**Функции**

Функция — это блок кода, который можно многократно вызывать на выполнение. Она является фундаментальной частью любого языка программирования.

Функция позволяет разделять программу на самостоятельные, но связанные части. Программисты используют функции, чтобы сделать программу модульной и избежать повторения кода.

Функция может использоваться для обработки данных, она получает на вход значения, обрабатывает его и возвращает результат в программу. Также она может не возвращать значение, а выводить его на экран или записывать в файл.

Программист может написать собственную функцию или использовать готовые решения языка, если они есть, конечно. Например, лучше самому не написать функцию для определения максимального числа, а воспользоваться стандартной max().

**Объявление**

**Объявляя функцию, нужно следовать определенным правилам**:

* Объявление происходит с помощью ключевого слова def, за ним идёт имя функции и круглые скобки ().
* Аргументы, передаваемые в функцию, должны находится в круглых скобках. Там же можно определить их значения по умолчанию, указав их после знака равно.
* Перед основным содержимым желательно включить строку документации ([docstring](https://all-python.ru/osnovy/kommentarii.html)), которая обычно описывает назначение и основные принципы работы функции.
* Тело функции начинается после знака двоеточия. Важно не забыть об отступах.
* Чтобы выйти из функции в Python, используют оператор return [значение]. Если оператор опущен, будет возвращено значение None.

Функцию можно объявить где угодно: внутри модуля, класса или другой функции. Если она объявляет внутри класса, то называется методом класса и вызывается так: class\_name.function().

**Синтаксис объявления**

Параметры (аргументы) нужно передавать в том порядке, в котором они определены в круглых скобках.

def Имя(аргументы):

"Документация"

Тело (инструкции)

return [значение]

**Пример**

Функция складывает два числа, переданные в качестве аргументов. Если один или оба аргумента не были переданы, используются значения по умолчанию.

def print\_sum(a = 2, b = 2):

sum = a + b

print(sum)

return # вернёт None

**Вызов**

После определения функции её можно вызвать в любой точке скрипта, как в теле самого скрипта, так и в теле другой функции:

# определяем функцию

def print\_sum(a = 2, b = 2):

sum = a + b

print(sum)

#вызываем её

print\_sum(5, 1)

Значение функции можно сразу передать в переменную или в другую функцию:

def sum(a = 2, b = 2):

sum = a + b

return sum # вернёт сумму

c = sum(4, 3) # переменная c будет равна возвращаемому значению

print(sum(5, 5)) # можно передать значения в аргументы другой функции

Вызов функции работает медленнее, чем обычный код. Это незаметно, пока её не вызывают в цикле. Если от цикла необходимо добиться максимального быстродействия, нужно отказаться от вызова функции и просто вставить внуть тела цикла её код.

**Необязательные параметры**

При описании функции в Python 3 аргументы, которым задаются начальные значения, являются необязательными. Вначале должны описываться обязательные параметры и только после них необязательные.

При вызове функции нам не обязательно указывать значения необязательных параметров. Если мы хотим изменить значение аргумента, не меняя начальные значения других аргументов, можно обращаться к нему по ключу.

Вот пример:

def example(first, second=3, third=5):

print(first)

print(second)

print(third)

example('my string', third=4)

Вывод будет следующим:

my string

3

4

**Функция с переменным числом аргументов**

Часто возникает необходимость создать такую функцию, которая может принимать разное количество аргументов. Это можно реализовать с помощью передачи списка или массива, однако Python позволяет использоваться более удобный подход (также с использованием списка).

Для того чтобы функция могла принять переменное количество аргументов, перед именем аргумента ставится символ ” \* “. Когда программист передаёт аргументы, они записываются в кортеж, имя которого соответствует имени аргумента:

def variable\_len(\*args):

for x in args:

print(x)

variable\_len(1,2,3) # Выведет 1,2,3

Вместо одного символа звёздочки можно использовать два, тогда аргументы будут помещаться не в список, а в словарь:

def variable\_len(\*\*args):

print(type(args))

for x, value in args.items():

print(x, value)

variable\_len(apple = "яблоко", bread = "хлеб")

# Выведет apple яблоко bread хлеб

**Анонимные функции**

Это особый вид функций, которые объявляются с помощью ключевого слова lambda вместо def:

Лямбда функции принимают любое количество аргументов, но не могут содержать несколько выражений и всегда возвращают только одно значение.

В программировании на Python можно обойтись без анонимных функций, которые по сути являются обычными, но без имени и с ограничением в одно выражение. Однако их использование в нужных местах упрощает написание и восприятие кода. Например, в программе калькулятор мы с её помощью сделали обработчики нажатия кнопок.

**Синтаксис**

Синтаксис лямбда функции в Python 3 предусматривает использование только одного выражения: lambda arg1, arg2, … argn: выражение.

На практике они работают так:

x = lambda arg1, arg2: arg1 \* arg2

print(x(5,5)) # Выведет 25

print(x(3,5)) # Выведет 15

**Возврат значений**

С помощью оператора return из функции можно вернуть одно или несколько значений. Возвращаемым объектом может быть: число, строка, None.

Чтобы вернуть несколько значений, нужно написать их через запятую. Python позволяет вернуть из функции список или другой контейнер: достаточно указать после ключевого слова return имя контейнера.

Вот пример, когда возвращается список:

def x(n):

a = [1,3]

a = a \* n

return a

print(x(2)) # выведет [1,3,1,3]

А это пример того, как функция в Python 3 возвращает несколько значений. Так как переменные перечислены через запятую, то они образуют список. Эти значения можно присвоить сразу нескольким переменным, как это показано в следующем примере:

def example():

language = "python"

version = 3

flag = True

return language, version, flag

language, version, flag = example()

print(language, version, flag) # выведено будет python 3 True

**Рекурсия**

Рекурсией называется процесс, когда функция вызывает саму себя. Её можно использовать вместо циклов, например, для задачи по нахождению факториала.

def f(num):

if num == 0:

return 1

return f(num-1) \* num

print(f(5)) # Выведет число 120

Рекурсию рекомендуется использовать только там, где это действительно необходимо. Интерпретатор Python автоматически выделяет память для выполняющейся функции, если вызовов самой себя будет слишком много, это приведёт к переполнению стека и аварийному завершению программы. Следующий код вызовет исключение “RecursionError”, которая показывает, что превышен максимальный лимит рекурсии.

def x(num):

a = num - 1

print(a)

x(a)

x(5)

Узнать максимальный лимит и его изменить можно с помощью getrecursionlimit() и setrecursionlimit(предел) из библиотеки [sys](https://all-python.ru/osnovy/sys.html).

Один из примеров применения рекурсии – это расчёт чисел Фибоначчи.

**Пустая функция**

Чтобы создать пустую функцию, нужно в её теле использовать оператор заглушки pass. Тогда она будет существовать и не выполнять никаких действий.

Такие функции могут использоваться для различных специфичных задач, например, при работе с классами, асинхронной отправкой форм.

Обычно программисты делают функцию пустой, когда хотят отложить её реализацию на потом (например, алгоритм её уже спланирован, но ещё не реализован, а запустить на выполнение код надо, чтобы проверить работоспособность остального кода).

Вот пример:

def example():

pass

**Области видимости**

Область видимости — важная составляющая любого языка программирования. С её помощью в одном модуле можно использовать одно и то же имя переменной несколько раз.

Области видимости также делают программу более безопасной, не позволяя получить доступ к важным переменным.

**В Python существует две области видимости**:

* Глобальная. Переменные объявляются непосредственно внутри модуля и доступны из любой точки программы.
* Локальная. Переменные объявляются в теле функции и доступны только внутри неё.

Важно понимать, из локальной области видимости можно обратить к переменным, находящимся в глобальной, но не наоборот.

Причём чтение глобальной переменной можно осуществить, просто обратившись к ней. А вот для записи надо отдельно указывать с помощью ключевого слова global, что мы работаем именно с глобальной переменной.

Вот пример:

val1 = "my global"

val2 = "another global"

def example():

print(val1) # Выведет my global

global val2

val2 = "change another global"

example()

print(val2) # Выведет change another global

**Замыкания: функции, возвращающие функции. Динамически сгенерированная функция**

Замыканием называется любая динамически сгенерированная функция, возвращенная из другой функции. Основное свойство замыкания состоит в том, что оно имеет доступ к переменным, определенным в том локальном пространстве имен, в котором было создано.

Пример:

**def make\_closure(a):**

**def closure():**

**print('Я знаю секрет: %d' % a)**

**return closure**

**closure = make\_closure(5)**

**closure()**

**Результат:**

**Я знаю секрет: 5**

Разница между замыканием и обычной функцией Python состоит в том, что замыкание сохраняет доступ к пространству имен (функции), в котором было создано, даже если создавшая ее функция уже завершилась.

Так, в примере выше, возвращенное замыкание печатает строку «Я знаю секрет: 5», в какой бы момент ее ни вызвать. Хотя обычно создают замыкания со статическим внутренним состоянием (в данном случае только значение a), ничто не мешает включить в состав состояния изменяемый объект – словарь, множество, список – и затем модифицировать его.

Например, ниже приведена функция, которая возвращает функцию, запоминающую, с какими аргументами вызывалась объемлющая функция:

**def make\_watcher():**

**have\_seen = {}**

**def has\_been\_seen(x):**

**if x in have\_seen:**

**return True**

**else:**

**have\_seen[x] = True**

**return False**

**return has\_been\_seen**

**watcher = make\_watcher()**

**vals = [5, 6, 1, 5, 1, 6, 3, 5]**

**print([watcher(x) for x in vals])**

**Результат:**

**[False, False, False, True, True, True, False, True]**

Однако следует иметь в виду одно техническое ограничение: изменять внутреннее состояние объектов (например, добавлять в словарь пары ключ-значение) можно, но связывать переменные в области видимости объемлющей функции – нельзя. Обойти это ограничение можно, например, путем модификации словаря или списка вместо присваивания значений переменным.

**def make\_counter():**

**count = [0]**

**def counter():**

**# увеличить счетчик и вернуть новое значение**

**count[0] += 1**

**return count[0]**

**return counter**

**counter = make\_counter()**

**print(counter())**

**Результат:**

**1**

**Каррирование: частичное фиксирование аргументов функции**

В информатике термином **каррирование** обозначается процедура порождения новых функций из существующих путем фиксирования некоторых аргументов.

Функция сложения двух чисел:

**def add\_numbers(x, y):**

**return x + y**

Мы можем породить на ее основе новую функцию одной переменной, **add\_five**, которая прибавляет к своему аргументу 5:

**add\_five = lambda y: add\_numbers(5, y)**

Говорят, что второй аргумент функции **add\_numbers** каррирован. Ничего особо примечательного здесь нет, поскольку мы просто определили новую функцию, которая вызывает существующую.

**def add\_numbers(x, y):**

**return x + y**

**add\_five = lambda y: add\_numbers(5, y)**

**print(add\_five(10))**

**Результат:**

**15**

Стандартный модуль **functools** упрощает эту процедуру за счет функции **partial**:

**from functools import partial**

**def add\_numbers(x, y):**

**return x + y**

**add\_five = partial(add\_numbers, 5)**

**print(add\_five(10))**

**Результат:**

**15**