# Python

Условный оператор ветвления If

# Введение

- Оператор ветвления *if* позволяет выполнить определенный набор инструкций в зависимости от некоторого условия. Возможны следующие варианты использования:

1. **Конструкция *if***
2. **Конструкция *if – else***
3. **Конструкция *if – elif – else***

# 1. Конструкция *if*

- После оператора *if* записывается выражение. Если это выражение истинно, то выполняются инструкции, определяемые данным оператором. Выражение является истинным, если его результатом является число не равное нулю, непустой объект, либо логическое *True*. После выражения нужно поставить двоеточие “:”.

- **ВАЖНО**: блок кода, который необходимо выполнить, в случае истинности выражения, отделяется четырьмя пробелами слева!

```
if выражение:
    инструкция_1
    инструкция_2
    ...
    инструкция_n
```

# 1. Конструкция *if. Пример*

```
if 1: print("hello 1")
```

- Напечатает: *hello 1*

```
a = 3
if a == 3:
    print("hello 2")
```

- Напечатает: *hello 2*

```
a = 3
if a > 1:
    print("hello 3")
```

- Напечатает: *hello 3*

```
lst = [1, 2, 3]
if lst :
    print("hello 4")
```

- Напечатает: *hello 4*

# 2. Конструкция *if – else*

- Бывают случаи, когда необходимо предусмотреть альтернативный вариант выполнения программы. Т.е. при истинном условии нужно выполнить один набор инструкций, при ложном – другой. Для этого используется конструкция *if – else*.

```
if выражение:
    инструкция_1
    инструкция_2
    ...
    инструкция_n
else:
    инструкция_a
    инструкция_b
    ...
    инструкция_x
```

# 2. Конструкция *if – else. Пример*

```
a = 3
if a > 2:
    print("H")
else:
    print("L")
```

- Напечатает: *H*

```
a = 1
if a > 2:
    print("H")
else:
    print("L")
```

- Напечатает: *L*

# 3. Конструкция *if – elif – else*

- Для реализации выбора из нескольких альтернатив можно использовать конструкцию*if – elif – else*.

```
if выражение_1:
    инструкции_(блок_1)
elif выражение_2:
    инструкции_(блок_2)
elif выражение_3:
    инструкции_(блок_3)
else:
    инструкции_(блок_4)
```

# 3. Конструкция *if – elif – else. Пример*

```
a = int(input("введите число:"))
if a < 0:
    print("Neg")
elif a == 0:
   print("Zero")
else:
   print("Pos")
```

- Если пользователь введет число меньше нуля, то будет напечатано *“Neg“*, равное нулю – *“Zero“*, большее нуля – *“Pos“*.

# Цикл while

# Введение

Цикл while также используется для повторения частей кода, но вместо зацикливания на n количество раз, он выполняет работу до тех пор, пока не достигнет определенного условия. Давайте взглянем на простой пример:

```
i = 0
while i < 10:
    print(i)
    i = i + 1
```

Цикл while является своего рода условным оператором. Вот что значит этот код: пока переменная i меньше единицы, её нужно выводить на экран. Далее, в конце, мы увеличиваем её значение на единицу. Если вы запустите этот код, он выдаст от 0 до 9, каждая цифра будет в отдельной строке, после чего задача будет выполнена. Если вы удалите ту часть, в которой мы увеличиваем значение i, то мы получим бесконечный цикл. Как правило – это плохо. Бесконечные циклы известны как логические ошибки, и их нужно избегать. Существует другой способ вырваться из цикла, для этого нужно использовать встроенную функцию break. Давайте посмотрим, как это работает:

```
i = 0
while i < 10:
    print(i)

    if i == 5:
     break

    i += 1
```

В этой части кода мы добавили условное выражение для проверки того, равняется ли когда-либо переменная i цифре 5. Если нет, тогда мы разрываем цикл. Как вы видите в выдаче кода, как только значение достигает пяти, код останавливается, даже если мы ранее указали while продолжать цикл, пока переменная не достигнет значения 10. Обратите внимание на то, что мы изменили то, как мы увеличиваем значение при помощи +=. Это удобный ярлык, который вы можете также использовать в других операциях, таких как вычитание -= и умножение *=. Встроенный break также известен как инструмент управления потока. Существует еще один, под названием continue, который в основном используется для пропуска итерации, или перейти к следующей итерации. Вот один из способов его применения:

```
i = 0

while i < 10:
    if i == 3:
        i += 1
        continue

    print(i)
    if i == 5:
        break

    i += 1
```

# Заключение

Слегка запутанно, не так ли? Мы добавили второе условное выражение, которое проверяет, не равняется ли i трем. Если да, мы увеличиваем переменную и переходим к следующему циклу, который удачно пропускает вывод значения 3 на экран. Как и ранее, когда мы достигаем значения 5, мы разрываем цикл. Существует еще одна тема, касающаяся циклов, которую нам нужно затронуть – это оператор else.

# Оператор else

Оператор else в циклах выполняется только в том случае, если цикл выполнен успешно. Главная задача оператора else, это поиск объектов:

```
my_list = [1, 2, 3, 4, 5]

for i in my_list:
    if i == 3:
        print("Item found!")
        break
    print(i)
else:
    print("Item not found!")
```

# Заключение

- В этом коде мы разорвали цикл, когда i равно 3. Это приводит к пропуску оператора else. Если вы хотите провести эксперимент, вы можете изменить условное выражение, чтобы посмотреть на значение, которое находится вне списка, и которое приведет оператор else к выполнению. Честно, ни разу не видел, чтобы кто-либо использовал данную структуру за все годы работы. Большая часть примеров, которые я видел, приведена блогерами, которые пытаются объяснить, как это работает. Я видел несколько людей, которые использовали эту структуру для провоцирования ошибки, когда объект не удается найти в искомом цикле.

# Python

Условный оператор цикла for

Оператор *for* выполняет указанный набор инструкций заданное количество раз, которое определяется количеством элементов в наборе.

Пример.

```
for i in range(5):
print("Hello")
```

В результате *“Hello”* будет выведено пять раз.

# Внутри тела цикла можно использовать операторы ***break*** и ***continue***, принцип работы их точно такой же как и в операторе *while*

Если у вас есть заданный список, и вы хотите выполнить над каждым элементом определенную операцию (возвести в квадрат и напечатать получившееся число), то с помощью *for* такая задача решается так.

```
lst = [1, 3, 5, 7, 9]
for i in lst:
   print(i ** 2)
```

Также можно пройти по всем буквам в строке.

```
word_str = "Hello, world!"
for l in word_str:
   print(l)
```

Строка “Hello, world!” будет напечатана в столбик.

# Пример цикла for с диапазоном чисел

```
1  s=0
2
3  for i in range(1,n):
4
5      s=3+s
6
7
```

то есть пока i идёт от 1 до n-значения переменная s будет накапливать сумму чисел, делящихся на 3.

# Пример цикла for со строкой

```
1  a=Эта строка задаёт количество проходов цикла
2
3  s=0
4
5  for i in a:
6
7      s=1+s
8
9  print(Длина строки ",a,"  =,s)
10
```

В этом цикле в переменной s вычисляется длина строки, которая была передана в строковую переменную a.

# Массивы (списки)

# Что такое массив?

Как ввести 10000 переменных?

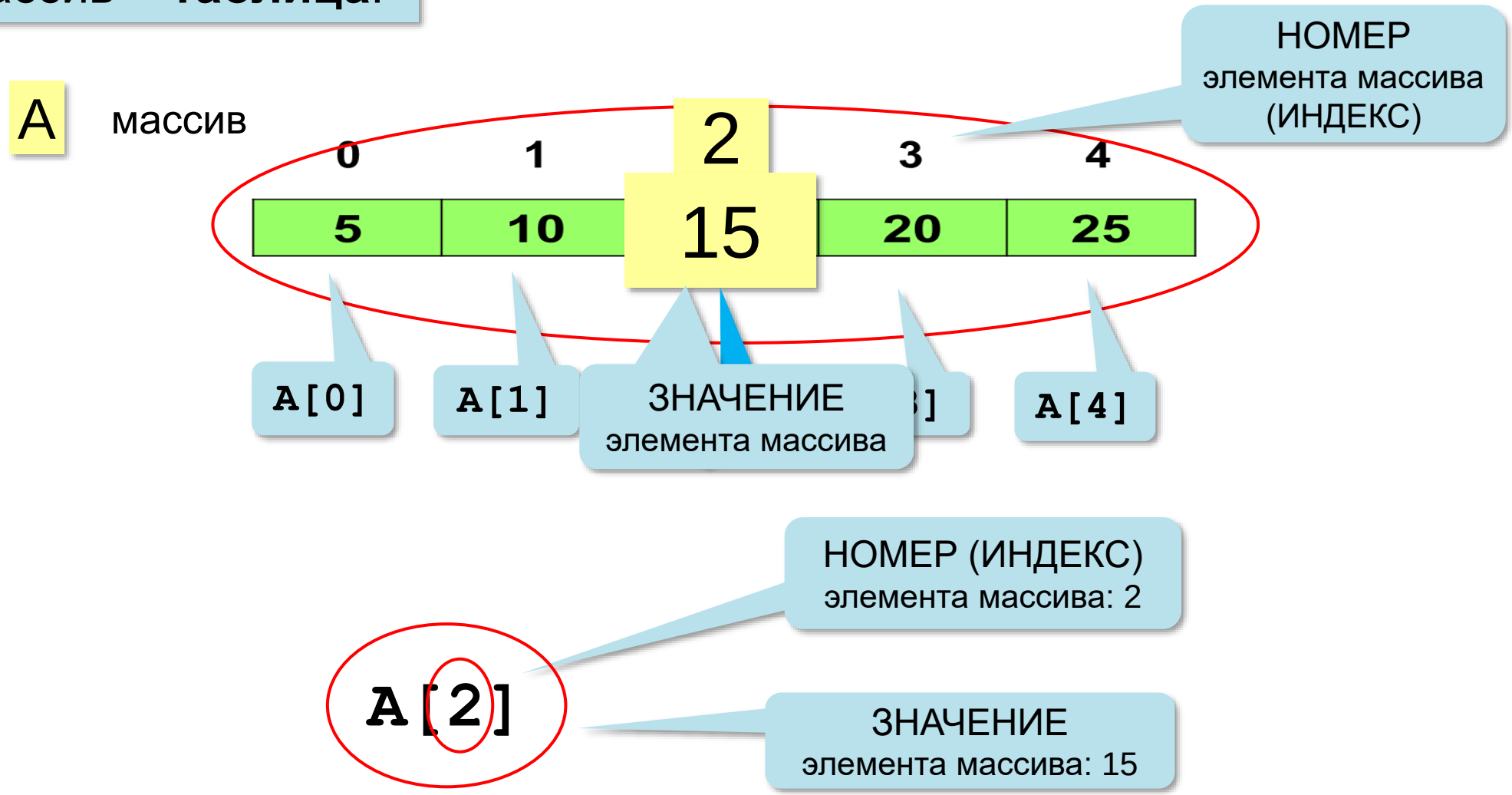
**Массив** – это группа переменных одного типа, расположенных в памяти рядом (в соседних ячейках) и имеющих общее имя. Каждая ячейка в массиве имеет уникальный номер (индекс).

**Надо**:

выделять память
записывать данные в нужную ячейку
читать данные из ячейки

# Что такое массив?

Массив = **таблица**!



# Массивы в Python



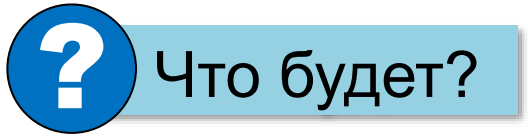
```
A = [1, 3, 4, 23, 5]
```

```
A = [1, 3] + [4, 23] + [5]
```

```
[1, 3, 4, 23, 5]
```

```
A = [0]*10
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```




## Создание массива из N элементов:

```
N = 10
A = [0]*N
```

# Заполнение массива

**Целыми числами (начиная с 0!):**

```
N = 10        # размер массива
A = [0]*N     # выделить память
for i in range(N):
   A[i] = i i
```

**В краткой форме:**

```
N = 10        # размер массива
A = [ i for i in range(N) ]
```

? Как заполнить, начиная с 1?

? Как заполнить квадратами чисел?

# Заполнение массива

из библиотеки (модуля) random

взять функцию **randint**

```
from random import randint
N = 10       # размер массива
A = [0]*N    # выделить память
for i in range(N):
  A[i] = randint(20,100)
```

**В краткой форме:**

```
from random import randint
N = 10
A = [ randint(20,100)
      for i in range(N) ]
```

# Вывод массива на экран

**Как список:**

```
print ( A )
```

```
[1, 2, 3, 4, 5]
```

**В строчку через пробел:**

```
for i in range(N):
  print ( A[i], end=" " )
```

```
1 2 3 4 5
```

пробел после вывода очередного числа

**или так:**

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

```
1 2 3 4 5
```

**или так:**

```
print ( *A )
```

⟺

```
print (1, 2, 3, 4, 5)
```

разбить список на элементы

# Ввод массива с клавиатуры

## Создание массива:

```
N = 10
A = [0]*N
```

## Ввод по одному элементу в строке:

```
for i in range(N):
  A[i] = int( input() )
```

## или кратко:

```
A = [int(input())
        for i in range(N)]
```

# Ввод массива с клавиатуры

**Ввод всех чисел в одной строке:**

```
data = input()      # "1 2 3 4 5"
s = data.split()    # ["1","2","3","4","5"]
A = [ int(x) for x in s ]
                    # [1,2,3,4,5]
```

**или так:**

```
A = [int(x) for x in input().split()]
```

# Как обработать все элементы массива?

**Создание массива**:

```
N = 5
A = [0]*N
```

**Обработка**:

```
# обработать A[0]
# обработать A[1]
# обработать A[2]
# обработать A[3]
# обработать A[4]
```

1) если N велико (1000, 1000000)?

2) при изменении N программа не должна меняться!

# Как обработать все элементы массива?

## Обработка с переменной:

```
i = 0
# обработать A[i]
i += 1
# обработать A[i]
i += 1
# обработать A[i]
i += 1
# обработать A[i]
i += 1
# обработать A[i]
```

```
i += 1
```

## Обработка в цикле:

```
i = 0
while  i < N:
   # обработать A[i]
   i += 1
```

## Цикл с переменной:

```
for i in range(N):
   # обработать A[i]
```

# Перебор элементов

## Общая схема (можно изменять A[i]):

```
for i in range(N):
    ... # сделать что-то с A[i]
```

```
for i in range(N):
    A[i] += 1
```

## Если не нужно изменять A[i]:

```
for x in A:
    ... # сделать что-то с x
```

x = A[0], A[1], ..., A[N-1]

```
for x in A:
    print( x )
```

# Что выведет программа?

```
A = [2, 3, 1, 4, 6, 5]
```

```
print( A[3] )                # 4
```

```
print( A[0]+2*A[5] )         # 12
```

```
A[1] = A[0] + A[5]           # 7
print( 3*A[1]+A[4] )         # 27
```

```
A[2] = A[1]*A[4]             # 18
print( 2*A[1]+A[2] )         # 22
```

```
for k in range(6):
  A[k] += 2          # [4,5,3,6,8,7]
print( 2*A[3]+3*A[4] )  # 36
```

# Python

## Индексы и срезы

# Индексы

- Нумерация в списках начинается с нуля, так как список по большей части своей это просто массив, то как в обычном массиве отсчет ведется от 0. Поэтому первый элемент по индексу будет 0, второй - 1, третий - 2 и так далее. Если мы попытаемся взять несуществующий элемент, то это приведет к ошибке.

```
a = [0, 23, "Hi"] # Список
print (a[4]) # Выдаст ошибку, так как элемента не существует
```

# Индексы

- Очень удобной функцией языка Python является возможность брать элементы с конца при помощи отрицательных индексов. К примеру, если нам нужен второй элемент с конца, то мы можем записать это так:

```
a = [0, 23, "Hi", 1.56, 9] # Список
print (a[-2]) # Будет выведено 1.56
```

# Срезы

Срезы позволяют обрезать список, взяв лишь те элементы, которые нам будут нужны. Они работают по следующей схеме: list[НАЧАЛО:КОНЕЦ:ШАГ].

- Начало - с какого элемента стоит начать (по умолчанию равна 0);
- Конец - по какой элемент мы берем элементы (по умолчанию равно длине списка);
- Шаг - с каким шагом берем элементы, к примеру каждый 2 или 3 (по умолчанию каждый 1).
- Один, несколько или даже все параметры могут быть пропущены.

# Срезы

```
list[::3] # Берем каждый третий элемент
list[2::2] # Начиная со второго элемента берем каждый второй элемент
list[4:6:] # Начиная с 4 элемента берем все элементы по 6 элемент
list[::] # Берем все элементы
```

- Также могут быть использованы отрицательные числа для срезов.

# Вывод списка и срез

## Исходный код:

```
l = [34, 'sd', 56, 34.34]

i = 0
while i < 4:
      print (l[i])
      i += 1

print (l[-2::-2])
```

## Вывод:

stdout

```
34
sd
56
34.34
[56, 34]
```

# Форматирование строк. Метод format

- Иногда (а точнее, довольно часто) возникают ситуации, когда нужно сделать строку, подставив в неё некоторые данные, полученные в процессе выполнения программы (пользовательский ввод, данные из файлов и т. д.). Подстановку данных можно сделать с помощью форматирования строк. Форматирование можно сделать с помощью оператора %, либо с помощью метода format.

# Форматирование строк с помощью метода format

- Если для подстановки требуется только один аргумент, то значение - сам аргумент:

```
>>> 'Hello, {}!'.format('Vasya')
'Hello, Vasya!'
```

- А если несколько, то значениями будут являться все аргументы со строками подстановки (обычных или именованных):

(далее пример)

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{}, {}, {}'.format('a', 'b', 'c')
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')
'abracadabra'
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-
↪115.81W')
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

- Однако метод **format** умеет большее. Вот его синтаксис:

```
поле замены      ::=  "{" [имя поля] ["!" преобразование] [":" спецификация] "}"
имя поля         ::=  arg_name ("." имя атрибута | "[" индекс "]")*
преобразование   ::=  "r" (внутреннее представление) | "s" (человеческое
↪представление)
спецификация     ::=  см. ниже
```

- Например:

```
>>> "Units destroyed: {players[0]}".format(players = [1, 2, 3])
'Units destroyed: 1'
>>> "Units destroyed: {players[0]!r}".format(players = ['1', '2', '3'])
"Units destroyed: '1'"
```

- Теперь спецификация формата:

```
спецификация ::=  [[fill]align][sign][#][0][width][,][.precision][type]
заполнитель  ::=  символ кроме '{' или '}'
выравнивание ::=  "<" | ">" | "=" | "^"
знак         ::=  "+" | "-" | " "
ширина       ::=  integer
точность     ::=  integer
тип          ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" |
                  "n" | "o" | "s" | "x" | "X" | "%"
```

- Выравнивание производится при помощи символа-заполнителя. Доступны следующие варианты выравнивания:

| Флаг | Значение |
|---|---|
| ‘<’ | Символы-заполнители будут справа (выравнивание объекта по левому краю) (по умолчанию). |
| ‘>’ | Выравнивание объекта по правому краю. |
| ‘=’ | Заполнитель будет после знака, но перед цифрами. Работает только с числовыми типами. |
| ‘^’ | Выравнивание по центру. |

- Опция “знак” используется только для чисел и может принимать следующие значения:

| **Флаг** | **Значение** |
|---|---|
| ‘+’ | Знак должен быть использован для всех чисел. |
| ‘-‘ | ‘-‘ для отрицательных, ничего для положительных. |
| ‘Пробел’ | ‘-‘ для отрицательных, пробел для положительных. |

# Форматирование строк с помощью метода format

| Тип | Значение |
|---|---|
| ‘d’, ‘i’, ‘u’ | Десятичное число. |
| ‘o’ | Число в восьмеричной системе счисления. |
| ‘x’ | Число в шестнадцатеричной системе счисления (буквы в нижнем регистре). |
| ‘X’ | Число в шестнадцатеричной системе счисления (буквы в верхнем регистре). |
| ‘e’ | Число с плавающей точкой с экспонентой (экспонента в нижнем регистре). |
| ‘E’ | Число с плавающей точкой с экспонентой (экспонента в верхнем регистре |
| ‘f’, ‘F’ | Число с плавающей точкой (обычный формат). |
| ‘g’ | Число с плавающей точкой. с экспонентой (экспонента в нижнем регистре), если она меньше, чем -4 или точности, иначе обычный формат |
| ‘G’ | Число с плавающей точкой. с экспонентой (экспонента в верхнем регистре), если она меньше, чем -4 или точности, иначе обычный формат. |
| ‘c’ | Символ (строка из одного символа или число - код символа). |
| ‘s’ | Строка |
| ‘%’ | Число умножается на 100, отображается число с плавающей точкой, а за ним знак %. |

- Пример форматирования:

```
>>> coord = (3, 5)
>>> 'X: {0[0]};  Y: {0[1]}'.format(coord)
'X: 3;  Y: 5'
>>> "repr() shows quotes: {!r}; str() doesn't: {!s}".format('test1', 'test2')
"repr() shows quotes: 'test1'; str() doesn't: test2"
>>> '{:<30}'.format('left aligned')
'left aligned                  '
>>> '{:>30}'.format('right aligned')
'                 right aligned'
>>> '{:^30}'.format('centered')
'           centered           '
>>> '{:*^30}'.format('centered')  # use '*' as a fill char
'**********centered**********'
>>> '{:+f}; {:+f}'.format(3.14, -3.14)  # show it always
'+3.140000; -3.140000'
>>> '{: f}; {: f}'.format(3.14, -3.14)  # show a space for positive numbers
' 3.140000; -3.140000'
>>> '{:-f}; {:-f}'.format(3.14, -3.14)  # show only the minus -- same as '{:f};
↪{:f}'
'3.140000; -3.140000'
>>> # format also supports binary numbers
>>> "int: {0:d};  hex: {0:x};  oct: {0:o};  bin: {0:b}".format(42)
'int: 42;  hex: 2a;  oct: 52;  bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
>>> "int: {0:d};  hex: {0:#x};  oct: {0:#o};  bin: {0:#b}".format(42)
'int: 42;  hex: 0x2a;  oct: 0o52;  bin: 0b101010'
>>> points = 19.5
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 88.64%'
```

# Именные функции, инструкция def

Функция – это блок кода, который начинается с ключевого слова def, названия функции и двоеточия

```python
def a_function():
    print("You just created a function!")
```

Такая функция выведет на экран строку «You just create a function!»

Чтобы вызвать функцию, вам нужно ввести название функции, за которой следует открывающаяся и закрывающаяся скобки

Python

```
1  a_function() # You just created a function!
```

# Передача аргументов функции

```python
def add(a, b):
    return a + b

print( add(1, 2) ) # 3
```

Количество передаваемых аргументов должно быть равно количеству объявленных в функции аргументов.

# Ключевые аргументы

```python
def keyword_function(a=1, b=2):
    return a+b

print( keyword_function(b=4, a=5) ) # 9
```

```python
keyword_function() # 3
```

# *args и **kwargs

```python
def many(*args, **kwargs):
    print( args )
    print( kwargs )

many(1, 2, 3, name="Mike", job="programmer")

# Результат:
# (1, 2, 3)
# {'job': 'programmer', 'name': 'Mike'}
```

Функция может принимать произвольное число аргументов. Для позиционных аргументов пишется *, для именованных - **. Слова args и kwargs приняты соглашением, но не обязательны, вместо них можно использовать любые другие.