**Functions**

**Functions**

A function is a reusable block of code that runs when you “call” it. Think of functions as a variable that stores functionality in the form of an executable block of code instead of storing data; functions.

Functions are named blocks of code that are designed to do one specific job.

When you want to perform a particular task that you’ve defined in a function, you call the name of the function responsible for it.

If you need to perform that task multiple times throughout your program, you don’t need to type all the code for the same task again and again; you call the function dedicated to handling that task, and the call tells Python to run the code inside the function.

Functions allow you to break your programs into small parts, each of which does one specific job.

Using functions, you’ll be able to write more efficient code that’s easier to troubleshoot and maintain, and that can be reused in many different programs.

**Syntax**

**Example of a simple function that says hello –**

def say\_hello():

"""A function that says hello"""

print("Hello!")

say\_hello()

**Definition –**

The first line uses the keyword def to inform Python that you’re defining a function.

This is the function definition, which tells Python the *name* of the function.

The parentheses hold some information about parameters; we’ll see this very soon. In this case, the name of the function is say\_hello(), and it needs no parameters, so its parentheses are empty (Even so, the parentheses are required.)

Finally, the definition ends with a colon, and all the code included in the function needs to be *indented*.

**Code –**

Any indented lines that follow def say\_hello(): make up the function’s body.

The text on the second line is a comment called a *docstring*, which describes what the function does. *Docstrings* are enclosed in triple quotes, which Python looks for when it generates documentation for the functions in your programs.

The line print("Hello!") is the only line of actual code in the body of this function, so say\_hello() has just one job: print “Hello!”.

**1. Function call**

When you want to use this function, you *call* it.

A function *call* tells Python to execute the code in the function. To *call* a function, you write the function’s name, followed by any necessary parameters in parentheses, as shown on the last line. Because no information is needed here, calling our function is as simple as entering say\_hello(). As expected, it prints Hello!

**2. Passing information to a function**

Powerful feature of functions is *arguments*, or *parameters*.

There are values in the functions that can be changed each time we call it. For example, the say\_hello function should say hello to someone, like “Hello Rick!”, but because we want to be able to change this name (and print hello to “Morty” as well), we don’t hard-code the name in the function.

Instead of this, we declare a *parameter* username and we pass an *argument* for it.

def say\_hello(username):

print("Hello "+username)

say\_hello("Rick") # "Rick" is an argument

# output "Hello Rick"

say\_hello("Morty") # "Morty" is an argument

# output "Hello Morty"

* In the function **definition**, we add the **parameter** that will accept a value.

def say\_hello(username):

print("Hello "+username)

* In the function **call**, we pass the value (i.e. an argument).

say\_hello("Rick") # "Rick" is an argument

# output "Hello Rick"

Here the function will execute the same code but with a different value for username.

say\_hello("Rick") # "Rick" is an argument

# output "Hello Rick"

say\_hello("Morty") # "Morty" is an argument

# output "Hello Morty"

**Be careful!:** If the function expects one argument, you can pass only one argument; less or more will throw an error.

**Example of a function that accept more than one argument –**

def say\_hello(username, language):

if language == "EN":

print("Hello "+username)

elif language == "FR":

print("Bonjour "+username)

else:

print("This language is not supported: " + language)

say\_hello("Rick", "FR")

This function accepts two arguments:

* The first one, username, is used to store the value of the username that we need to greet.
* The second one, language, is used to store the language of the greeting.

When *calling* the function, we pass two arguments in the right order (first the username and then the language).

Arguments matched up this way (by order) are called *positional arguments.*

* **Careful:** You can get unexpected results if you mix up the order of the arguments in a function call when using positional arguments.

If you get weird results, check to make sure the order of the arguments in your function call matches the order of the parameters in the function’s definition.

**3. Keyword arguments**

A *keyword argument* is a name-value pair that you pass to a function. You directly associate the name and the value within the argument, so when you pass the argument to the function, there’s no confusion.

Keyword arguments free you from having to worry about correctly ordering your arguments in the function call, and they clarify the role of each value in the function call.

Let’s rewrite the say\_hello function call using *keyword arguments*.

def say\_hello(username, language):

if language == "EN":

print("Hello "+username)

elif language == "FR":

print("Bonjour "+username)

else:

print("This language is not supported: " + language)

say\_hello(username="Rick", language="FR")

* **Careful:** When you use keyword arguments, be sure to use the exact names of the parameters in the function’s definition

The function say\_hello hasn’t changed. But when we call the function, we explicitly tell Python which parameter each argument should be matched with.

When Python reads the function call, it knows to store the argument ‘Rick’ in the parameter username and the argument ‘FR’ in language.

The order of keyword arguments doesn’t matter because Python knows where each value should go. The following two function calls are equivalent:

say\_hello(username="Rick", language="FR")

say\_hello(language="FR", username="Rick")

You can use both *keyword* and *positional* arguments in the same function call, but you need to place every *positional* argument before the *keyword* ones.

For example, this will work:

say\_hello("Rick", language="FR")

But this won’t:

say\_hello(username="Rick", "FR")

**4. Default values**

When writing a function, you can define a default value for each parameter.

If an argument for a parameter is provided in the function call, Python uses the argument value. If not, it uses the parameter’s default value.

So, when you define a default value for a parameter, you can exclude the corresponding argument you’d usually write in the function call.

Using default values can simplify your function calls and clarify how your functions are typically used.

For example, if you notice that most of the calls to say\_hello() are being used to prompt greetings in English, you can set the default value of language to “EN”. Now anyone calling say\_hello in English can omit that information:

def say\_hello(username, language="EN"):

if language == "EN":

print("Hello "+username)

elif language == "FR":

print("Bonjour "+username)

else:

print("This language is not supported: " + language)

say\_hello("Rick")

# OR

say\_hello(username="Rick")

We changed the definition of say\_hello to include a default value, “EN”, for language.

When the function is called with no language specified, Python knows to use the value “EN” for this parameter.

To use a language other than english, you could use a function call like this:

say\_hello("Rick", "FR")

# OR

say\_hello(username="Rick", language="FR")

Because an explicit argument for language is provided, Python will ignore the parameter’s default value.

When you use default values, any parameter with a default value must be listed after all the parameters that don’t have default values. This allows Python to continue interpreting positional arguments correctly.

**5. Avoiding argument errors**

When you start to use functions, don’t be surprised if you encounter errors about unmatched arguments. Unmatched arguments occur when you provide fewer or more arguments than a function needs to do its work.

For example, here’s what happens if we try to call say\_hello() with no arguments:

Traceback (most recent call last):

File "app.py", line 6, in <module> say\_hello()

TypeError: say\_hello() missing 2 required positional arguments: 'username' and 'language'

The first line of the traceback tells us the location of the problem, allowing us to look back and see that something went wrong in our function call.

On the second one, the offending function call is written out for us to see.

Then on the third, the traceback tells us the call is missing two arguments and reports the names of the missing arguments.

Python is helpful in that it reads the function’s code for us and tells us the names of the arguments we need to provide. This is another motivation for giving your variables and functions descriptive names. If you do, Python’s error messages will be more beneficial to you and anyone else who might use your code.

If you provide too many arguments, you should get a similar traceback that can help you correctly match your function call to the function definition.

**6. Local and Global scope**

**Local scope –**

def number\_by\_three(name, day):

sentence = f'Hello {name}! Today is {day}.'

print(sentence)

print(day)

>> NameError: name 'day' is not defined

**Global scope –**

name = 'Avner'

def say\_hi():

print(name)

say\_hi()

>> 'Avner'

**7. Return values**

A function doesn’t always have to display its output directly. Instead, it can process some data and then return a value or set of values.

The value the function returns is called a return value.

The return statement takes a value from inside a function and sends it back to the line that called the function.

Return values allow you to move much of your program’s grunt work into functions, which can simplify the body of your program.

**Returning a simple value –**

Let’s look at a function that takes a first and last name, and returns a neatly formatted full name:

def get\_formatted\_name(first\_name, last\_name):

"""Return a full name, neatly formatted."""

full\_name = first\_name + ' ' + last\_name

return full\_name.title()

musician = get\_formatted\_name('jimi', 'hendrix')

print(musician)

# Jimi Hendrix

The definition of get\_formatted\_name() takes as parameters a first and last name.

The function combines these two names, adds a space between them, and stores the result in full\_name.

The value of full\_name is converted to title case and then returned to the calling line.

When you call a function that returns a value, you need to provide a variable where the return value can be stored.

In this case, the returned value is stored in the variable musician. The output shows a neatly formatted name made up of the parts of a person’s name:

**Example**:

def divide\_by\_three(number):

return number / 3

first\_number = 12

first\_number\_computed = divide\_by\_three(first\_number)

print(first\_number\_computed)

>> 4.0

second\_number = 27

second\_number\_computed = divide\_by\_three(second\_number)

print(second\_number\_computed)

>> 9.0

**Returning more than one value with a tuple**

We already saw that a tuple can be unpacked:

my\_tuple = ("jimi", "hendrix")

first\_name, last\_name = my\_tuple

print("First name is: " + first\_name)

print("Last name is: " + last\_name)

First name is jimi

Last name is hendrix

If so, we can return a tuple at the end of the function, and unpack it when we *call* it:

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

return (first\_name.title(), last\_name.title())

first, last = format\_name("RICk", "mORTY")

print(first)

print(last)

Rick

Morty

The function receives a first\_name and a last\_name, then formats them and returns them as a tuple.

On the function call, the tuple is unpacked into two variables, first and last.

**Exercise**

Write a function calculation() such that it can accept two variables and calculate the addition and subtraction of it. And also, it must return both addition and subtraction in a single return call.

For example:

def calculation(a, b):

# Your Code

res = calculation(40, 10)

print(res)

def calculation\_1(a,b):

    addition = a+b

    subtraction = a-b

    return addition, subtraction

res = calculation\_1(40,10)

print(res)

def calculation\_2 (a,b):

    return(a+b, a-b)

x,y = calculation\_2(40,10)

print(x)

**Passing list as function arguments**

Function arguments can be any Python data type; you can pass strings as well as a list.

For example, a function that greets a list of users:

def greet\_users(users): # users should be a list

for user in users: # Because it's a list,

we can loop through it

print("Hello " + user.title() + " !") # For each user, print

"hello" and then his name

usernames = ["steve", "stan", "debbie"]

greet\_users(usernames)

# Hello Steve!

# Hello Stan!

# Hello Debbie!

**Example with functions**

For example, let’s create three functions:

def my\_f1():

print("Hello")

def my\_f2():

print("Word")

def my\_f3():

print("This is Rick!")

If we refer to functions as variables, we can store them in a list:

list\_of\_functions = [my\_f1, my\_f2, my\_f3]

Be sure to remove the round brackets, else the function will be executed, and you will store the result.

And now that we have a list of variables that can be executed when adding round brackets let’s run them with a for loop

for function in list\_of\_functions:

function()

**Modifying a list in a function**

When you pass a list to a function, the function can modify the list. Any changes made to the list inside the function’s body are permanent (even outside the function), allowing you to work efficiently even when you’re dealing with large amounts of data.

For example, consider a company that creates 3D printed models of designs that users submit. Designs that need to be printed are stored in a list, and after being printed they’re moved to a separate list. The following code does this without using functions.

unprinted\_designs = ['iphone case', 'robot pendant', 'dodecahedron']

completed\_models = []

# Simulate printing each design until none are left.

# Move each design to completed\_models after printing.

while len(unprinted\_designs) != 0:

current\_design = unprinted\_designs.pop()

# Simulate creating a 3D print from the design.

print("Printing model: " + current\_design)

completed\_models.append(current\_design)

# Display all completed models.

print("\nThe following models have been printed:")

for completed\_model in completed\_models:

print(completed\_model)

Printing model: iPhone case

Printing model: robot pendant

Printing model: dodecahedron

The following models have been printed:

iPhone case

robot pendant

dodecahedron

This program starts with a list of designs that need to be printed and an empty list called completed\_models that each design will be moved to after it has been printed.

As long as designs remain in unprinted\_designs, the while loop simulates printing each design by removing a design from the end of the list, storing it in current\_design, and displaying a message the current plan is being printed.

It then adds the design to the list of completed models. When the loop is finished running, a list of the methods that have been printed is displayed.

We can reorganize this code by writing two functions, each of which does one specific job.

Most of the code won’t change; we’re just making it more efficient.

The first function will handle printing the designs, and the second will summarize the points that have been made:

def print\_models(unprinted\_designs, completed\_models):

"""

Simulate printing each design until none are left.

Move each design to completed\_models after printing.

"""

while unprinted\_designs:

current\_design = unprinted\_designs.pop()

# Simulate creating a 3D print from the design.

print("Printing model: " + current\_design)

completed\_models.append(current\_design)

def show\_completed\_models(completed\_models):

"""

Show all the models that were printed.

"""

print("\nThe following models have been printed:")

for completed\_model in completed\_models:

print(completed\_model)

unprinted\_designs = ['iphone case', 'robot pendant', 'dodecahedron']

completed\_models = []

print\_models(unprinted\_designs, completed\_models)

show\_completed\_models(completed\_models)

This program is easier to extend and maintain than the version without functions.

If we need to print more designs later on, we can call print\_models() again.

If we realize the printing code needs to be modified, we can change the code once, and our changes will take place everywhere the function is called.

This technique is more efficient than having to update code separately in several places in the program.

This example also demonstrates the idea that every function should have one specific job.

The first function prints each design, and the second displays the completed models.

This is more beneficial than using one function to do both jobs.

If you’re writing a function and notice the function is doing too many different tasks, try to split the code into two functions.

Remember that you can always call a function from another function, which can be helpful when splitting a complex task into a series of steps.

If you want to prevent a function from modifying a list, pass my\_list.copy() instead of my\_list in the arguments.