Python Functions

What is a function

A function is a named code block that performs a job or returns a value.

Why do you need functions in Python

Sometimes, you need to perform a task multiple times in a program. And you don’t want to copy the code for that same task all over places.

To do so, you wrap the code in a function and use this function to perform the task whenever you need it.

For example, whenever you want to display a value on the screen, you need to call the print() function. Behind the scene, Python runs the code inside the print() function to display a value on the screen.

In practice, you use functions to divide a large program into smaller and more manageable parts. The functions will make your program easier to develop, read, test, and maintain.

The print() function is one of many built-in functions in Python. It means that these functions are available everywhere in the program.

you’ll learn how to define user-defined Python functions.

Defining a Python function

Here’s a simple function that shows a greeting:

def greet():

""" Display a greeting to users """

print('Hi')

This example shows the simplest structure of a function. A function has two main parts: a function definition and body.

**1) Function definition**

A function definition starts with the def keyword and the name of the function (greet).

If the function needs some information to do its job, you need to specify it inside the parentheses (). The greet function in this example doesn’t need any information, so its parentheses are empty.

The function definition always ends in a colon (:).

**2) Function body**

All the indented lines that follow the function definition make up the function’s body.

The text string surrounded by triple quotes is called a [docstring](https://www.pythontutorial.net/python-basics/python-function-docstrings/). It describes what the function does. Python uses the docstring to generate documentation for the function automatically.

The line print('Hi') is the only line of actual code in the function body. The greet() function does one task: print('Hi').

**Calling a function**

When you want to use a function, you need to call it. A function call instructs Python to execute the code inside the function.

To call a function, you write the function’s name, followed by the information that the function needs in parentheses.

The following example calls the greet() function. Since the greet() function doesn’t need any information, you need to specify empty parentheses like this:

greet()

If you run the program, it’ll show a greeting on the screen:

Hi

**Passing information to Python functions**

Suppose that you want to greet users by their names. To do it, you need to specify a name in parentheses of the function definition as follows:

def greet(name):

The name is called a function parameter or simply a parameter.

When you add a parameter to the function definition, you can use it as a variable inside the function body:

def greet(name):

print(f"Hi {name}")

And you can access the name parameter only within the body of the greet() function, not the outside.

When you call a function with a parameter, you need to pass the information. For example:

greet('John')

**Output:**

Hi John

The value that you pass into a function is called an argument. In this example 'John' is an argument.

Also, you **can call the function by passing a variable into it**:

first\_name = 'Jane'

greet(first\_name)

In this example, the first\_name variable is also the argument of the greet() function.

**Parameters vs. Arguments**

Sometimes, parameters and arguments are used interchangeably. It’s important to distinguish between the parameters and arguments of a function.

A parameter is a piece of information that a function needs. And you specify the parameter in the function definition. For example, the greet() function has a parameter called name.

An argument is a piece of data that you pass into the function. For example, the text string 'John' or the variable jane is the function argument.

**Returning a value**

A function can perform a task like the greet() function. Or it can return a value. The value that a function returns is called a **return value**.

To return a value from a function, you use the return statement inside the function body.

return value

The following example modifies the greet() function to return a greeting instead of displaying it on the screen:

def greet(name):

return f"Hi {name}"

When you call the greet() function, you can assign its return value to a variable:

greeting = greet('John')

And show it on the screen:

print(greeting)

The new greet() function is better than the old one because it doesn’t depend on the print() function.

Later, you can reuse the greet() function in other applications. For example, you can use it in a web application to greet users after they log in.

**Python functions with multiple parameters**

A function can have zero, one, or multiple parameters.

The following example defines a function called sum() that calculates the sum of two numbers:

def sum(a, b):

return a + b

total = sum(10,20)

print(total)

**Output:**

30

In this example, the sum() function has two parameters a and b, and returns the sum of them.

When a function has multiple parameters, you need to use a comma to separate them.

When you call the function, you need to pass all the arguments. If you pass more or fewer arguments to the function, you’ll get an error.

In the following function call, a will be 10 and b will be 20 inside the function body.

total = sum(10, 20)

**Summary**

* A Python function is a reusable named block of code that performs a task or returns a value.
* Use the def keyword to define a new function. A function consists of function definition and body.
* A function can have zero or more parameters. If a function has one or more parameters, you need to pass the same number of arguments into it.
* A function can perform a job or return a value. Use the return statement to return a value from a function.

# **Python Default Parameters**

## **Introduction to Python default parameters**

When you [define a function](https://www.pythontutorial.net/python-basics/python-functions/), you can specify a default value for each parameter.

To specify default values for parameters, you use the following syntax:

def function\_name(param1, param2=value2, param3=value3, ...):

In this syntax, you specify default values (value2, value3, …) for each parameter using the assignment operator (=).

When you call a function and pass an argument to the parameter that has a default value, the function will use that argument instead of the default value.

However, if you don’t pass the argument, the function will use the default value.

To use default parameters, you need to place parameters with the default values after other parameters. Otherwise, you’ll get a syntax error.

For example, you cannot do something like this:

def function\_name(param1=value1, param2, param3):

This causes a syntax error.

## **Python default parameters example**

The following example defines the greet() function that returns a greeting message:

def greet(name, message='Hi'):

return f"{message} {name}"

The greet() function has two parameters: name and message. And the message parameter has a default value of 'Hi'.

The following calls the greet() function and passes the two arguments:

def greet(name, message='Hi'):

return f"{message} {name}"

greeting = greet('John', 'Hello')

print(greeting)

Output:

Hello John

Since we pass the second argument to the greet() function, the function uses the argument instead of the default value.

The following example calls the greet() function without passing the second argument:

def greet(name, message='Hi'):

return f"{message} {name}"

greeting = greet('John')

print(greeting)

**Output:**

Hi John

In this case, the greet() function uses the default value of the message parameter.

## **Multiple default parameters**

The following redefines the greet() function with the two parameters that have default values:

def greet(name='there', message='Hi'):

return f"{message} {name}"

In this example, you can call the greet() function without passing any parameters:

def greet(name='there', message='Hi'):

return f"{message} {name}"

greeting = greet()

print(greeting)

Output:

Hi there

Suppose that you want the greet() function to return a greeting like Hello there. You may come up with the following function call:

def greet(name='there', message='Hi'):

return f"{message} {name}"

greeting = greet('Hello')

print(greeting)

Unfortuntely, it returns an unexpected value:

Hi Hello

Because when you pass the 'Hello' argument, the greet() function treats it as the first argument, not the second one.

To resolve this, you need to call the greet() function using [keyword arguments](https://www.pythontutorial.net/python-basics/python-keyword-arguments/) like this:

def greet(name='there', message='Hi'):

return f"{message} {name}"

greeting = greet(message='Hello')

print(greeting)

Output:

Hello there

## **Summary**

* Use Python default parameters to simplify the function calls.
* Place default parameters after the non-default parameters.

# **Python Keyword Arguments**

**Summary**: in this tutorial, you’ll learn about the Python keyword arguments, and how to use them to make function calls more obvious.

## **Introduction to the Python keyword arguments**

Let’s start with a simple [function](https://www.pythontutorial.net/python-basics/python-functions/) that calculates the net price from the selling price and discount:

def get\_net\_price(price, discount):

return price \* (1-discount)

The get\_net\_price() function has two parameters: price and discount.

The following shows how to call the get\_net\_price() function to calculate the net price from the price 100 and discount 10%:

net\_price = get\_net\_price(100, 0.1)

print(net\_price)

Output:

90.0

In the get\_net\_price(100, 0.1) function call, we pass each argument as a positional argument. In other words, we pass the price argument first and the discount argument second.

However, the function call get\_net\_price(100, 0.1) has a readability issue. Because by looking at that function call only, you don’t know which argument is price and which one is the discount.

On top of that, when you call the get\_net\_price() function, you need to know the position of each argument.

If you don’t, the function will calculate the net\_price incorrectly. For example:

net\_price = get\_net\_price(0.1, 100)

print(net\_price)

Output:

-9.9

To improve the readability, Python introduces the keyword arguments.

The following shows the keyword argument syntax:

fn(parameter1=value1,parameter2=value2)

By using the keyword argument syntax, you don’t need to specify the arguments in the same order as defined in the function.

Therefore, you can call a function by swapping the argument positions like this:

fn(parameter2=value2,parameter1=value1)

The following shows how to use the keyword argument syntax to call the get\_net\_price() function:

net\_price = get\_net\_price(price=100, discount=0.1)

Or:

net\_price = get\_net\_price(discount=0.1, price=100)

Both of them returns the same result.

When you use the keyword arguments, their names that matter, not their positions.

Note that you can call a function by mixing positional and keyword arguments. For example:

net\_price = get\_net\_price(100, discount=0.1)

## **Keyword arguments and default parameters**

Suppose that you have the following get\_net\_price() function that calculates the net price from the selling price, tax, and discount.

def get\_net\_price(price, tax=0.07, discount=0.05):

return price \* (1 + tax - discount)

In the get\_net\_price() function, the tax and discount [parameters have default values](https://www.pythontutorial.net/python-basics/python-default-parameters/) of 7% and 5% respectively.

The following calls the get\_net\_price() function and uses the default values for tax and discount parameters:

net\_price = get\_net\_price(100)

print(net\_price)

Output:

102.0

Suppose that you want to use the default value for the tax parameter but not discount. The following function call doesn’t work correctly.

net\_price = get\_net\_price(100, 0.06)

… because Python will assign 100 to price and 0.1 to tax, not discount.

To fix this, you must use keyword arguments:

net\_price = get\_net\_price(price=100, discount=0.06)

print(net\_price)

Output:

101.0

Or you can mix the positional and keyword arguments:

net\_price = get\_net\_price(100, discount=0.06)

print(net\_price)

Output:

101.0

## **Python keyword argument requirements**

Once you use a keyword argument, you need to use keyword arguments for the remaining parameters.

The following will result in an error because it uses the positional argument after a keyword argument:

net\_price = get\_net\_price(100, tax=0.08, 0.06)

Error:

SyntaxError: positional argument follows keyword argument

To fix this, you need to use the keyword argument for the third argument like this:

net\_price = get\_net\_price(100, tax=0.08, discount=0.06)

print(net\_price)

## **Summary**

* Use the Python keyword arguments to make your function call more readable and obvious, especially for functions that accept many arguments.
* All the arguments after the first keyword argument must also be keyword arguments too.

# **Python Recursive Functions**

## **Introduction to recursive functions**

A recursive function is a [function](https://www.pythontutorial.net/python-basics/python-functions/) that calls itself until it doesn’t.

The following fn() function is a recursive function because it has a call to itself:

def fn():

*# ...*

fn()

*# ...*

In the fn() function, the #... means other code.

Also, a recursive function needs to have a condition to stop calling itself. So you need to add an [if statement](https://www.pythontutorial.net/python-basics/python-if/) like this:

def fn():

*# ...*

if condition:

*# stop calling itself*

else:

fn()

*# ...*

Typically, you use a recursive function to divide a big problem that’s difficult to solve into smaller problems that are easier to solve.

In programming, you’ll often find the recursive functions used in data structures and algorithms like trees, graphs, and binary searches.

## **Python recursive function examples**

Let’s take some examples of using Python recursive functions.

### **1) A simple recursive function example in Python**

Suppose you need to develop a countdown function that counts down from a specified number to zero.

For example, if you call the function that counts down from 3, it’ll show the following output:

3

2

1

The following defines the count\_down() function:

def count\_down(start):

""" Count down from a number """

print(start)

If you call the count\_down() function now:

count\_down(3)

…it’ll show only the number 3.

To show the numbers 3, 2, and 1, you need to:

* First, call the count\_down(3) to show the number 3.
* Second, call the count\_down(2) to show the number 2.
* Finally, call the count\_down(1) to show the number 1.

In order to do so, inside the count\_down() function, you’ll need to define a logic to call the function count\_down() with argument 2, and 1.

To do it, you need to make the count\_down() function recursive.

The following defines a recursive count\_down() function and calls it by passing the number 3:

def count\_down(start):

""" Count down from a number """

print(start)

count\_down(start-1)

count\_down(3)

If you execute the program, you’ll see the following error:

RecursionError: maximum recursion depth exceeded while calling a Python object

The reason is that the count\_down() calls itself indefinitely until the system stops it.

Since you need to stop counting down the number reaches zero. To do so, you add a condition like this:

def count\_down(start):

""" Count down from a number """

print(start)

*# call the count\_down if the next*

*# number is greater than 0*

next = start - 1

if next > 0:

count\_down(next)

count\_down(3)

Output:

3

2

1

In this example, the count\_down() function only calls itself when the next number is greater than zero. In other words, if the next number is zero, it stops calling itself.

## **2) Using a recursive function to calculate the sum of a sequence**

Suppose that you need to calculate a sum of a sequence e.g., from 1 to 100. A simple way to do this is to use a [for loop with the range() function](https://www.pythontutorial.net/python-basics/python-for-range/):

def sum(n):

total = 0

for index in range(n+1):

total += index

return total

result = sum(100)

print(result)

Output:

5050

To apply the recursion technique, you can calculate the sum of the sequence from 1 to n as follows:

* sum(n) = n + sum(n-1)
* sum(n-1) = n-1 + sum(n-2)
* …
* sum(0) = 0

The sum() function keeps calling itself as long as its argument is greater than zero.

The following defines the recursive version of the sum() function:

def sum(n):

if n > 0:

return n + sum(n-1)

return 0

result = sum(100)

print(result)

As you can see, the recursive function is much shorter and more readable.

If you use the [ternary operator](https://www.pythontutorial.net/python-basics/python-ternary-operator/), the sum() will be even more concise:

def sum(n):

return n + sum(n-1) if n > 0 else 0

result = sum(100)

print(result)

## **Summary**

* A recursive function is a function that calls itself until it doesn’t.
* And a recursive function always has a condition that stops calling itself.

# **Python Lambda Expressions**

Sometimes, you need to write a simple [function](https://www.pythontutorial.net/python-basics/python-functions/) that contains one expression. However, you need to use this function once. And it’ll unnecessary to define that function with the def keyword.

That’s where the Python lambda expressions come into play.

## **What are Python lambda expressions**

Python lambda expressions allow you to define anonymous functions.

Anonymous functions are functions without names. The anonymous functions are useful when you need to use them once.

A lambda expression typically contains one or more arguments, but it can have **only one expression**.

The following shows the lambda expression syntax:

lambda parameters: expression

It’s equivalent to the following function without the "anonymous" name:

def anonymous(parameters):

return expression

## **Python lambda expression examples**

In Python, you can pass a function to another function or return a function from another function.

### **1) Functions that accept a function example**

The following defines a function called get\_full\_name() that format the full name from the first name and last name:

def get\_full\_name(first\_name, last\_name, formatter):

return formatter(first\_name, last\_name)

The get\_full\_name() function accepts three arguments:

* First name (first\_name)
* Last name (last\_name)
* A function that will format the full name (formatter). In turn, the formatter function accepts two arguments first name and last name.

The following defines two functions that return a full name from the first name and last name in different formats:

def first\_last(first\_name, last\_name):

return f"{first\_name} {last\_name}"

def last\_first(first\_name, last\_name):

return f"{last\_name}, {first\_name}"

And this shows you how to call the get\_full\_name() function by passing the first name, last name, and first\_last / last\_first functions:

full\_name = get\_full\_name('John', 'Doe', first\_last)

print(full\_name) *# John Doe*

full\_name = get\_full\_name('John', 'Doe', last\_first)

print(full\_name) *# Doe, John*

Output:

John Doe

Doe, John

Instead of defining the first\_last and last\_first functions, you can use lambda expressions.

For example, you can express the first\_last function using the following lambda expression:

lambda first\_name,last\_name: f"{first\_name} {last\_name}"

This lambda expression accepts two arguments and concatenates them into a formatted string in the order first\_name, space, and last\_name.

And the following converts the last\_first function using a lambda expression that returns the full name in the format: last name, space, and first name:

lambda first\_name, last\_name: f"{last\_name} {first\_name}";

By using lambda expressions, you can call the get\_full\_name() function as follows:

def get\_full\_name(first\_name, last\_name, formatter):

return formatter(first\_name, last\_name)

full\_name = get\_full\_name(

'John',

'Doe',

lambda first\_name, last\_name: f"{first\_name} {last\_name}"

)

print(full\_name)

full\_name = get\_full\_name(

'John',

'Doe',

lambda first\_name, last\_name: f"{last\_name} {first\_name}"

)

print(full\_name)

Output:

John Doe

Doe, John

### **2) Functions that return a function example**

The following times() function returns a function which is a lambda expression:

def times(n):

return lambda x: x \* n

And this example shows how to call the times() function:

double = times(2)

Since the times() function returns a function, the double is also a function. To call it, you place parentheses like this:

result = double(2)

print(result)

result = double(3)

print(result)

Output:

4

6

The following shows another example of using the times() function:

triple = times(3)

print(triple(2)) *# 6*

print(triple(3)) *# 9*

## **Python lambda in a loop**

See the following example:

callables = []

for i in (1, 2, 3):

callables.append(lambda: i)

for f in callables:

print(f())

How it works.

* First, define a list with the name callables.
* Second, iterate from 1 to 3, create a new lambda expression in each iteration, and add it to the callables list.
* Third, loop over the callables and call each function.

The expected output will be:

1

2

3

However, the program shows the following output:

3

3

3

The problem is that all the there lambda expressions reference the i variable, not the current value of i. When you call the lambda expressions, the value of the variable i is 3.

To fix this, you need to bind the i variable to each lambda expression at the time the lambda expression is created. One way to do it is to use the [default argument](https://www.pythontutorial.net/python-basics/python-default-parameters/):

callables = []

for i in (1, 2, 3):

callables.append(lambda a=i: a)

for f in callables:

print(f())

In this example, the value of a is evaluated at the time the lambda expression is created. Therefore, the program returns the expected output.

## **Summary**

* Use Python lambda expressions to create anonymous functions, which are functions without names.
* A lambda expression accepts one or more arguments, contains an expression, and returns the result of that expression.
* Use lambda expressions to pass anonymous functions to a function and return a function from another function.

# **Python Function Docstrings**

**Summary**: in this tutorial, you’ll learn about how to use docstrings to add documentation to a function.

## **Introduction to the help() function**

Python provides a built-in function called help() that allows you to show the documentation of a [function](https://www.pythontutorial.net/python-basics/python-functions/).

The following example shows the documentation of the print() function:

help(print)

Output:

print(...)

print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.

Optional keyword arguments:

file: a file-like object (stream); defaults to the current sys.stdout.

sep: string inserted between values, default a space.

end: string appended after the last value, default a newline.

flush: whether to forcibly flush the stream.

)

Note that you can use the help() function to show the documentation of [modules](https://www.pythontutorial.net/python-basics/python-module/), [classes](https://www.pythontutorial.net/python-oop/python-class/), [functions](https://www.pythontutorial.net/python-basics/python-functions/), and keywords. This tutorial focuses on function documentation only.

## **Using docstrings to document functions**

To document your functions, you can use docstrings. The [PEP 257](https://www.python.org/dev/peps/pep-0257/) provides the docstring conventions.

When the first line in the function body is a string, Python will interpret it as a docstring. For example:

def add(a, b):

"Return the sum of two arguments"

return a + b

And you can use the help() function to find the documentation of the add() function:

help(add)

Output:

add(a, b)

Return the sum of two arguments

Typically, you use multi-line docstrings:

def add(a, b):

""" Add two arguments

Arguments:

a: an integer

b: an integer

Returns:

The sum of the two arguments

"""

return a + b

Output:

add(a, b)

Add the two arguments

Arguments:

a: an integer

b: an integer

Returns:

The sum of the two arguments Code language: Shell Session (shell)

Python stores the docstrings in the \_\_doc\_\_ property of the function.

The following example shows how to access the \_\_doc\_\_ property of the add() function:

add.\_\_doc\_\_

## **Summary**

* Use the help() function to get the documentation of a function.
* Place a string, either single-line or multi-line strings, as the first line in the function to add documentation to it.