавычек в строках можно использовать спецпоследовательности начинающиеся с обратного слэша, например, символ переноса строки \n, символ табуляции \t и т.д.

Внутри обычных одинарных или двойных кавычек осуществлять переход на новую строку нельзя! Для этого есть конструкция три двойных кавычки.

Базовые операции со строками

Сложение (конкатенация)

```
s1 = "Hello "
s2 = "world!"
s3 = s1 + s2
print(s3) # Hello world!
```

Дублирование (умножение)

```
s1 = "Hello "
s3 = s1*3
print(s3) # Hello Hello Hello
```

Принадлежность строки

```
s = 'a'
if s in 'abcd':
    print('ok')
```

Базовые операции со строками

Определение длины строки

```
l = len('Какая у меня длинна?')
print(l)                                # 20
str1 = 'А у меня?'
print(len(str1))                        # 9
```

Доступ по индексу.
Нумерация символов начинается с нуля и заканчивается длиной строки −1.

```
str1 = 'Привет, мир!'
print(str1[0], str1[1], str1[2], str1[11])
```

Индексацию также можно начинать с конца строки. В таком случае последний символ будет иметь индекс, равный −1, а индекс первого символа будет равен длине строки со знаком −.

```
str1 = 'Привет, мир!'
print(str1[-1], str1[-2], str1[-3], str1[-12])
```

Условные конструкции

```
if Условие1:
    Блок инструкций 1
elif Условие2:
    Блок инструкций 2
else:
    Блок инструкций 3
```

`Блок инструкций 1` будет выполнен, если `Условие1` истинно. Если `Условие1` ложно, а `Условие2` истинно, то будет выполнен `Блок инструкций 2`. Если Оба `Условия` ложны, то будет выполнен `Блок инструкций 3`

elif и else не являются обязательными.

```
x = int(input())
y = int(input())
if x > 0 and y > 0:
    print("Первая четверть")
elif x > 0 and y < 0:
    print("Четвертая четверть")
elif y > 0:
    print("Вторая четверть")
else:
    print("Третья четверть")
```

Операторы сравнения

Как правило, в качестве проверяемого условия используется результат вычисления одного из следующих операторов сравнения:

`<` Меньше — условие верно, если первый операнд меньше второго.
`>` Больше — условие верно, если первый операнд больше второго.
`<=` Меньше или равно.
`>=` Больше или равно.
`==` Равенство. Условие верно, если два операнда равны.
`!=` Неравенство. Условие верно, если два операнда неравны.
Например, условие (`x * x < 1000`) означает «значение x * x меньше 1000», а условие (`2 * x != y`) означает «удвоенное значение переменной x не равно значению переменной y».

Операторы сравнения в Python можно объединять в цепочки (в отличии от большинства других языков программирования, где для этого нужно использовать логические связки), например, `a == b == c` или `1 <= x <= 10`.

Логические операторы

В Python существуют стандартные логические операторы: логическое И, логическое ИЛИ, логическое отрицание.

Логическое И является бинарным оператором (то есть оператором с двумя операндами: левым и правым) и имеет вид `and`. Оператор `and` возвращает `True` тогда и только тогда, когда оба его операнда имеют значение `True`.

Логическое ИЛИ является бинарным оператором и возвращает `True` тогда и только тогда, когда хотя бы один операнд равен `True`. Оператор «логическое `ИЛИ`» имеет вид `or`.

Логическое НЕ (отрицание) является унарным (то есть с одним операндом) оператором и имеет вид `not`, за которым следует единственный операнд. Логическое НЕ возвращает `True`, если операнд равен `False` и наоборот.

```
a,b,c = 2,3,4
print(a<b and b<c or not a>b)
```

Цикл while

Цикл `while` позволяет выполнить одну и ту же последовательность действий, пока проверяемое условие истинно. Условие записывается до тела цикла и проверяется до выполнения тела цикла. Как правило, цикл `while` используется, когда невозможно определить точное значение количества проходов исполнения цикла.

```
i = 1
while i <= 10:
    print(i ** 2)
    i += 1      # i = i + 1
```

После тела цикла можно написать слово `else:` и после него блок операций, который будет выполнен один раз после окончания цикла, когда проверяемое условие станет неверно:

```
i = 1
while i <= 10:
    print(i ** 2)
    i += 1      # i = i + 1
else:
    print(f"{i = }")
```

Последовательные типы данных

В языке Python есть целый ряд последовательных типов данных (последовательностей), имеющих структуру, схожую с массивом (занумерованной последовательностью элементов). Их можно поделить на изменяемые (`mutable`) и неизменяемые (`immutable`).

Прежде всего, это:
- диапазоны – `range (immutable);`
- списки – `list (mutable);`
- кортежи – `tuple (immutable).`

Строки `str` в языке `Python` – это тоже последовательный тип данных. С ними можно работать также, как и с другими `immutable` конструкциями.

Чем `mutable` отличается от `immutable`? Объекты, относящиеся к `mutable` типам данных, позволяют изменять своё содержимое (поэлементно), а также изменять число элементов (увеличивать или уменьшать), не создавая нового объекта.

`Immutable` объекты данных действий не позволяют.

Цикл for

Цикл for используют, когда количество повторов известно заранее.

Итерация — это повтор какого-либо действия. То есть один шаг цикла. Например, цикл из пяти повторений — пять итераций.

Итератор — это интерфейс, который позволяет получить следующий объект последовательности. Цикл for – итератор.

Итерируемые объекты — это объекты, которые можно повторять (диапазоны, списки, кортежи, строки и т.д.).

```
for <переменная> in <последовательность>:
    <действие>
else:
    <действие>
```

Оператор прерывания в `python` — `break`
Если в программе цикл `for` должен быть прерван оператором `break`, цикл будет завершен, и поток программы будет продолжен без выполнения действий из `else`.

`Else` будет выполнен только в том случае, если цикл не был «остановлен» оператором `break`. Таким образом, он будет выполнен только после того, как все элементы последовательности будут пройдены.

Диапазоны – range

В Python функция range предоставляет генерацию последовательности чисел в определенном диапазоне. Это мощный инструмент для создания итерируемых объектов, часто используемых в циклах for или для генерации списков чисел.

```
range(stop)
range(start, stop, step)
```

`start` (опциональный): Начальное значение последовательности (по умолчанию 0).
`stop`: Конечное значение последовательности (не включается).
`step` (опциональный): Шаг изменения значения (по умолчанию 1).

```
for i in range(5):  # 0 1 2 3 4
    print(i, end=' ')
print()

for i in range(3, 6):  # 3 4 5
    print(i, end=' ')
else:
    print()

for i in range(10, 0, -1):  # 10 9 8 7 6 5 4 3 2 1
    print(i, end=' ')
else:
    print()

for i in range(3, 11, 2):  # 3 5 7 9
    print(i, end=' ')
    if i == 5:
        print("Find 5, exit for")
        break
else:
    print("all ok")
```

Списки – list

Списки в Python – упорядоченные изменяемые коллекции объектов произвольных типов. Каждый из элементов списка имеет свой номер, или индекс, позволяющий быстро получить к нему доступ. Нумерация элементов в списке начинается с 0. В одном списке одновременно могут лежать данные разных типов — например, и строки, и числа, и другие списки. Списки – динамические структуры данных, их можно менять на ходу: удалить один или несколько элементов, заменить или добавить новые.

```
l1 = []
l2 = [1,2,4,5,6]
l3 = [0, 100, "Список", [[1,2], ["a", "b"]]]
l4 = list(range(5,16))
print(l1, l2, l3, l4, sep="\n")

for x in l4:    # обход списка
    print(x)
```

Встроенные функции для работы с последовательностями

`list([object])` – создает список.

`all(последовательность)` – возвращает `True`, если все элементы истинные (или, если последовательность пуста).

`any(последовательность)` – возвращает `True`, если хотя бы один элемент – истина. Для пустой последовательности возвращает `False`.

`len(x)` – возвращает число элементов в указанном объекте.

`max(iter, [args ...] * [, key])` – максимальный элемент последовательности.

`min(iter, [args ...] * [, key])` – минимальный элемент последовательности.

Методы списков

`list.append(x)`	Добавляет элемент в конец списка
`list.extend(L)` `list+L`	Расширяет список `list`, добавляя в конец все элементы списка `L`
`list.insert(i, x)`	Вставляет на `i`-ый элемент значение x
`list.remove(x)`	Удаляет первый элемент в списке, имеющий значение x. ValueError, если такого элемента не существует
`list.pop([i])`	Удаляет i-ый элемент и возвращает его. Если индекс не указан, удаляется последний элемент
`list.index(x, [start [, end]])`	Возвращает положение первого элемента со значением `x` (при этом поиск ведется от `start` до `end`)
`list.count(x)`	Возвращает количество элементов со значением x
`list.sort([key=функция])`	Сортирует список на основе функции
`list.reverse()`	Разворачивает список
`list.copy()`	Поверхностная копия списка
`list.clear()`	Очищает список

Копирование и сравнение списков

```
a = [5,2,7,4,9]
b = a
b[0] = 999
print(a, b) # [999, 2, 7, 4, 9] [999, 2, 7, 4, 9]

a = [5,2,7,4,9]
b = a.copy()
b[0] = 999
print(a, b) # [5, 2, 7, 4, 9] [999, 2, 7, 4, 9]

c = [1,2,3]
d = [1,2,3]
print(c==d) # True

c = [1,2,3]
d = [1,2,3.0]
print(c==d) # True

c = [1,2,3]
d = [1,2,"3"]
print(c==d) # False

a = [1,2]
b = [1,2]
c = a
print(a is b, a is c) # False True
#оператор is проверяет, имеют ли две переменные один и тот же адрес в памяти
```

Обход списков

```
nums  = [3,5,2,6,9]
for n in nums:
  print(n)

for i in range(len(nums)):
  print(i, nums[i])

for i in range(-1, -len(nums)-1, -1): # обход с конца
    print(i, nums[i])

i = 0
while i<len(nums):
  print(nums[i])
  i+=1

sum = 0
for x in nums:
  sum+=x
print(sum)
```

Вырезки в Python

С изменяемыми последовательностями возможна операция `s[i:j]`. Данная конструкция называется вырезкой (`slicing`).
В целом для списков доступно несколько видов вырезок:

- `[start:end]` – элементы, начиная со `start` до (включительно) `end-1`
- `[start:]` – элементы, начиная со `start` и до конца списка
- `[:end]` – элементы, начиная с начала списка до (включительно) `end-1`
- [:] – копия всего списка

```
a = list("Это строка")
print(len(a))
print(a) #['Э', 'т', 'о', ' ', 'с', 'т', 'р', 'о', 'к', 'а']
print(a[2:5]) # ['о', ' ', 'с']
print(a[:7]) # ['Э', 'т', 'о', ' ', 'с', 'т', 'р']
print(a[3:]) # [' ', 'с', 'т', 'р', 'о', 'к', 'а']
print(a[:])  # ['Э', 'т', 'о', ' ', 'с', 'т', 'р', 'о', 'к', 'а']
```

Дополнительно: map, filter, reduce и lambda

```
from functools import reduce
a = [4,6,2,8,4]

print(list(map(lambda x: f"{x = }", a)))
#['x = 4', 'x = 6', 'x = 2', 'x = 8', 'x = 4’]

print(list(filter(lambda x: x>4, a))) # [6, 8]

print(reduce(lambda x,y: x+y,a, 0 )) # 24
```

Кортежы

Кортеж, по сути – неизменяемый список.

- Кортеж защищен от изменений, как намеренных, так и случайных.
- Также кортежи имеют меньший размер.
- Кортежи можно использовать как ключи словаря (списки – нельзя).

```
a = (2,3,1,4)
b = tuple(range(10))
print(a, b)  # (2, 3, 1, 4) (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
print(len(b), b[6])  # 10 6
sum = 0
for x in a:
    sum+=x
print(sum) # 10

b[7]=88 # TypeError: 'tuple' object does not support item assignment
```

Все операции над списками, не изменяющие список (сложение, умножение на число, методы index() и count() и некоторые другие операции) можно применять к кортежам.

Множества

Множество (set) — это изменяемый набор уникальных и неупорядоченных элементов.
Свойства множеств:

- Уникальность: каждый элемент множества неповторим. Если попробовать создать в наборе дубликат, Python не даст этого сделать.
- Неупорядоченность: у элементов множества нет порядкового номера.
- Изменяемость: можно добавлять во множество или удалять из него элементы.

Множества могут содержать только неизменяемые объекты:

- целые числа (integer);
- числа с плавающей точкой (float);
- строки (string);
- кортежи (tuple).

```
s1 = {1,2,3,3,4,4,5} # Создание множества, повторные элементы игнорируются
print(s1)

l1 = [1,2,3,3,4,5,5,5,5,8]
print(list(set(l1))) # Убираем повторы в списке

s2 = set() # Пустое множество
d  = {} # Пустой словарь!
```

Множества

К типу данных `set` можно применять встроенные функции len, max, min, sum, а также проверку на принадлежность `in и not in.`

Тип данных `set` в Python поддерживает операции, которые проводятся над множествами в математике:

- пересечение;
- объединение;
- разность.

```
s1 = {1, 2, 3, 4}
s2 = {3, 4, 5, 6}

print(s1|s2)   # Объединение
print(s1&s2)   # Пересечение
print(s1-s2)   # Разность
s1.add(99)     # Добавление элемента
print(s1)
s1.remove(99)  # Удаление элемента
print(s1)
#s1.remove(77) # Ошибка, если удаляемого элемента нет
s1.discard(77) # Удаление, если элемент есть, если нет, то ошибки нет.
```

Словари – dict

Словарь (dictionary) в языке Python хранит коллекцию элементов, где каждый элемент имеет уникальный ключ и ассоциированое с ним некоторое значение. Ключом может быть произвольный неизменяемый тип данных: различные числа, строки, кортежи. Значением элемента словаря может быть любой изменяемый или неизменяемый тип данных.

Пустой словарь в Python, как и в JavaScript, можно создать двумя способами: через функцию dict() или с помощью фигурных скобок.

Не пустой словарь – в фигурных скобках через запятую определяется последовательность элементов, где для каждого элемента сначала указывается ключ и через двоеточие его значение.

```
#Создание через функцию:
d1 = dict()
d2 = dict(car='машина', apple='яблоко')
#Создание через фигурные скобки:
d3 = {}
d4 = {
'car': 'машина',
'apple': 'яблоко'
}
d5 = {1: "123", 2: "456", 67: "90980", "657": [1,2,3,4]}
```

Обращение к элементам словаря

```
users = {
    "Tom": {
        "phone": "+971478745",
        "email": "tom12@gmail.com"
    },
    "Bob": {
        "phone": "+876390444",
        "email": "bob@gmail.com",
        "skype": "bob123"
    }
}

print(users) # {'Tom': {'phone': '+971478745', 'email': 'tom12@gmail.com'}, 'Bob': {'phone':
'+876390444', 'email': 'bob@gmail.com', 'skype': 'bob123'}}
print(users["Tom"]) #{'phone': '+971478745', 'email': 'tom12@gmail.com'}
print(users["Tom"]["phone"]) # +971478745
print(users["Tom"]["skype"]) # KeyError: 'skype'
tom_skype = users["Tom"].get("skype", "Undefined") # если ключа нет, то получим значение по
умолчанию
print(tom_skype) # Undefined

users["Jane"] = {"phone": "+87326873468", "email": "jane@gmail.com"}  # добавление пары ключ-
значение в словарь


new_users = users.copy() # создает копию словаря.

users.clear() # очищает словарь -> {}
```

Обход словаря.

```
print(users.keys()) # Набор ключей dict_keys(['Tom', 'Bob', 'Jane'])
print(users.values()) # Набор значений dict_values([{'phone': '+971478745',
'email': 'tom12@gmail.com'}, {'phone': '+876390444', 'email': 'bob@gmail.com',
'skype': 'bob123'}, {'phone': '+87326873468', 'email': 'jane@gmail.com'}])

for key in users.keys():
    print(key)
    if 'skype' in users[key].keys():
        print("\tskype: "+users[key]['skype’])
--------------------
Tom
Bob
        skype: bob123
Jane
--------------------

for key, value in users.items(): # набор пар ключ-значение
    print(key, value)
--------------------
Tom {'phone': '+971478745', 'email': 'tom12@gmail.com'}
Bob {'phone': '+876390444', 'email': 'bob@gmail.com', 'skype': 'bob123'}
Jane {'phone': '+87326873468', 'email': 'jane@gmail.com'}
```

Функции и их аргументы

Функция в `python` – объект, принимающий аргументы и возвращающий значение. Обычно функция определяется с помощью инструкции `def`.

```
def имя_функции (аргументы):
    тело_функции
    return результат
```

Инструкция `return` говорит, что нужно вернуть значение. Функция может и не заканчиваться инструкцией `return`, при этом функция вернет значение `None`.

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

def print_yes():
    print("yes")

print_yes()
print(add(5,6), add("asd", "ytr")) # 11 asdytr
```

Функции и их аргументы

Функция может принимать произвольное количество аргументов или не принимать их вовсе.

Необязательные аргументы задаются значениями по умолчанию.

```
def f1(x, y=6, z=10):
    print(f"{x = } {y = } {z = }")

f1(1,2,3)      # x = 1 y = 2 z = 3
f1(1,2)        # x = 1 y = 2 z = 10
f1(1)          # x = 1 y = 6 z = 10
f1(1, z=99)    # x = 1 y = 6 z = 99
f1(y=8, z=99) # TypeError: f1() missing 1 required positional argument: 'x’

a = [1,2,3]
f1(*a) # x = 1 y = 2 z = 3
f1(*[5,6]) # x = 5 y = 6 z = 10
c = [1,2,3,4]
f1(*c) # TypeError: f1() takes from 1 to 3 positional arguments but 4 were given
```

Функции и их аргументы

Функция может принимать переменное количество **позиционных** аргументов, тогда перед именем ставится *

```
def f2(*args):
    print(args)  # args это кортеж
    return True

f2(1,2,3,"kjh")  # (1, 2, 3, 'kjh')
f2("a", "b", 1,3,4,5,False, "hello") # ('a', 'b', 1, 3, 4, 5, False, 'hello')
f2() # ()
```

Функция может принимать произвольное число **именованных** аргументов, тогда перед именем ставится **:

```
def f3(**kwargs):
    print(kwargs)  # kwargs это словарь
    return kwargs

f3() # {}
f3(a=1, b=6, c = 9, value="some values") # {'a': 1, 'b': 6, 'c': 9, 'value': 'some
values'}
f3(id = "687268736", name="Tom Hanks", email="tom@tom.com") # {'id': '687268736',
'name': 'Tom Hanks', 'email': 'tom@tom.com'}
```