Python Functions (With Examples)

programiz.com/python-programming/function

Python Functions

A function is a block of code that performs a specific task.

Suppose we need to create a program to make a circle and color it. We can create two functions to solve this problem:

1. function to create a circle
2. function to color the shape

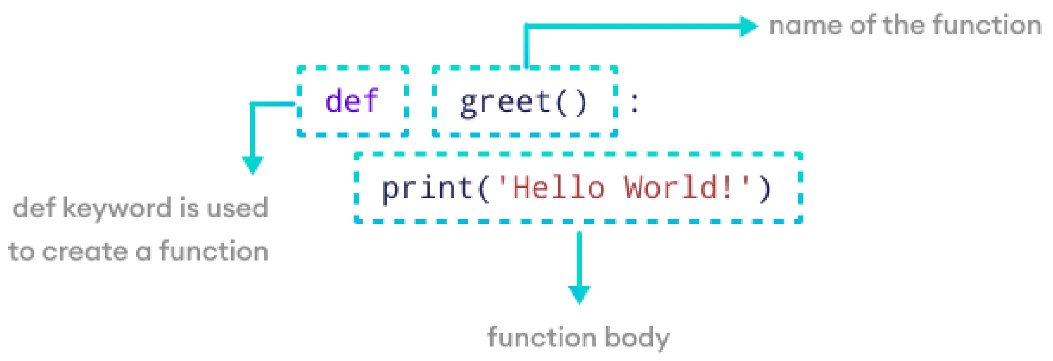
Dividing a complex problem into smaller chunks makes our program easy to understand and reuse.

Create a Function

Let's create our first function.

```
def greet():
    print('Hello World!')
```

Here are the different parts of the program:



Create a Python Function

Here, we have created a simple function named `greet()` that prints **Hello World!**

Note: When writing a function, pay attention to indentation, which are the spaces at the start of a code line.

In the above code, the `print()` statement is indented to show it's part of the function body, distinguishing the function's definition from its body.

Calling a Function

In the above example, we have declared a function named `greet()`.

```
def greet():
    print('Hello World!')
```

If we run the above code, we won't get an output.

It's because creating a function doesn't mean we are executing the code inside it. It means the code is there for us to use if we want to.

To use this function, we need to call the function.

Function Call

```
greet()
```

Example: Python Function Call

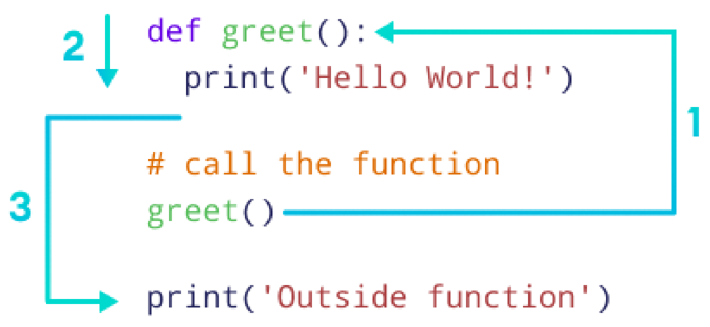
```
def greet():
    print('Hello World!')

# call the function
greet()

print('Outside function')
```

Run Code

Output

```
Hello World!
Outside function
```

In the above example, we have created a function named `greet()`. Here's how the control of the program flows:



Working of Python Function

Here,

1. When the function `greet()` is called, the program's control transfers to the function definition.
2. All the code inside the function is executed.
3. The control of the program jumps to the next statement after the function call.

Python Function Arguments

Arguments are inputs given to the function.

```
def greet(name):
    print("Hello", name)

# pass argument
greet("John")
```

Run Code

Sample Output 1

```
Hello John
```

Here, we passed '`John'` as an argument to the `greet()` function.

We can pass different arguments in each call, making the function re-usable and dynamic.

Let's call the function with a different argument.

```
greet("David")
```

Sample Output 2

```
Hello David
```

Example: Function to Add Two Numbers

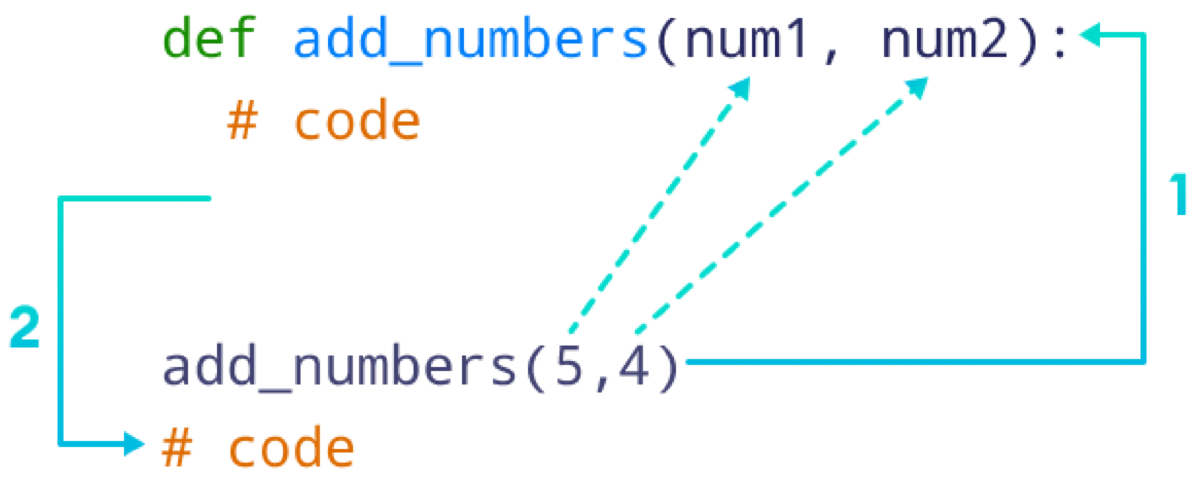
```
# function with two arguments
def add_numbers(num1, num2):
    sum = num1 + num2
    print("Sum: ", sum)

# function call with two values
add_numbers(5, 4)
```

Run Code

Output

```
Sum: 9
```

In the above example, we have created a function named `add_numbers()` with arguments: *num1* and *num2*.



Python Function with Arguments

Parameters

Parameters are the variables listed inside the parentheses in the function definition. They act like placeholders for the data the function can accept when we call them.

Think of parameters as the **blueprint** that outlines what kind of information the function expects to receive.

```
def print_age(age):  # age is a parameter
    print(age)
```

In this example, the `print_age()` function takes `age` as its input. However, at this stage, the actual value is not specified.

The `age` parameter is just a placeholder waiting for a specific value to be provided when the function is called.

Arguments

Arguments are the actual values that we pass to the function when we call it.

Arguments replace the parameters when the function executes.

```
print_age(25)  # 25 is an argument
```

Here, during the function call, the argument **25** is passed to the function.

The return Statement

We return a value from the function using the `return` statement.

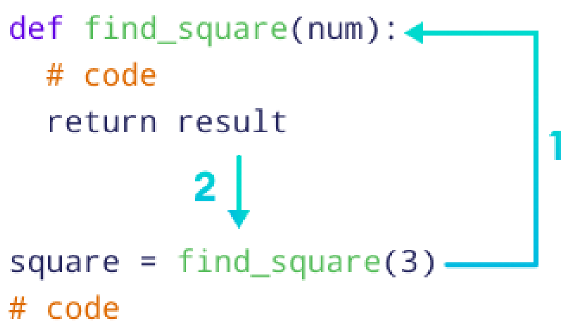
```
# function definition
def find_square(num):
    result = num * num
    return result


# function call
square = find_square(3)

print('Square:', square)
```

Run Code

Output

```
Square: 9
```

In the above example, we have created a function named `find_square()`. The function accepts a number and returns the square of the number.



Working of functions in Python

Note: The `return` statement also denotes that the function has ended. Any code after `return` is not executed.

The pass Statement

The `pass` statement serves as a placeholder for future code, preventing errors from empty code blocks.

It's typically used where code is planned but has yet to be written.

```
def future_function():
    pass

# this will execute without any action or error
future_function()
```

Run Code

Note: To learn more, visit Python Pass Statement.

Python Library Functions

Python provides some built-in functions that can be directly used in our program.

We don't need to create the function, we just need to call them.

Some Python library functions are:

These library functions are defined inside the module. And to use them, we must include the module inside our program.

For example, `sqrt()` is defined inside the math module.

Note: To learn more about library functions, please visit Python Library Functions.

Example: Python Library Function

```
import math

# sqrt computes the square root
square_root = math.sqrt(4)

print("Square Root of 4 is",square_root)

# pow() comptes the power
power = pow(2, 3)

print("2 to the power 3 is",power)
```

Run Code

Output

```
Square Root of 4 is 2.0
2 to the power 3 is 8
```

Here, we imported a `math` module to use the library functions `sqrt()` and `pow()`.

More on Python Functions

In Python, functions are divided into two categories: user-defined functions and standard library functions. These two differ in several ways:

User-Defined Functions

These are the functions we create ourselves. They're like our custom tools, designed for specific tasks we have in mind.

They're not part of Python's standard toolbox, which means we have the freedom to tailor them exactly to our needs, adding a personal touch to our code.

Standard Library Functions

Think of these as Python's pre-packaged gifts. They come built-in with Python, ready to use.

These functions cover a wide range of common tasks such as mathematics, file operations, working with strings, etc.

They've been tried and tested by the Python community, ensuring efficiency and reliability.

Python allows functions to have default argument values. Default arguments are used when no explicit values are passed to these parameters during a function call.

Let's look at an example.

```
def greet(name, message="Hello"):
    print(message, name)

# calling function with both arguments
greet("Alice", "Good Morning")

# calling function with only one argument
greet("Bob")
```

Run Code
Output

```
Good Morning Alice
Hello Bob
```

Here, `message` has the default value of `Hello`. When `greet()` is called with only one argument, `message` uses its default value.

Note: To learn more about default arguments in a function, please visit Python Function Arguments.

We can handle an arbitrary number of arguments using special symbols `*args` and `**kwargs`.

`*args` in Functions

Using `*args` allows a function to take any number of positional arguments.

```
# function to sum any number of arguments
def add_all(*numbers):
    return sum(numbers)

# pass any number of arguments
print(add_all(1, 2, 3, 4))
```

Run Code

Output

```
10
```

`*kwargs` in Functions

Using `**kwargs` allows the function to accept any number of keyword arguments.

```
# function to print keyword arguments
def greet(**words):
    for key, value in words.items():
        print(f"{key}: {value}")

# pass any number of keyword arguments
greet(name="John", greeting="Hello")
```

Run Code

Output

```
name: John
greeting: Hello
```

Python Lambda/ Function (With Examples)

Python Lambda/Anonymous Function

In Python, a lambda function is a special type of function without the function name. For example,

```
lambda : print('Hello World')
```

Here, we have created a lambda function that prints `'Hello World'`.

Before you learn about lambdas, make sure to know about Python Functions.

Python Lambda Function Declaration

We use the `lambda` keyword instead of `def` to create a lambda function. Here's the syntax to declare the lambda function:

```
lambda argument(s) : expression
```

Here,

- `argument(s)` - any value passed to the lambda function
- `expression` - expression is executed and returned

Let's see an example,

```
greet = lambda : print('Hello World')
```

Here, we have defined a lambda function and assigned it to the variable named *greet*.

To execute this lambda function, we need to call it. Here's how we can call the lambda function

```
# call the lambda
greet()
```

The lambda function above simply prints the text `'Hello World'`.

Note: This lambda function doesn't have any argument.

Example: Python Lambda Function

```
# declare a lambda function
greet = lambda : print('Hello World')

# call lambda function
greet()

# Output: Hello World
```

Run Code

In the above example, we have defined a lambda function and assigned it to the *greet* variable.

When we call the lambda function, the print() statement inside the lambda function is executed.

Python lambda Function with an Argument

Similar to normal functions, a `lambda` function can also accept arguments. For example,

```
# lambda that accepts one argument
greet_user = lambda name : print('Hey there,', name)

# lambda call
greet_user('Delilah')

# Output: Hey there, Delilah
```

Run Code

In the above example, we have assigned a lambda function to the *greet_user* variable.

Here, `name` after the `lambda` keyword specifies that the lambda function accepts the argument named `name`.

Notice the call of the lambda function,

```
greet_user('Delilah')
```

Here, we have passed a string value `'Delilah'` to our lambda function.

Finally, the statement inside the lambda function is executed.

Frequently Asked Questions

The filter() function in Python takes in a function and an iterable (lists, tuples, and strings) as arguments.

The function is called with all the items in the list, and a new list is returned, which contains items for which the function evaluates to `True`.

Let's see an example,

```
# program to filter out only the even items from a list
my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(filter(lambda x: (x%2 == 0) , my_list))

print(new_list)

# Output: [4, 6, 8, 12]
```

Run Code

Here, the `filter()` function returns only even numbers from a list.

The map() function in Python takes in a function and an iterable (lists, tuples, and strings) as arguments.

The function is called with all the items in the list, and a new list is returned, which contains items returned by that function for each item.

Let's see an example,

```
# Program to double each item in a list using map()

my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(map(lambda x: x * 2 , my_list))

print(new_list)

# Output: [2, 10, 8, 12, 16, 22, 6, 24]
```

Run Code

Here, the `map()` function doubles all the items in a list.

Python Generators

programiz.com/python-programming/generator

In Python, a generator is a function that returns an iterator that produces a sequence of values when iterated over.

Generators are useful when we want to produce a large sequence of values, but we don't want to store all of them in memory at once.

Create Python Generator

In Python, similar to defining a normal function, we can define a generator function using the `def` keyword, but instead of the `return` statement we use the `yield` statement.

```
def generator_name(arg):
    # statements
    yield something
```

Here, the `yield` keyword is used to produce a value from the generator.

When the generator function is called, it does not execute the function body immediately. Instead, it returns a generator object that can be iterated over to produce the values.

Example: Python Generator

Here's an example of a generator function that produces a sequence of numbers,

```
def my_generator(n):

    # initialize counter
    value = 0

    # loop until counter is less than n
    while value < n:

        # produce the current value of the counter
        yield value

        # increment the counter
        value += 1

# iterate over the generator object produced by my_generator
for value in my_generator(3):

    # print each value produced by generator
    print(value)
```

Run Code

Output

```
0
1
2
```

In the above example, the `my_generator()` generator function takes an integer `n` as an argument and produces a sequence of numbers from **0** to `n-1` using while loop.

The `yield` keyword is used to produce a value from the generator and pause the generator function's execution until the next value is requested.

The `for` loop iterates over the generator object produced by `my_generator()`, and the print statement prints each value produced by the generator.

We can also create a generator object from the generator function by calling the function like we would any other function as,

```
generator = my_range(3)
print(next(generator))  # 0
print(next(generator))  # 1
print(next(generator))  # 2
```

Note: To learn more, visit range() and for loop().

Python Generator Expression

In Python, a generator expression is a concise way to create a generator object.

It is similar to a list comprehension, but instead of creating a list, it creates a generator object that can be iterated over to produce the values in the generator.

Generator Expression Syntax

A generator expression has the following syntax,

```
(expression for item in iterable)
```

Here, `expression` is a value that will be returned for each item in the `iterable`.

The generator expression creates a generator object that produces the values of `expression` for each item in the `iterable`, one at a time, when iterated over.

Example 2: Python Generator Expression

```
# create the generator object
squares_generator = (i * i for i in range(5))

# iterate over the generator and print the values
for i in squares_generator:
    print(i)
```

Run Code

Output

```
0
1
4
9
16
```

Here, we have created the generator object that will produce the squares of the numbers **0** through **4** when iterated over.

And then, to iterate over the generator and get the values, we have used the `for` loop.

Use of Python Generators

There are several reasons that make generators a powerful implementation.

1. Easy to Implement

Generators can be implemented in a clear and concise way as compared to their iterator class counterpart. Following is an example to implement a sequence of power of **2** using an iterator class.

```
class PowTwo:
    def __init__(self, max=0):
        self.n = 0
        self.max = max

    def __iter__(self):
        return self

    def __next__(self):
        if self.n > self.max:
            raise StopIteration

        result = 2 ** self.n
        self.n += 1
        return result
```

The above program was lengthy and confusing. Now, let's do the same using a generator function.

```
def PowTwoGen(max=0):
    n = 0
    while n < max:
        yield 2 ** n
        n += 1
```

Since generators keep track of details automatically, the implementation was concise and much cleaner.

2. Memory Efficient

A normal function to return a sequence will create the entire sequence in memory before returning the result. This is an overkill, if the number of items in the sequence is very large.

Generator implementation of such sequences is memory friendly and is preferred since it only produces one item at a time.

3. Represent Infinite Stream

Generators are excellent mediums to represent an infinite stream of data. Infinite streams cannot be stored in memory, and since generators produce only one item at a time, they can represent an infinite stream of data.

The following generator function can generate all the even numbers (at least in theory).

```
def all_even():
    n = 0
    while True:
        yield n
        n += 2
```

4. Pipelining Generators

Multiple generators can be used to pipeline a series of operations. This is best illustrated using an example.

Suppose we have a generator that produces the numbers in the Fibonacci series. And we have another generator for squaring numbers.

If we want to find out the sum of squares of numbers in the Fibonacci series, we can do it in the following way by pipelining the output of generator functions together.

```
def fibonacci_numbers(nums):
    x, y = 0, 1
    for _ in range(nums):
        x, y = y, x+y
        yield x

def square(nums):
    for num in nums:
        yield num**2

print(sum(square(fibonacci_numbers(10))))

# Output: 4895
```

Run Code

This pipelining is efficient and easy to read (and yes, a lot cooler!).

functions-in-python

May 11, 2024

1 Functions in Python - Learn by Solving 10 problems

1.1 10 Problems

1. Basic Function Syntax Problem: Write a function to calculate and return the square of a number.

2. Function with Multiple Parameters Problem: Create a function that takes two numbers as parameters and returns their sum.

3. Polymorphism in Functions Problem: Write a function multiply that multiplies two numbers, but can also accept and multiply strings.

4. Function Returning Multiple Values Problem: Create a function that returns both the area and circumference of a circle given its radius.

5. Default Parameter Value Problem: Write a function that greets a user. If no name is provided, it should greet with a default name.

6. Lambda Function Problem: Create a lambda function to compute the cube of a number.

7. Function with *args Problem: Write a function that takes variable number of arguments and returns their sum.

8. Function with **kwargs Problem: Create a function that accepts any number of keyword arguments and prints them in the format key: value.

9. Generator Function with yield Problem: Write a generator function that yields even numbers up to a specified limit.

10. Recursive Function Problem: Create a recursive function to calculate the factorial of a number.

1. Basic Function Syntax Problem: Write a function to calculate and return the square of a number.

[ ]:
```python
def square_of_num(number):
  return number ** 2

result=square_of_num(5)
print(result)
```

```
25
```

2. Function with Multiple Parameters Problem: Create a function that takes two numbers as parameters and returns their sum.

[ ]:
```
def sum_of_two(a,b):
  return a+b

print(sum_of_two(5,6))
```

```
11
```

3. Polymorphism in Functions Problem: Write a function multiply that multiplies two numbers, but can also accept and multiply strings.

[ ]:
```
def multiply(pOne,pTwo):
  return pOne*pTwo

print(multiply(5,5))
print(multiply(5,"a"))
print(multiply("a",5))
# print(multiply("a","5")) //error - TypeError: can't multiply sequence by␣
 ↪non-int of type 'str'
```

```
25
aaaaa
aaaaa
```

4. Function Returning Multiple Values Problem: Create a function that returns both the area and circumference of a circle given its radius.

[ ]:
```
import math
def circle_calc(r):
  circumference = round(2*math.pi*r,2)
  area = round(math.pi*r*r,2)
  return area,circumference

a,c=circle_calc(7)

print("Area: ",a, "Circumference: ",c)
```

```
Area:  153.94 Circumference:  43.98
```

5. Default Parameter Value Problem: Write a function that greets a user. If no name is provided, it should greet with a default name.

[ ]:
```
def greet(name="User"):
  return "Hello, "+name+" "

print(greet("Gautam"))
print(greet())
```

```
Hello, Gautam
Hello, User
```

6. Lambda Function Problem: Create a lambda function to compute the cube of a number.

[ ]:
```
cube = lambda x: x ** 3
print(cube(3))
```

```
27
```

7. Function with *args Problem: Write a function that takes variable number of arguments and returns their sum.

[ ]:
```
def sum_all(*args):
  return sum(args) # using sum function

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

```
3
6
15
```

[ ]:
```
def sum_all(*args):
  print(*args)
  print(args) #tuple - which is iterable
  sum = 0
  for i in args: # don't use *args here
    sum = sum+i
  return sum

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

```
1 2
(1, 2)
3
1 2 3
(1, 2, 3)
6
1 2 3 4 5
(1, 2, 3, 4, 5)
15
```

8. Function with **kwargs Problem: Create a function that accepts any number of keyword arguments and prints them in the format key: value.

[ ]:
```
def print_kwargs(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
```

```python



print_kwargs(name="shaktiman", power="lazer")
print_kwargs(name="shaktiman")
print_kwargs(name="shaktiman", power="lazer", enemy = "Dr. Jackaal")
```

```
name: shaktiman
power: lazer
name: shaktiman
name: shaktiman
power: lazer
enemy: Dr. Jackaal
```

9. Generator Function with yield Problem: Write a generator function that yields even numbers up to a specified limit.

[ ]:
```python
def even_generator(limit):
    for i in range(2, limit + 1, 2):
        yield i



for num in even_generator(10):
    print(num)
```

```
2
4
6
8
10
```

10. Recursive Function Problem: Create a recursive function to calculate the factorial of a number.

[ ]:
```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5))
```

```
120
```