Python Function Arguments

programiz.com/python-programming/function-argument

Join our newsletter for the latest updates.

In Python, you can define a function that takes variable number of arguments. In this article, you will learn to define such functions using default, keyword and arbitrary arguments.

Arguments

In the user-defined function topic, we learned about defining a function and calling it. Otherwise, the function call will result in an error. Here is an example.

```
def greet(name, msg):
    """This function greets to
    the person with the provided message"""
    print("Hello", name + ', ' + msg)

greet("Monica", "Good morning!")
```

Output

```
Hello Monica, Good morning!
```

Here, the function `greet()` has two parameters.

Since we have called this function with two arguments, it runs smoothly and we do not get any error.

If we call it with a different number of arguments, the interpreter will show an error message. Below is a call to this function with one and no arguments along with their respective error messages.

```
>>> greet("Monica")    # only one argument
TypeError: greet() missing 1 required positional argument: 'msg'

>>> greet()    # no arguments
TypeError: greet() missing 2 required positional arguments: 'name' and 'msg'
```

Variable Function Arguments

Up until now, functions had a fixed number of arguments. In Python, there are other ways to define a function that can take variable number of arguments.

Three different forms of this type are described below.

Python Default Arguments

Function arguments can have default values in Python.

We can provide a default value to an argument by using the assignment operator (=). Here is an example.

```
def greet(name, msg="Good morning!"):
    """
    This function greets to
    the person with the
    provided message.

    If the message is not provided,
    it defaults to "Good
    morning!"
    """

    print("Hello", name + ', ' + msg)


greet("Kate")
greet("Bruce", "How do you do?")
```

Output

```
Hello Kate, Good morning!
Hello Bruce, How do you do?
```

In this function, the parameter `name` does not have a default value and is required (mandatory) during a call.

On the other hand, the parameter `msg` has a default value of `"Good morning!"`. So, it is optional during a call. If a value is provided, it will overwrite the default value.

Any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.

This means to say, non-default arguments cannot follow default arguments. For example, if we had defined the function header above as:

```
def greet(msg = "Good morning!", name):
```

We would get an error as:

```
SyntaxError: non-default argument follows default argument
```

Python Keyword Arguments

When we call a function with some values, these values get assigned to the arguments according to their position.

For example, in the above function `greet()`, when we called it as `greet("Bruce", "How do you do?")`, the value `"Bruce"` gets assigned to the argument *name* and similarly `"How do you do?"` to *msg*.

Python allows functions to be called using keyword arguments. When we call functions in this way, the order (position) of the arguments can be changed. Following calls to the above function are all valid and produce the same result.

```
# 2 keyword arguments
greet(name = "Bruce",msg = "How do you do?")

# 2 keyword arguments (out of order)
greet(msg = "How do you do?",name = "Bruce")

1 positional, 1 keyword argument
greet("Bruce", msg = "How do you do?")
```

As we can see, we can mix positional arguments with keyword arguments during a function call. But we must keep in mind that keyword arguments must follow positional arguments.

Having a positional argument after keyword arguments will result in errors. For example, the function call as follows:

```
greet(name="Bruce","How do you do?")
```

Will result in an error:

```
SyntaxError: non-keyword arg after keyword arg
```

Python Arbitrary Arguments

Sometimes, we do not know in advance the number of arguments that will be passed into a function. Python allows us to handle this kind of situation through function calls with an arbitrary number of arguments.

In the function definition, we use an asterisk (*) before the parameter name to denote this kind of argument. Here is an example.

```
def greet(*names):
    """This function greets all
    the person in the names tuple."""

    # names is a tuple with arguments
    for name in names:
        print("Hello", name)


greet("Monica", "Luke", "Steve", "John")
```

Output

```
Hello Monica
Hello Luke
Hello Steve
Hello John
```

Here, we have called the function with multiple arguments. These arguments get wrapped up into a tuple before being passed into the function. Inside the function, we use a `for` loop to retrieve all the arguments back.