

Функция в программировании представляет собой обособленный участок кода, который можно вызывать, обратившись к нему по имени, которым он был назван. При вызове происходит выполнение команд тела функции.

Функции можно сравнить с небольшими программками, которые сами по себе, т. е. автономно, не исполняются, а встраиваются в обычную программу. Нередко их так и называют – подпрограммы. Других ключевых отличий функций от программ нет. Функции также при необходимости могут получать и возвращать данные. Только обычно они их получают не с ввода (клавиатуры, файла и др.), а из вызывающей программы. Сюда же они возвращают результат своей работы.

Существует множество встроенных в язык программирования функций. С некоторыми такими в Python мы уже сталкивались. Это `print()`, `input()`, `int()`, `float()`, `str()`, `type()`. Код их тела нам не виден, он где-то "спрятан внутри языка". Нам же предоставляется только интерфейс – имя функции.

С другой стороны, программист всегда может определять свои функции. Их называют пользовательскими. В данном случае под "пользователем" понимают программиста, а не того, кто пользуется программой. Разберемся, зачем нам эти функции, и как их создавать.

Предположим, надо три раза подряд запрашивать на ввод пару чисел и складывать их. С этой целью можно использовать цикл:

```
i = 0
while i < 3:
    a = int(input())
    b = int(input())
    print(a+b)
    i += 1
```

Однако, что если перед каждым запросом чисел, надо выводить надпись, зачем они нужны, и каждый раз эта надпись разная. Мы не можем прервать цикл, а затем вернуться к тому же циклу обратно. Придется отказаться от него, и тогда получится длинный код, содержащий в разных местах одинаковые участки:

```
print("Сколько бананов и ананасов для обезьян?")
a = int(input())
b = int(input())
print("Всего", a+b, "шт.")

print("Сколько жуков и червей для ежей?")
a = int(input())
b = int(input())
print("Всего", a+b, "шт.")

print("Сколько рыб и моллюсков для выдр?")
a = int(input())
b = int(input())
print("Всего", a+b, "шт.")
```

Пример исполнения программы:

Сколько бананов и ананасов для обезьян?

15

5

Всего 20 шт.

Сколько жуков и червей для ежей?

50

12

Всего 62 шт.

Сколько рыб и моллюсков для выдр?

16

8

Всего 24 шт.

Внедрение функций позволяет решить проблему дублирования кода в разных местах программы. Благодаря им можно исполнять один и тот же участок кода не сразу, а только тогда, когда он понадобится.

Определение функции. Оператор def

В языке программирования Python функции определяются с помощью оператора `def`. Рассмотрим код:

```
def countFood():  
    a = int(input())  
    b = int(input())  
    print("Всего", a+b, "шт.")
```

Это пример определения функции. Как и другие сложные инструкции вроде условного оператора и циклов функция состоит из заголовка и тела. Заголовок оканчивается двоеточием и переходом на новую строку. Тело имеет отступ.

Ключевое слово `def` сообщает интерпретатору, что перед ним определение функции. За `def` следует имя функции. Оно может быть любым, также как и всякий идентификатор, например, переменная. В программировании весьма желательно давать всему осмысленные имена. Так в данном случае функция названа "посчитатьЕду" в переводе на русский.

После имени функции ставятся скобки. В приведенном примере они пустые. Это значит, что функция не принимает никакие данные из вызывающей ее программы. Однако она могла бы их принимать, и тогда в скобках были бы указаны так называемые параметры.

После двоеточия следует тело, содержащее инструкции, которые выполняются при вызове функции. Следует различать определение функции и ее вызов. В программном коде они не рядом и не вместе. Можно определить функцию, но ни разу ее не вызвать. Нельзя вызвать функцию, которая не была определена. Определив функцию, но ни разу не вызвав ее, вы никогда не выполните ее тела.

Вызов функции

Рассмотрим полную версию программы с функцией:

```
def countFood():
    a = int(input())
    b = int(input())
    print("Всего", a+b, "шт.")

print("Сколько бананов и ананасов для обезьян?")
countFood()

print("Сколько жуков и червей для ежей?")
countFood()

print("Сколько рыб и моллюсков для выдр?")
countFood()
```

После вывода на экран каждого информационного сообщения осуществляется вызов функции, который выглядит просто как упоминание ее имени со скобками. Поскольку в функцию мы ничего не передаем скобки опять же пустые. В приведенном коде функция вызывается три раза.

Когда функция вызывается, поток выполнения программы переходит к ее определению и начинает исполнять ее тело. После того, как тело функции исполнено, поток выполнения возвращается в основной код в то место, где функция вызывалась. Далее исполняется следующее за вызовом выражение.

В языке Python определение функции должно предшествовать ее вызовам. Это связано с тем, что интерпретатор читает код строка за строкой и о том, что находится ниже по течению, ему еще неизвестно. Поэтому если вызов функции предшествует ее определению, то возникает ошибка (выбрасывается исключение `NameError`):

```
print("Сколько бананов и ананасов для обезьян?")
countFood()

print("Сколько жуков и червей для ежей?")
countFood()

print("Сколько рыб и моллюсков для выдр?")
countFood()

def countFood():
    a = int(input())
    b = int(input())
    print("Всего", a+b, "шт.")
```

Результат:

```
Сколько бананов и ананасов для обезьян?
Traceback (most recent call last):
  File "test.py", line 2, in <module>
    countFood()
NameError: name 'countFood' is not defined
```

Для многих компилируемых языков это не обязательное условие. Там можно определять и вызывать функцию в произвольных местах программы. Однако для удобочитаемости кода программисты даже в этом случае предпочитают соблюдать определенные правила.

Функции придают программе структуру

Польза функций не только в возможности многократного вызова одного и того же кода из разных мест программы. Не менее важно, что благодаря им программа обретает истинную структуру. Функции как бы разделяют ее на обособленные части, каждая из которых выполняет свою конкретную задачу.

Пусть надо написать программу, вычисляющую площади разных фигур. Пользователь указывает, площадь какой фигуры он хочет вычислить. После этого вводит исходные данные. Например, длину и ширину в случае прямоугольника. Чтобы разделить поток выполнения на несколько ветвей, следует использовать оператор if-elif-else:

```
figure = input("1-прямоугольник, 2-треугольник, 3-круг: ")

if figure == '1':
    a = float(input("Ширина: "))
    b = float(input("Высота: "))
    print("Площадь: %.2f" % (a*b))
elif figure == '2':
    a = float(input("Основание: "))
    h = float(input("Высота: "))
    print("Площадь: %.2f" % (0.5 * a * h))
elif figure == '3':
    r = float(input("Радиус: "))
    print("Площадь: %.2f" % (3.14 * r**2))
else:
    print("Ошибка ввода")
```

Здесь нет никаких функций, и все прекрасно. Но напомним вариант с функциями:

```
def rectangle():
    a = float(input("Ширина: "))
    b = float(input("Высота: "))
    print("Площадь: %.2f" % (a*b))

def triangle():
    a = float(input("Основание: "))
    h = float(input("Высота: "))
    print("Площадь: %.2f" % (0.5 * a * h))

def circle():
    r = float(input("Радиус: "))
    print("Площадь: %.2f" % (3.14 * r**2))

figure = input("1-прямоугольник, 2-треугольник, 3-круг: ")
if figure == '1':
    rectangle()
elif figure == '2':
    triangle()
elif figure == '3':
    circle()
else:
    print("Ошибка ввода")
```

Он кажется сложнее, а каждая из трех функций вызывается всего один раз. Однако из общей логики программы как бы убраны и обособлены инструкции для нахождения площадей. Программа теперь состоит из отдельных "кирпичиков Лего". В основной ветке мы можем комбинировать их как угодно. Она играет роль управляющего механизма.

Если нам когда-нибудь захочется вычислять площадь треугольника по формуле Герона, а не через высоту, то не придется искать код во всей программе (представьте, что она состоит из тысяч строк кода как реальные программы). Мы пойдем к месту определения функций и изменим тело одной из них.

Если понадобится использовать эти функции в какой-нибудь другой программе, то мы сможем импортировать их туда, сославшись на данный файл с кодом (как это делается в Python, будет рассмотрено позже).

Практическая работа

В программировании можно из одной функции вызывать другую. Для иллюстрации этой возможности напишите программу по следующему описанию.

Основная ветка программы, не считая заголовков функций, состоит из одной строки кода. Это вызов функции `test()`. В ней запрашивается на ввод целое число. Если оно положительное, то вызывается функция `positive()`, тело которой содержит команду вывода на экран слова "Положительное". Если число отрицательное, то вызывается функция `negative()`, ее тело содержит выражение вывода на экран слова "Отрицательное".

Понятно, что вызов `test()` должен следовать после определения функций. Однако имеет ли значение порядок определения самих функций? То есть должны ли определения `positive()` и `negative()` предшествовать `test()` или могут следовать после него? Проверьте вашу гипотезу, поменяв объявления функций местами. Попробуйте объяснить результат.

В о з в р а т з н а ч е н и й и з ф у н к ц и и .

О п е р а т о р `return`

Функции могут передавать какие-либо данные из своих тел в основную ветку программы. Говорят, что функция возвращает значение. В большинстве языков программирования, в том числе Python, выход из функции и передача данных в то место, откуда она была вызвана, выполняется оператором `return`.

Если интерпретатор Питона, выполняя тело функции, встречает `return`, то он "забирает" значение, указанное после этой команды, и "уходит" из функции.

```
def cylinder():
    r = float(input())
    h = float(input())
    # площадь боковой поверхности цилиндра:
    side = 2 * 3.14 * r * h
    # площадь одного основания цилиндра:
    circle = 3.14 * r**2
    # полная площадь цилиндра:
    full = side + 2 * circle
    return full

square = cylinder()
print(square)
```

Пример выполнения:

```
3
7
188.4
```

В данной программе в основную ветку из функции возвращается значение локальной переменной `full`. Не сама переменная, а ее значение, в данном случае – какое-либо число, полученное в результате вычисления площади цилиндра.

В основной ветке программы это значение присваивается глобальной переменной `square`. То есть выражение `square = cylinder()` выполняется так:

1. Вызывается функция `cylinder()`.
2. Из нее возвращается значение.
3. Это значение присваивается переменной `square`.

Не обязательно присваивать результат переменной, его можно сразу вывести на экран:

```
...
print(cylinder())
```

Здесь число, полученное из `cylinder()`, непосредственно передается функции `print()`. Если мы в программе просто напишем `cylinder()`, не присвоив полученные данные переменной или не передав их куда-либо дальше, то эти данные будут потеряны. Но синтаксической ошибки не будет.

В функции может быть несколько операторов `return`. Однако всегда выполняется только один из них. Тот, которого первым достигнет поток выполнения. Допустим, мы решили обработать исключение, возникающее на некорректный ввод. Пусть тогда в ветке `except` обработчика исключений происходит выход из функции без всяких вычислений и передачи значения:

```
def cylinder():
    try:
        r = float(input())
        h = float(input())
    except ValueError:
        return
    side = 2 * 3.14 * r * h
    circle = 3.14 * r**2
    full = side + 2 * circle
    return full

print(cylinder())
```

Если попытаться вместо цифр ввести буквы, то сработает `return`, вложенный в `except`. Он завершит выполнение функции, так что все нижеследующие вычисления, в том числе `return full`, будут опущены. Пример выполнения:

```
r
None
```

Но постойте! Что это за слово `None`, которое нам вернул "пустой" `return`? Это ничего, такой объект – "ничто". Он принадлежит классу `NoneType`. До этого мы знали четыре типа данных, они же четыре класса: `int`, `float`, `str`, `bool`. Пришло время пятого.

Когда после `return` ничего не указывается, то по умолчанию считается, что там стоит объект `None`. Но никто вам не мешает явно написать `return None`.

Более того. Ранее мы рассматривали функции, которые вроде бы не возвращали никакого значения, потому что в них не было оператора `return`. На самом деле возвращали, просто мы не обращали на него внимание, не присваивали никакой переменной и не выводили на экран. В Python всякая функция что-либо возвращает. Если в ней нет оператора `return`, то она возвращает `None`. То же самое, как если в ней имеется "пустой" `return`.

Возврат нескольких значений

В Питоне позволительно возвращать из функции несколько объектов, перечислив их через запятую после команды `return`:

```
def cylinder():
    r = float(input())
    h = float(input())
    side = 2 * 3.14 * r * h
    circle = 3.14 * r**2
    full = side + 2 * circle
    return side, full

sCyl, fCyl = cylinder()
print("Площадь боковой поверхности %.2f" % sCyl)
print("Полная площадь %.2f" % fCyl)
```

Из функции `cylinder()` возвращаются два значения. Первое из них присваивается переменной `sCyl`, второе – `fCyl`. Возможность такого группового присвоения – особенность Python, обычно не характерная для других языков:

```
>>> a, b, c = 10, 15, 19
>>> a
10
>>> b
15
>>> c
19
```

Фокус здесь в том, что перечисление значений через запятую (например, `10, 15, 19`) создает объект типа `tuple`. На русский переводится как "кортеж". Это разновидность структур данных, которые будут изучены позже.

Когда же кортеж присваивается сразу нескольким переменным, то происходит сопоставление его элементов соответствующим в очереди переменным. Это называется распаковкой.

Таким образом, когда из функции возвращается несколько значений, на самом деле из нее возвращается один объект класса `tuple`. Перед возвратом эти несколько значений упаковываются в кортеж. Если же после оператора `return` стоит только одна переменная или объект, то ее/его тип сохраняется как есть.

Распаковка не является обязательной. Будет работать и так:

```
...  
print(cylinder())
```

Пример выполнения:

```
4  
3  
(75.36, 175.84)
```

На экран выводится кортеж, о чем говорят круглые скобки. Его также можно присвоить одной переменной, а потом вывести ее значение на экран.

Практическая работа

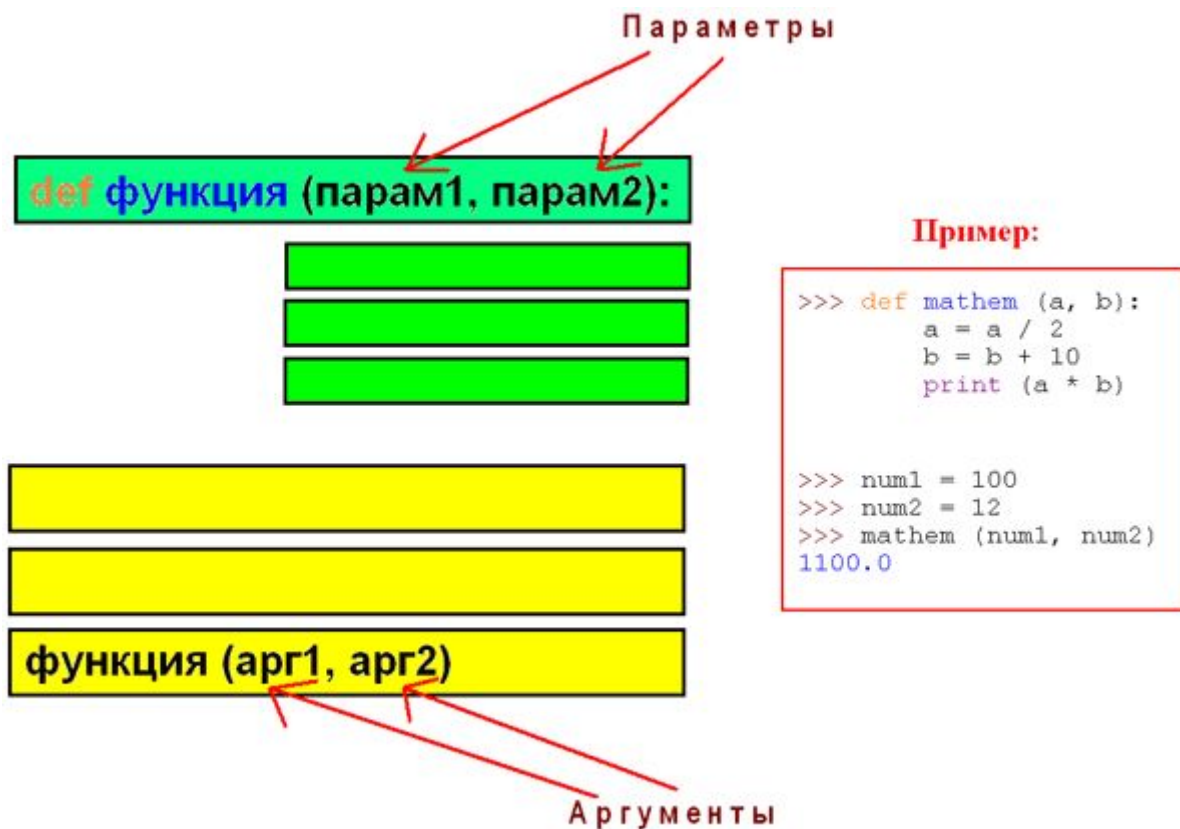
1. Напишите программу, в которой вызывается функция, запрашивающая с ввода две строки и возвращающая в программу результат их конкатенации. Выведите результат на экран.
2. Напишите функцию, которая считывает с клавиатуры числа и перемножает их до тех пор, пока не будет введен 0. Функция должна возвращать полученное произведение. Вызовите функцию и выведите на экран результат ее работы.

П а р а м е т р ы и а р г у м е н т ы ф у н к ц и и

В программировании функции могут не только возвращать данные, но также принимать их, что реализуется с помощью так называемых параметров, которые указываются в скобках в заголовке функции. Количество параметров может быть любым.

Параметры представляют собой локальные переменные, которым присваиваются значения в момент вызова функции. Конкретные значения, которые передаются в функцию при ее вызове, будем называть аргументами. Следует иметь в виду, что встречается иная терминология. Например, формальные параметры и фактические параметры. В Python же обычно все называют аргументами.

Рассмотрим схему и поясняющий ее пример:



Когда функция вызывается, то ей передаются аргументы. В примере указаны глобальные переменные `num1` и `num2`. Однако на самом деле передаются не эти переменные, а их значения. В данном случае числа 100 и 12. Другими словами, мы могли бы писать `mathem(100, 12)`. Разницы не было бы.

Когда интерпретатор переходит к функции, чтобы начать ее исполнение, он присваивает переменным-параметрам переданные в функцию значения-аргументы. В примере переменной `a` будет присвоено 100, `b` будет присвоено 12.

Изменение значений `a` и `b` в теле функции никак не скажется на значениях переменных `num1` и `num2`. Они останутся прежними. В Python такое поведение характерно для неизменяемых типов данных, к которым относятся, например, числа и строки. Говорят, что в функцию данные передаются по значению. Так, когда `a` присваивалось число 100, то это было уже другое число, не то, на которое ссылается переменная `num1`. Число 100 было скопировано и помещено в отдельную ячейку памяти для переменной `a`.

Существуют изменяемые типы данных. Для Питона, это, например, списки и словари. В этом случае данные передаются по ссылке. В функцию передается ссылка на них, а не сами данные. И эта ссылка связывается с локальной переменной. Изменения таких данных через локальную переменную обнаруживаются при обращении к ним через глобальную. Это есть следствие того, что несколько переменных ссылаются на одни и те же данные, на одну и ту же область памяти.

Необходимость передачи по ссылке связана в первую очередь с экономией памяти. Сложные типы данных, по сути представляющие собой структуры данных, обычно копировать не целесообразно. Однако, если надо, всегда можно сделать это принудительно.

Произвольное количество аргументов

Обратим внимание еще на один момент. Количество аргументов и параметров совпадает. Нельзя передать три аргумента, если функция принимает только два. Нельзя передать один аргумент, если функция требует два обязательных. В рассмотренном примере они обязательные.

Однако в Python у функций бывают параметры, которым уже присвоено значение по-умолчанию. В таком случае, при вызове можно не передавать соответствующие этим параметрам аргументы. Хотя можно и передать. Тогда значение по умолчанию заменится на переданное.

```
def cylinder(h, r = 1):  
    side = 2 * 3.14 * r * h  
    circle = 3.14 * r**2  
    full = side + 2 * circle  
    return full
```

```
figure1 = cylinder(4, 3)  
figure2 = cylinder(5)  
print(figure1)  
print(figure2)
```

Вывод:

```
131.88  
37.68
```

При втором вызове `cylinder()` мы указываем только один аргумент. Он будет присвоен переменной-параметру `h`. Переменная `r` будет равна 1.

Согласно правилам синтаксиса Python при определении функции параметры, которым присваивается значение по-умолчанию должны следовать (находиться сзади) за параметрами, не имеющими значений по умолчанию.

А вот при вызове функции, можно явно указывать, какое значение соответствует какому параметру. В этом случае их порядок не играет роли:

```
...  
figure3 = cylinder(10, 2)  
figure4 = cylinder(r=2, h=10)  
print(figure3)  
print(figure4)
```

В данном случае оба вызова – это вызовы с одними и теми же аргументами-значениями. Просто в первом случае сопоставление параметрам-переменным идет в порядке следования. Во-втором случае – по ключам, которыми выступают имена параметров.

В Python определения и вызовы функций имеют и другие нюансы, рассмотрение которых мы пока опустим, так как они требуют более глубоких знаний, чем у нас есть на данный момент. Скажем лишь, что функции может

быть определена так, что в нее можно передать хоть ни одного аргумента, хоть множество:

```
def oneOrMany(*a):  
    print(a)  
  
oneOrMany(1)  
oneOrMany('1', 1, 2, 'abc')  
oneOrMany()
```

Результат:

```
(1,)  
('1', 1, 2, 'abc')  
()
```

Опять же, судя по скобкам, здесь возникает упомянутый в прошлом уроке кортеж.

Практическая работа

Напишите программу, в которой определены следующие четыре функции:

1. Функция `getInput()` не имеет параметров, запрашивает ввод с клавиатуры и возвращает в основную программу полученную строку.
2. Функция `testInput()` имеет один параметр. В теле она проверяет, можно ли переданное ей значение преобразовать к целому числу. Если можно, возвращает логическое `True`. Если нельзя – `False`.
3. Функция `strToInt()` имеет один параметр. В теле преобразовывает переданное значение к целочисленному типу. Возвращает полученное число.
4. Функция `printInt()` имеет один параметр. Она выводит переданное значение на экран и ничего не возвращает.

В основной ветке программы вызовите первую функцию. То, что она вернула, передайте во вторую функцию. Если вторая функция вернула `True`, то те же данные (из первой функции) передайте в третью функцию, а возвращенное третьей функцией значение – в четвертую.

- 1) Напишите цикл, который должен “пробежаться” по цифрам от 0 до 100 и записать числа, которые делятся на 3 без остатка в список **three**, а числа, которые делятся на 5 без остатков в список **five**. Числа, которые делятся без остатков как на 3, так и на 5, запишите в список **three_five**. Используйте паттерн “аккумулятор” и деление по модулю.
- 2) Напишите функцию, которая берет два аргумента (**x** и **y**) и делит их друг на друга. То есть, делит **x** на **y**. Оберните тело функции конструкцией `try/except`, чтобы в случае возникновения арифметических ошибок, таких как деление на ноль, выдавался результат 0.

- 3) Напишите функцию, которая принимает цифру и возвращает список от 0 до (не включительно) переданной цифры. Например, если дана цифра 3, то функция должна возвращать [0,1,2].