Functions

Functions are the first step to code reuse. They allow you to define a reusable block of code that can be used repeatedly in a program.

Python provides several built-in functions such as print(), len() or type(), but you can also define your own functions to use within your programs.

Syntax

The basic syntax for a Python function definition is:

Def function_name(arguments):

Statement

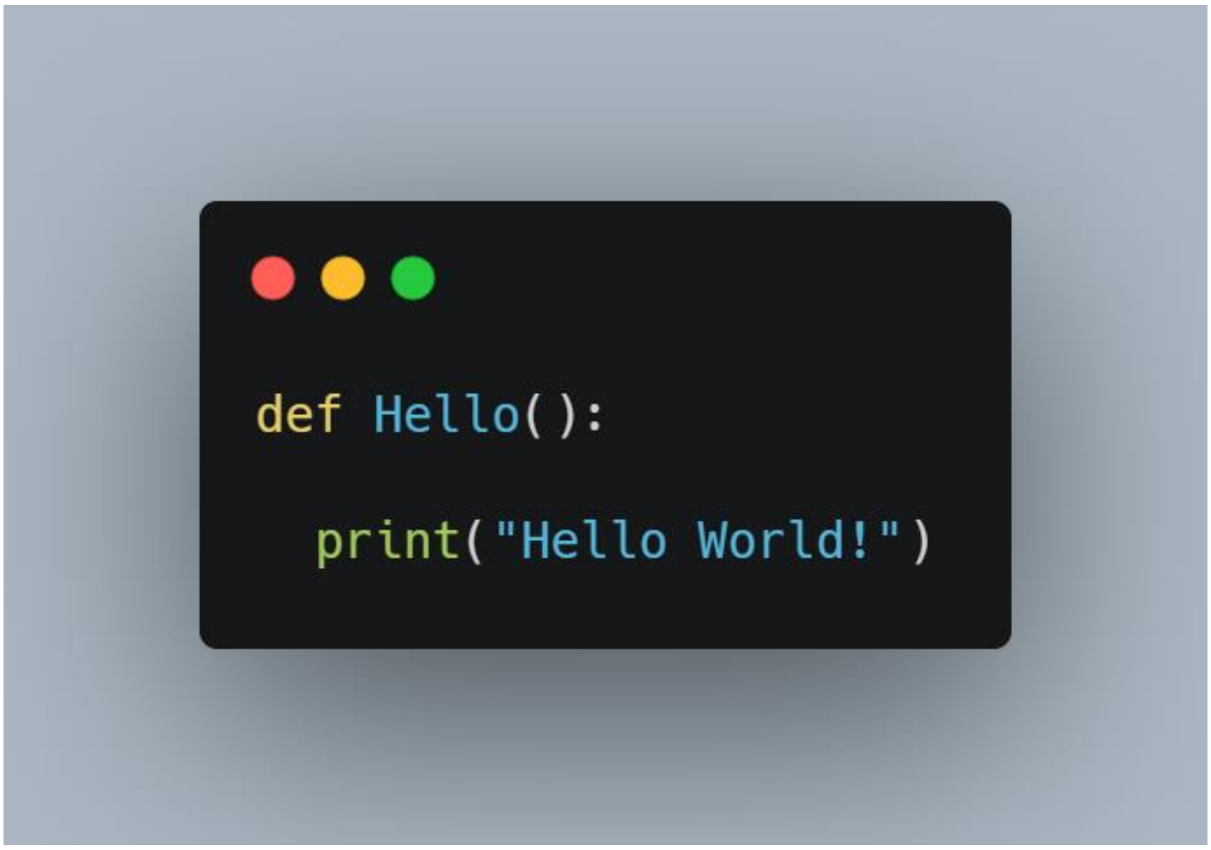
Return value

Function_name is the name of the function you wish to define, it has the same naming convention as a variable.

Statement is the function's body, it is executed whenever the function is called.

Return value ends the function call and sends data back to the program.

To define a Python function, use the def keyword. Here's the simplest possible function that prints 'Hello, World!' on the screen.



```python
def Hello():
  print("Hello World!")
```

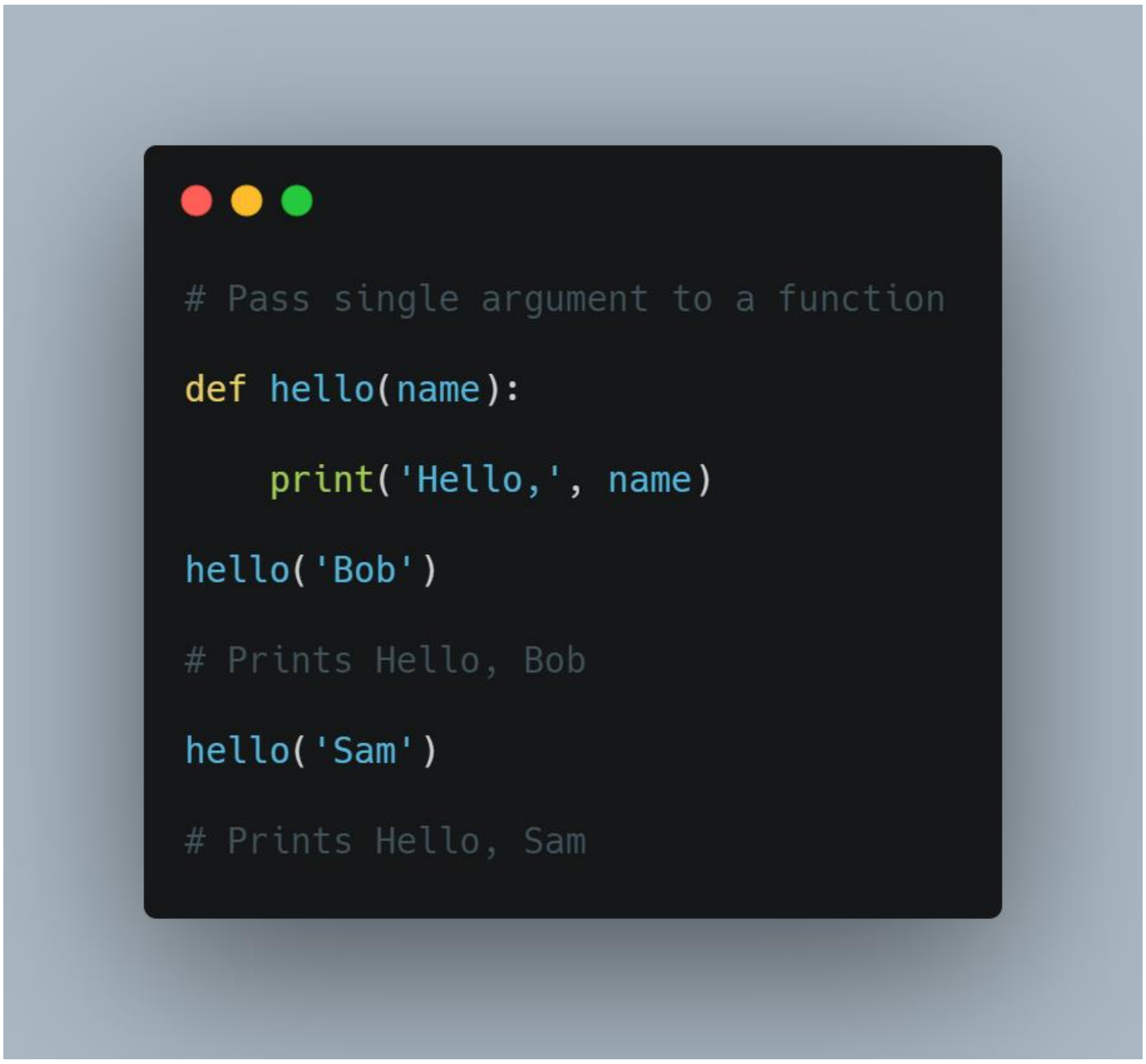
The def statement only creates a function but does not call it. After the def has run, you can call (run) the function by adding parentheses after the function's name.

Hello()

Pass Arguments

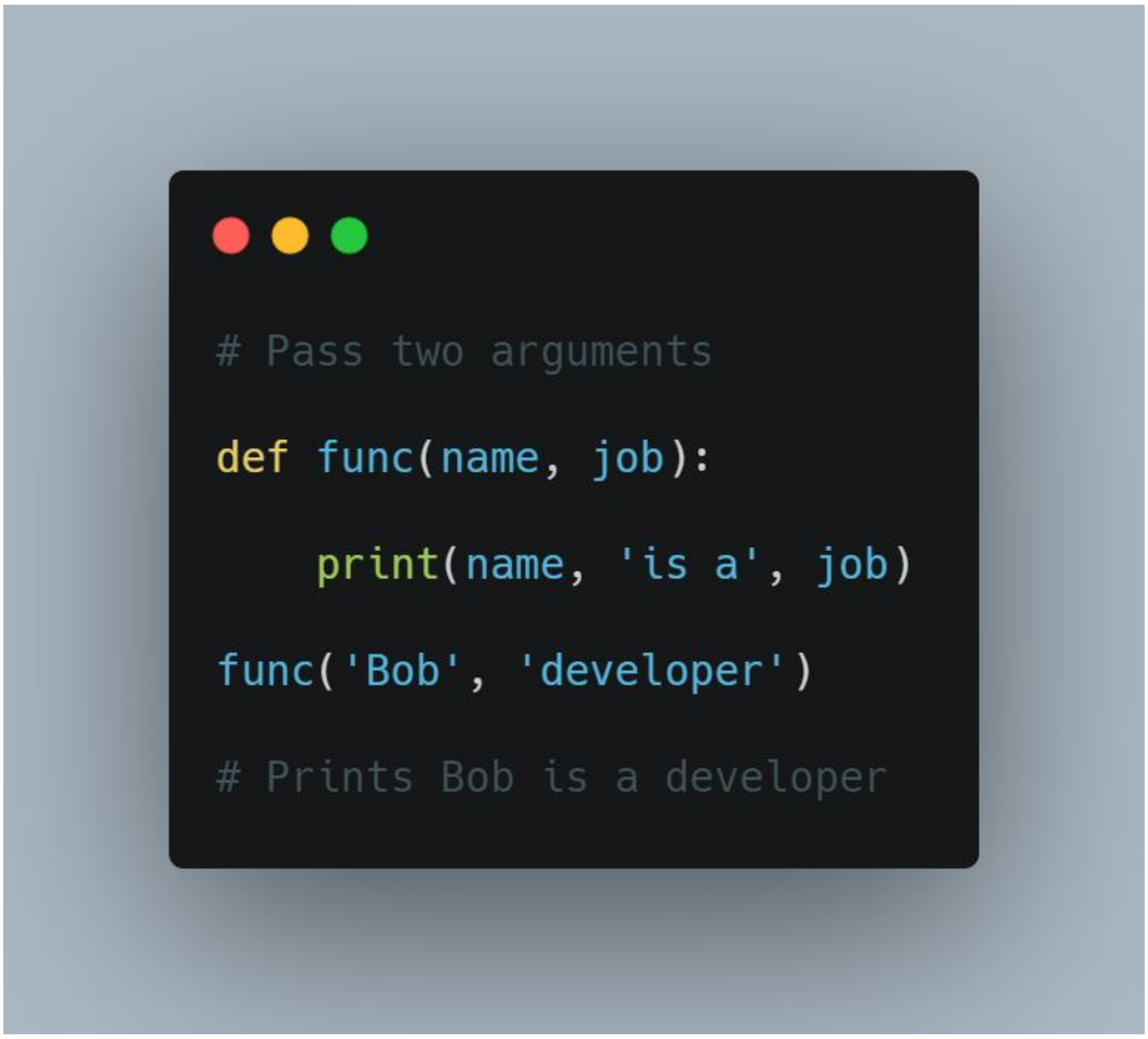
You can send information to a function by passing values, known as arguments. Arguments are declared after the function name in parentheses.

When you call a function with arguments, the values of those arguments are copied to their corresponding parameters inside the function.



```python
# Pass single argument to a function

def hello(name):

    print('Hello,', name)

hello('Bob')

# Prints Hello, Bob

hello('Sam')

# Prints Hello, Sam
```

You can send as many arguments as you like, separated by commas ,.



```python
# Pass two arguments

def func(name, job):

    print(name, 'is a', job)

func('Bob', 'developer')

# Prints Bob is a developer
```

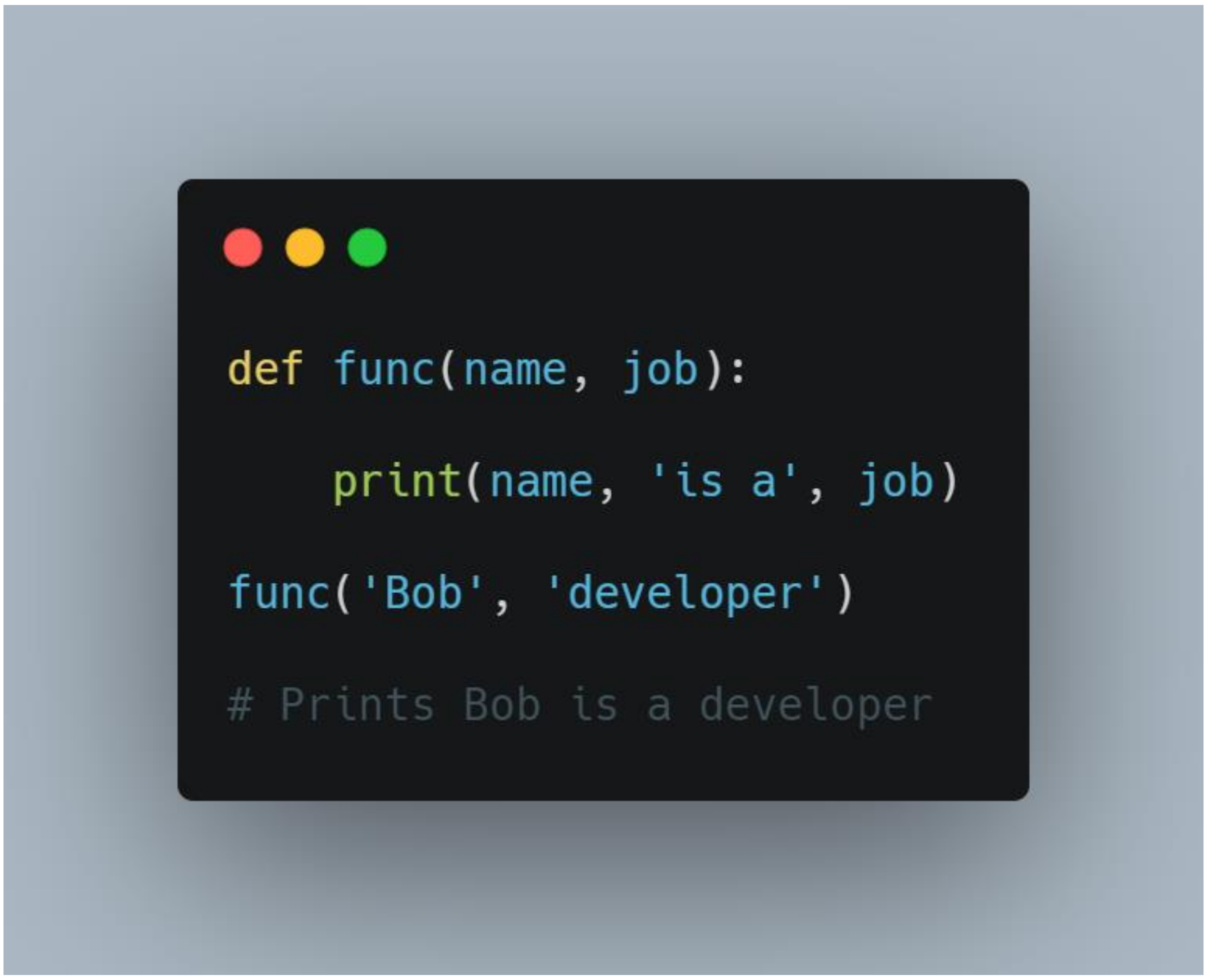
Types of Arguments

Python handles function arguments in a very flexible manner, compared to other languages. It supports multiple types of arguments in the function definition. Here's the list:

1. Positional Arguments
2. Keyword Arguments
3. Default Arguments
4. Variable Length Positional Arguments (*args)
5. Variable Length Keyword Arguments (**kwargs)

Positional Arguments

The most common are positional arguments, whose values are copied to their corresponding parameters in order.



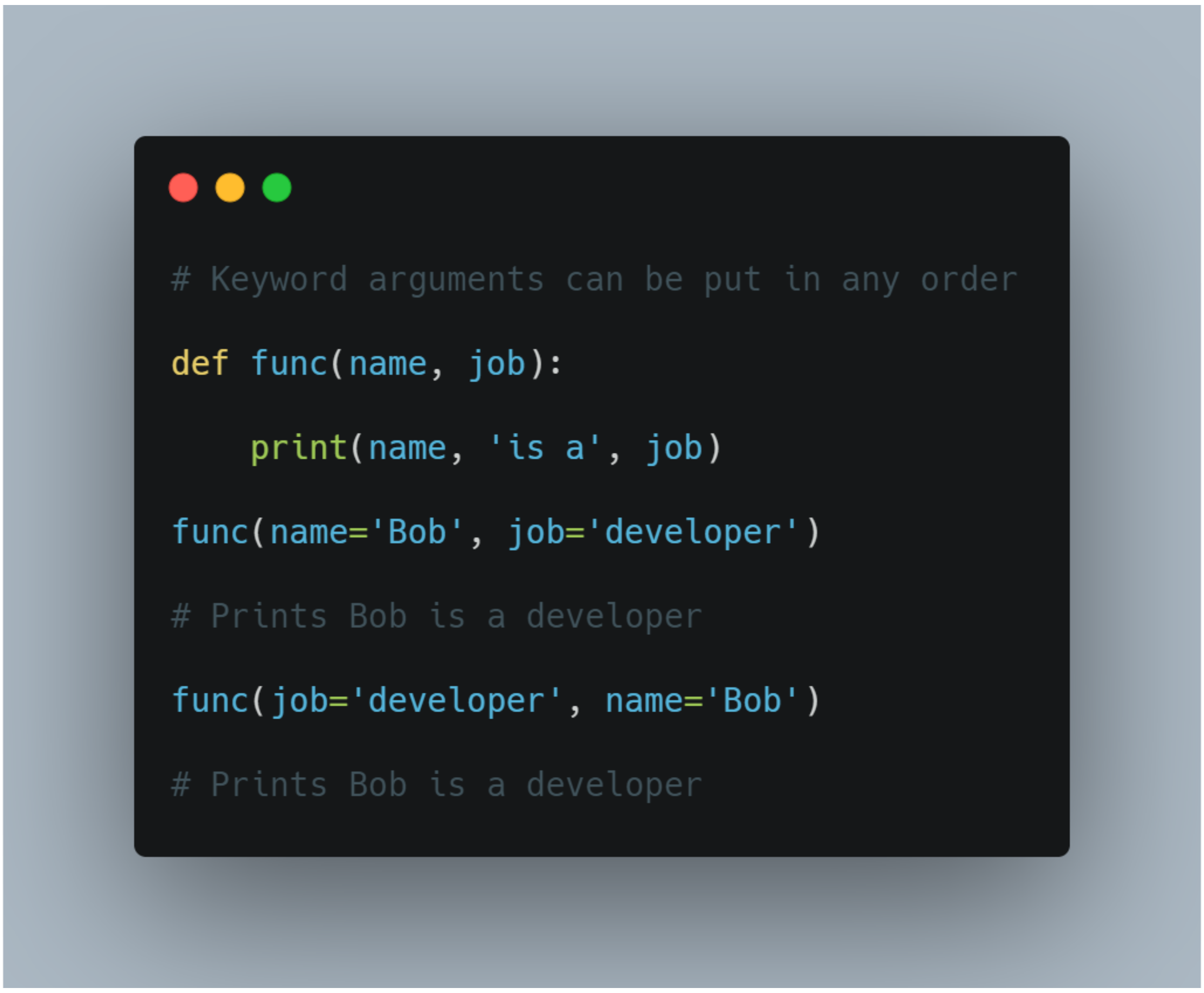
```python
def func(name, job):

    print(name, 'is a', job)

func('Bob', 'developer')

# Prints Bob is a developer
```

The only downside of positional arguments is that you need to pass arguments in the order in which they are defined.

Keyword Arguments

To avoid positional argument confusion, you can pass arguments using the names of their corresponding parameters.

In this case, the order of the arguments no longer matters because arguments are matched by name, not by position.

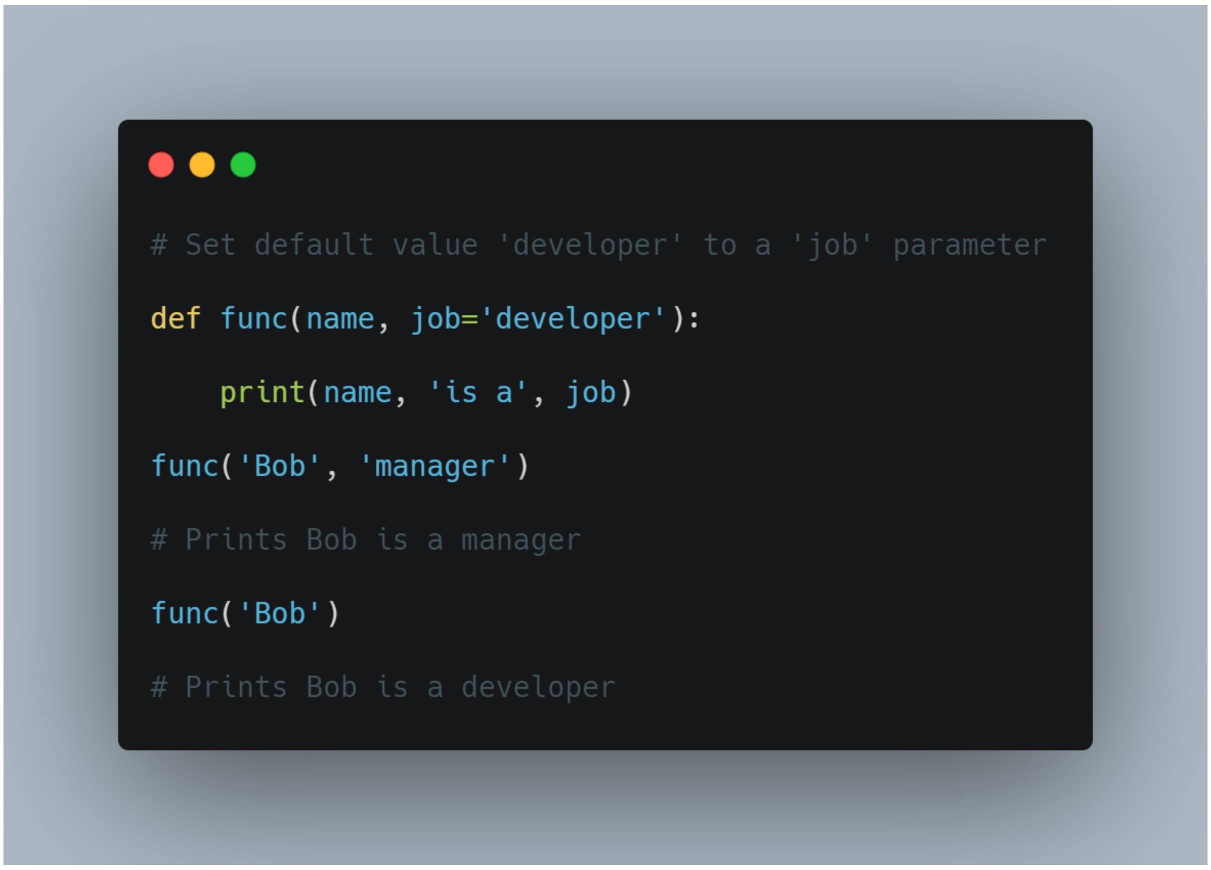


```python
# Keyword arguments can be put in any order

def func(name, job):

    print(name, 'is a', job)

func(name='Bob', job='developer')

# Prints Bob is a developer

func(job='developer', name='Bob')

# Prints Bob is a developer
```

Default Arguments

You can specify default values for arguments when defining a function. The default value is used if the function is called without a corresponding argument.

In short, defaults allow you to make selected arguments optional.



```python
# Set default value 'developer' to a 'job' parameter
def func(name, job='developer'):
    print(name, 'is a', job)
func('Bob', 'manager')
# Prints Bob is a manager
func('Bob')
# Prints Bob is a developer
```

Variable Length Arguments (*args and **kwargs)

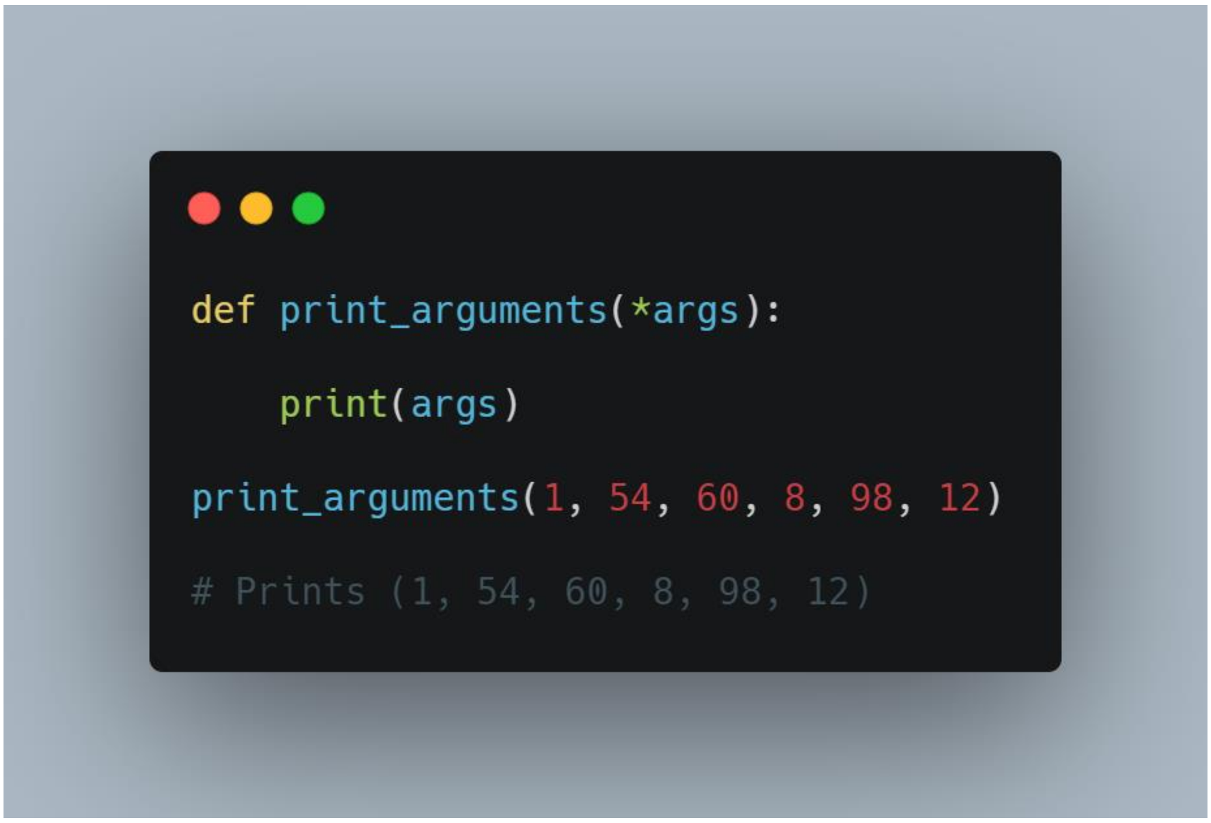
Variable length arguments are useful when you want to create functions that take an unlimited number of arguments. Unlimited in the sense that you do not know beforehand how many arguments can be passed to your function by the user.

This feature is often referred to as var-args.

*args

When you prefix a parameter with an asterisk * , it collects all the unmatched positional arguments into a tuple. Due to the fact that it is a normal tuple object, you can perform any operation that a tuple supports, like indexing, iteration etc.

Following function prints all the arguments passed to the function as a tuple.

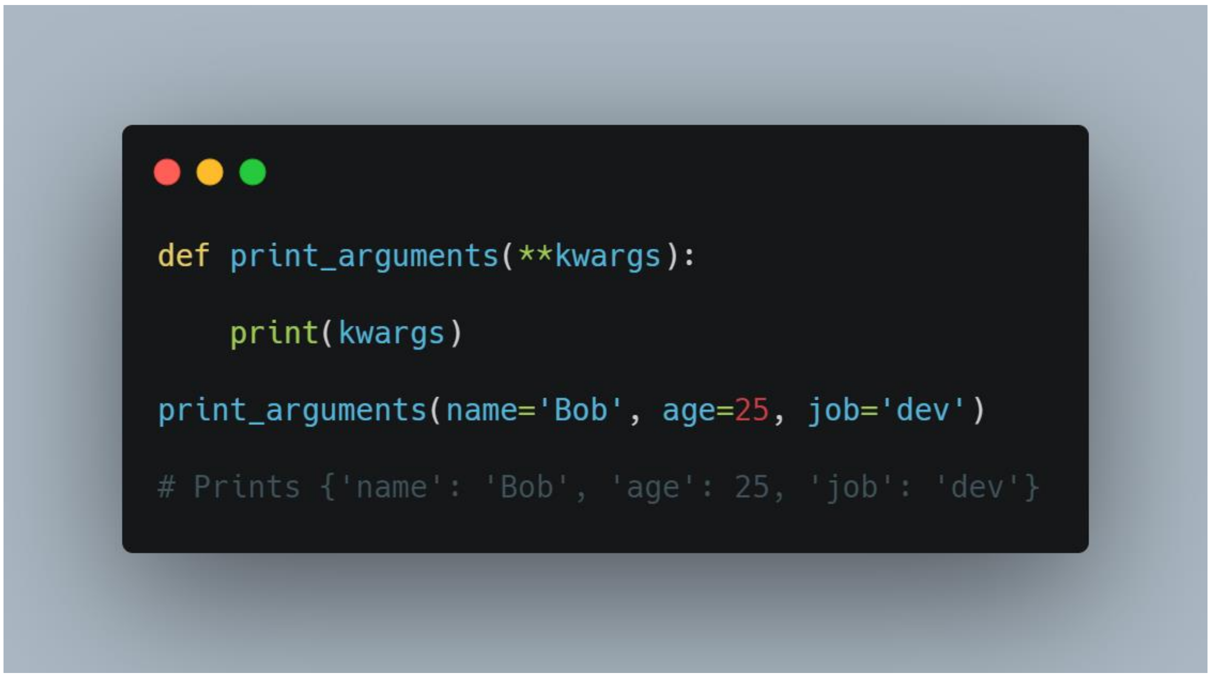


```python
def print_arguments(*args):
    print(args)
print_arguments(1, 54, 60, 8, 98, 12)
# Prints (1, 54, 60, 8, 98, 12)
```

You don’t need to call this keyword parameter args, but it is standard practice.

**kwargs

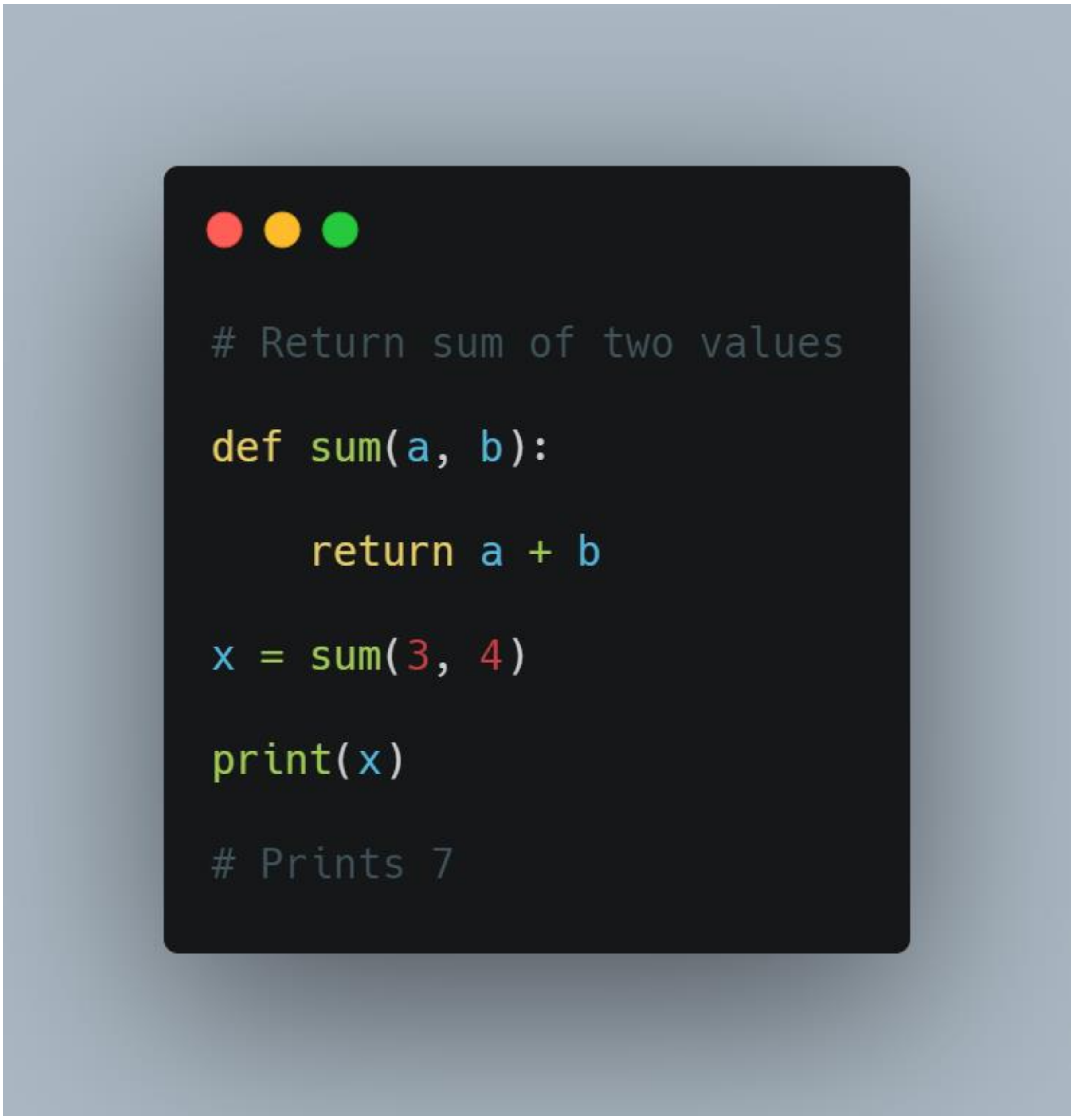
The ** syntax is similar, but it only works for keyword arguments. It collects them into a new dictionary, where the argument names are the keys, and their values are the corresponding dictionary values.



```python
def print_arguments(**kwargs):
    print(kwargs)
print_arguments(name='Bob', age=25, job='dev')
# Prints {'name': 'Bob', 'age': 25, 'job': 'dev'}
```

Return Value

To return a value from a function, simply use a return statement. Once a return statement is executed, nothing else in the function body is executed.

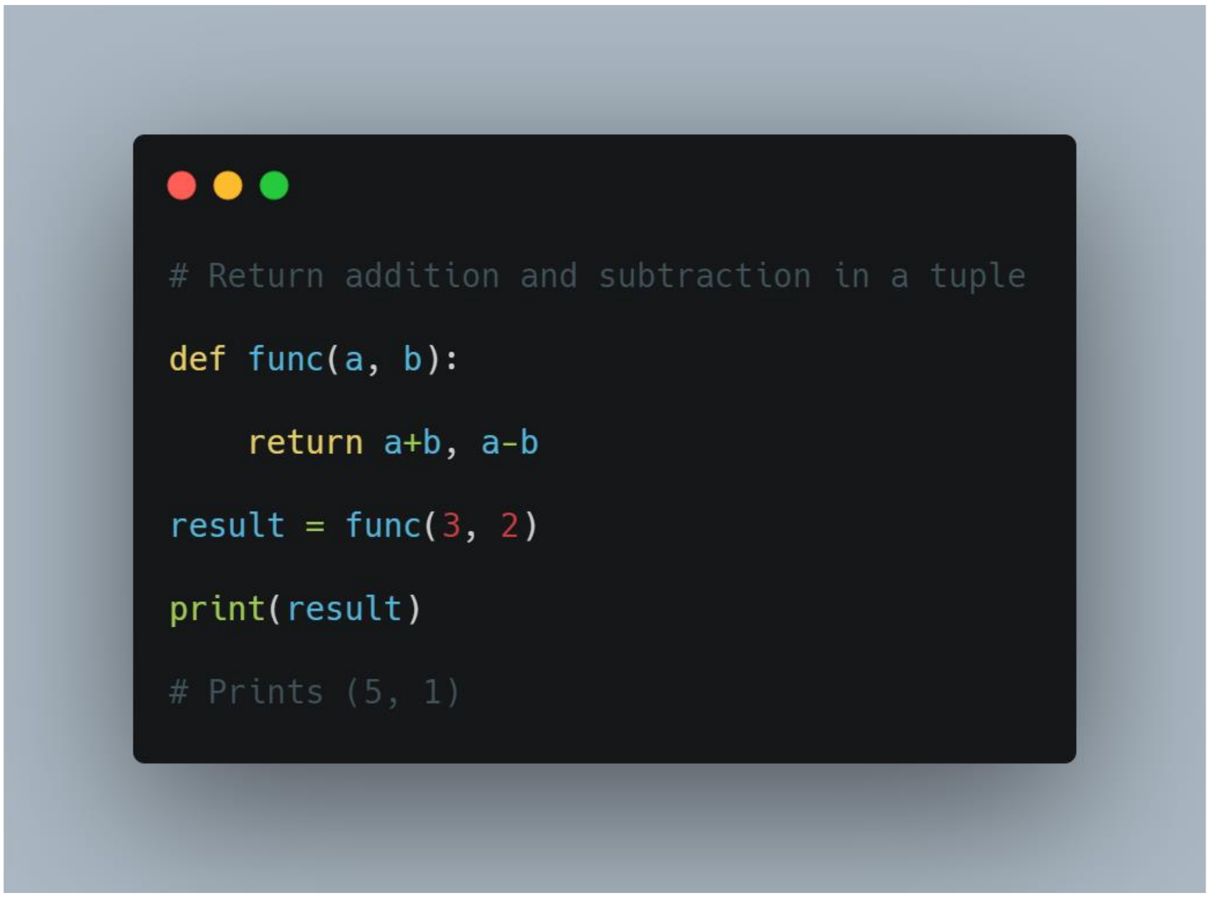


```python
# Return sum of two values

def sum(a, b):

    return a + b

x = sum(3, 4)

print(x)

# Prints 7
```

Remember! a python function always returns a value. So, if you do not include any return statement, it automatically returns None.

Return Multiple Values

Python has the ability to return multiple values, something missing from many other languages. You can do this by separating return values with a comma.



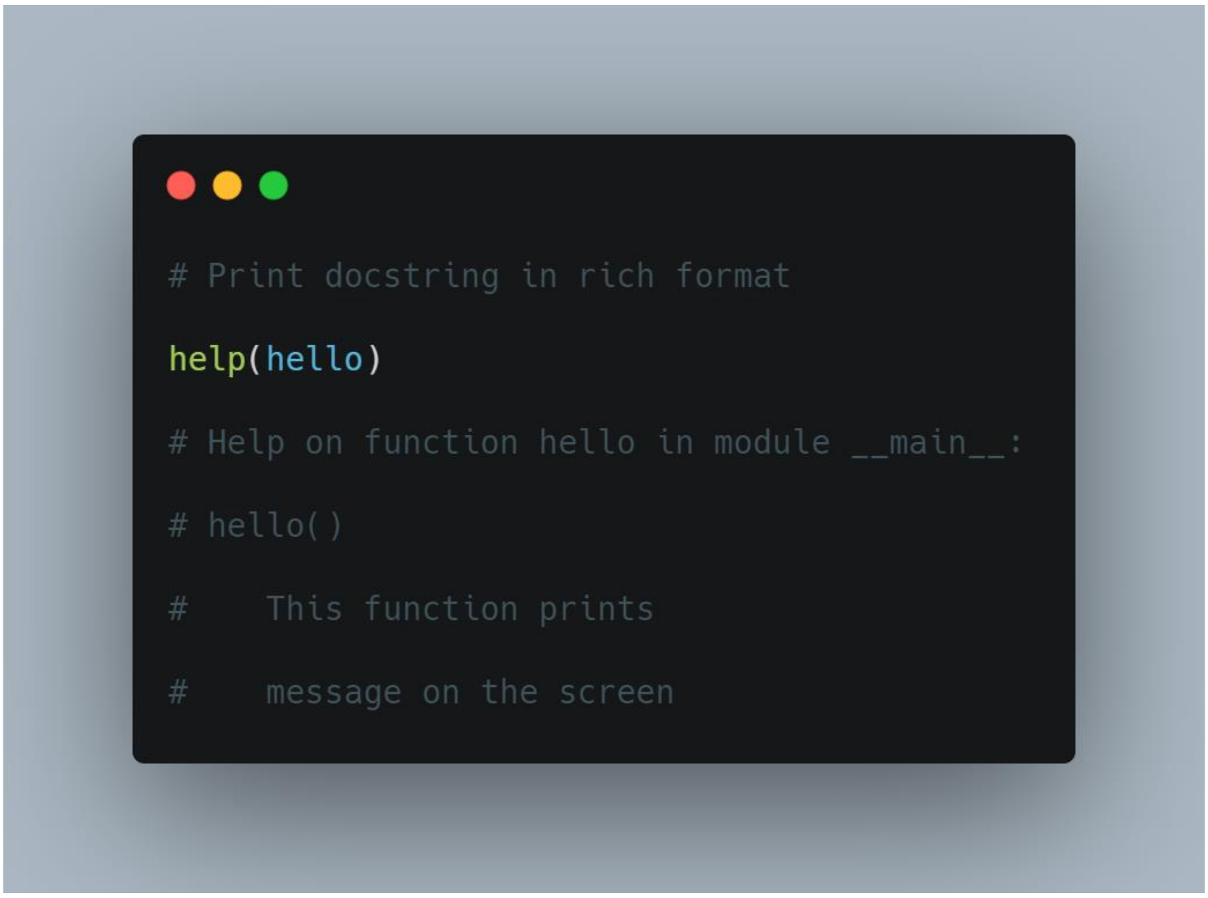
```python
# Return addition and subtraction in a tuple

def func(a, b):

    return a+b, a-b

result = func(3, 2)

print(result)

# Prints (5, 1)
```

Docstring

You can attach documentation to a function definition by including a string literal just after the function header. Docstrings are usually triple quoted to allow for multi-line descriptions.

```
def hello():

   """This function prints

      message on the screen"""

   print('Hello, World!')
```

To print a function's docstring, use the Python help() function and pass the function's name.



```
# Print docstring in rich format

help(hello)

# Help on function hello in module __main__:

# hello()

#    This function prints

#    message on the screen
```