
Яндекс

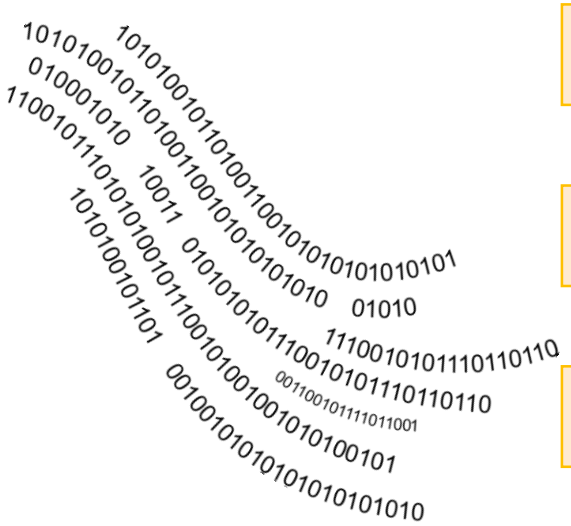
Академия Яндекса

Функции

Введение

Функция как способ группировать команды и именовать участки кода

Работа любой компьютерной программы — это выполнение процессором большого набора элементарных инструкций. В машинном коде, с которым работает процессор, все команды очень простые:



- считать из оперативной памяти одно целое число в специальную ячейку,
- прибавить к одному числу значение другой ячейки,
- сравнить ячейку с нулём,
- вернуться на пару команд назад и пр.

Функция как способ группировать команды и именовать участки кода

Многие из команд **Python**, которые вы уже знаете, требуют от процессора выполнения десятков команд. Если бы программист писал их вручную, то даже простейшие программы — вроде наших учебных заданий — создавались бы несколько дней. При этом даже опытному программисту было бы очень легко допустить ошибку.

Пример

```
print('Как тебя зовут?')
name_1 = input()
print('Привет', name_1)
print('А тебя?')
name_2 = input()
print('Привет', name_2)
print('А твоего пса?')
name_3 = input()
print('Привет', name_3)
```

```
Как тебя зовут?
Вася
Привет Вася
А тебя?
Коля
Привет Коля
А твоего пса?
Шарик
Привет Шарик
```

Какие проблемы?

Функция как способ группировать команды и именовать участки кода

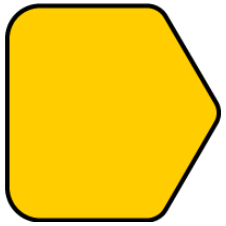
Итак, сформулируем, чего мы хотим добиться:

один раз определить, что значит "поприветствовать", т. е. сгруппировать и поименовать повторяющийся кусок кода;

многократно в дальнейшем "ссылаться" на это определение везде, где нам только потребуется.

Замечательно, что язык `Python` действительно обладает такими выразительными возможностями — **функциями**.

Функция как способ группировать команды и именовать участки кода



Функция — это особым образом сгруппированный набор команд, которые выполняются последовательно, но воспринимаются как единое целое. При этом функция может возвращать (или не возвращать) свой результат.

Для того, чтобы использовать какую-нибудь собственную функцию, вначале необходимо её **объявить**, т. е. рассказать, что именно она будет делать. В нашем примере мы объявим функцию **greet** с помощью ключевого слова **def.**

```
def greet():
    name = input()
    print('Привет', name)
```

Объявление функции

```
print('Как тебя зовут?')
greet()

print('А тебя?')
greet()

print('А твоего пса?')
greet()
```

Вызов функции

Функция как способ группировать команды и именовать участки кода

Что мы получили в результате:

Код сократился и стал понятнее. Теперь нам не нужно выискивать, где какая переменная заводится, где и для чего она используется. Функция сама говорит, что она делает: **greet** — «поприветствовать».

Нам не приходится заводить несколько разных переменных.

Чтобы поменять приветствие во всей программе, достаточно изменить одну строчку.

Итак, **функции** нужны, чтобы группировать команды, а заодно — чтобы не писать один и тот же код несколько раз.

Функция как способ группировать команды и именовать участки кода

Ещё одна важная вещь состоит в том, что функции имеют **имена**.

Имена функций должны состоять из маленьких букв, а слова разделяться символами подчеркивания — это необходимо, чтобы увеличить читабельность.

Попробуйте угадать, что делает такой код:

```
t = [-5, -10, 1, 11, 20, 25, 27, 23, 18, 8, 2, -3]
s = 0
mm = 1000
mx = -1000
for e in t:
    s += e
    if e < mm:
        mm = e
    if e > mx:
        mx = e
print(s / len(t))
print(mm)
print(mx)
```

Функция как способ группировать команды и именовать участки кода

А вот тот же самый код, но только переработанный с использованием встроенных функций и хороших названий переменных:

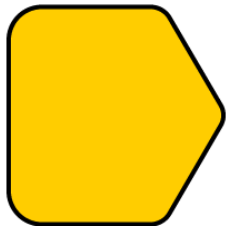
```
temperatures = [-5, -10, 1, 11, 20, 25, 27, 23, 18, 8, 2, -3]
average_temperature = sum(temperatures) / len(temperatures)

print(average_temperature)
print(min(temperatures))
print(max(temperatures))
```

Какой вариант вам нравится больше?

Когда этот код «сломается»? Как это исправить?

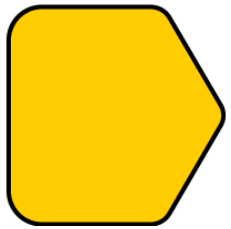
Определение простейших функций

Определение простейших функций



У каждой функции есть **заголовок** (его обычно называют сигнатурой) и **тело функции**. Сигнатура описывает, как функцию вызывать, а тело описывает, что эта функция делает. Сигнатура содержит имя функции, а также аргументы (то есть параметры), которые передаются в функцию.

```
def <имя функции>([аргументы]):
    <тело функции>
```

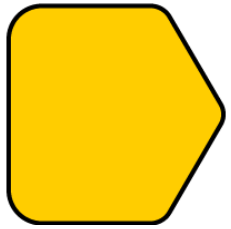


Тело функции, как и в операторе **`if`** или в операторе цикла, обязательно идёт с **отступом**. Это нужно, чтобы интерпретатор **`Python`** знал, где заканчивается код функции. Заодно это здорово помогает структурировать программу. Даже в языках, где отступы не требуются, их всё равно принято писать, чтобы упростить чтение программы.

Определение простейших функций

Давайте напишем еще одну совсем простую функцию из одной единственной команды, которая просто выводит на экран приветствие.

```
def simple_greetings():
    print('Привет!')
```



Теперь, чтобы поприветствовать пользователя, вам достаточно в основной программе написать: `simple_greetings()`. **Это называется вызвать функцию.**
Обратите внимание, что у этой функции нет аргументов ни в определении, ни при вызове. Однако пустые скобочки после названия функции писать всё равно нужно.

Определение простейших функций

Функцию, как и переменную, необходимо сначала объявить, а только потом использовать.
Поэтому следующая программа выдаст вам ошибку
`NameError: name 'simple_greetings2' is not defined`.

```
simple_greetings2()


def simple_greetings2():
    print('Привет, username!')
```

В виде функции стоит оформлять только логически законченный блок кода

После функции до кода, который находится вне функции необходимо делать отступ в 2 пустые строки для повышения читаемости кода. Если у вас есть несколько функций в одном файле, между кодом одной и сигнатурой другой функции также надо оставлять 2 пустые строки.

Отправка задач в LMS

Обратите внимание, что при сдаче в LMS задач на функции, есть некоторые особенности:

В LMS необходимо отправить файл с **только необходимой функцией**. Если в файле есть код вызова данной функции, то закомментируйте его. В **Pycharm** можно выделить код, который необходимо закомментировать и нажать `ctrl+/`. Для раскомментирования надо повторить тоже самое.

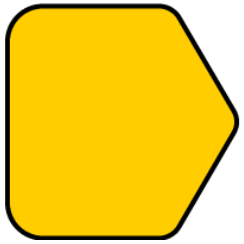
При проверке функции тестирующая система запускает программу на **Python**, которая осуществляет вызов вашей функции (с передачей в нее параметров), поэтому необходимо точно соблюсти формат сигнатуры функции, который дается в условии.

Начальные знания о локальных переменных

Начальные знания о локальных переменных

В тот момент, когда вы вызываете функцию **greet**, начинают выполняться команды, написанные в теле функции. Когда работа функции доходит до конца, исполнение программы продолжается со строки, которая вызывала функцию.

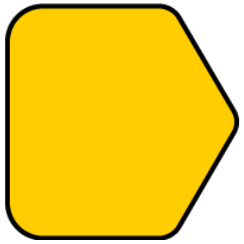
Обратите внимание, теперь в программе используется только одна переменная: **name**. Как же так? Ведь мы договорились, что не будем использовать одну и ту же переменную для разных имен? На самом деле мы не используем одну и ту же переменную. При каждом вызове функции эта переменная создается заново, а в конце работы функции — прекращает своё существование. Это очень важный момент:



Снаружи функции **greet** переменная **name** вообще не существует. Таким образом, функция очерчивает тот участок программы, где переменная нужна и используется. Этот участок, в котором переменная **живёт**, называется **областью видимости переменной** (по-английски — scope).

Начальные знания о локальных переменных

Благодаря ограничению области видимости переменной, программисту не нужно беспокоиться, не «всплывёт» ли эта переменная в другом месте программы. Изменяя переменную внутри функции, программист понимает, что он может что-то испортить **только внутри** функции, но не поломает работу остальной программы. Можно сказать, что вся работа с переменной локализована, т. е. сосредоточена внутри функции.



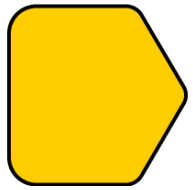
Переменные, создаваемые внутри функций, недоступны извне и существуют только внутри функции, они называются **локальными**. Создаваемые вне функции переменные могут быть доступны из функций — они являются **глобальными**.

По возможности избегайте использования глобальных переменных для предотвращения конфликтов.

Аргументы функций

Аргументы функций

Мы рассмотрели функции, которые выполняют всякий раз одни и те же действия. Это бывает полезно, но всё же большая часть программ требует выполнения немного разных действий. Например, функция **`print`** (а это именно функция) должна каждый раз выводить на экран разные сообщения — в зависимости от переданных аргументов.



Аргументы (параметры) могут изменять поведение функции. Например, функция **`len`** принимает строки или списки (и другие коллекции). В зависимости от конкретного аргумента она возвращает разный результат, а значит, выполняет внутри немного различные действия.

```
def print_array(array):
    for element in array:
        print(element)


print_array(['Hello', 'world'])
print()
print_array([123, 456, 789])
```

```
Hello
world

123
456
789
```

Аргументы функций

Разберемся, в каком порядке выполняется код при вызове функций. В примере:

```
print_array(['Hello'] + ['world'])
```

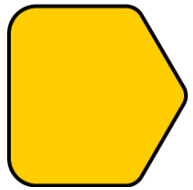
```
Hello
world
```

Ничего удивительного не происходит: списки складываются, а затем передаются в функцию.

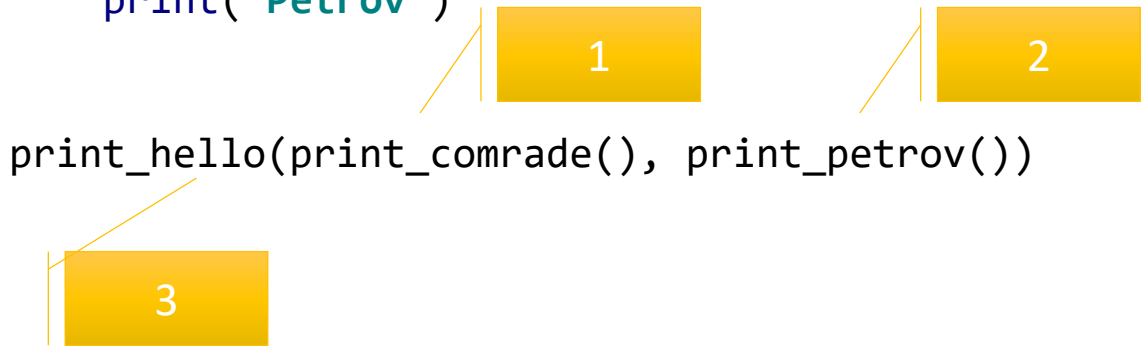
Аргументы функций

Давайте рассмотрим более сложный пример:

```
def print_hello(arg_1, arg_2):
    print('hello')


def print_comrade():
    print('comrade')


def print_petrov():
    print('Petrov')
```



В момент вызова функции ей необходимо передать вычисленные аргументы. Если аргументы не вычислены, то они вычисляются слева направо.




```
comrade
Petrov
hello
```


Яндекс