Functions in Python

Estimated time needed: **40** minutes

Objectives

After completing this lab you will be able to:

- Understand functions and variables
- Work with functions and variables

Functions in Python

Welcome! This notebook will teach you about the functions in the Python Programming Language. By the end of this lab, you'll know the basic concepts about function, variables, and how to use functions.

Table of Contents

Estimated time needed: **40 min**

Functions

A function is a reusable block of code which performs operations specified in the function. They let you break down tasks and allow you to reuse your code in different programs.

There are two types of functions :

- **Pre-defined functions**
- **User defined functions**

What is a Function?

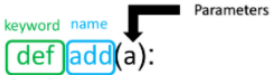
You can define functions to provide the required functionality. Here are simple rules to define a function in Python:

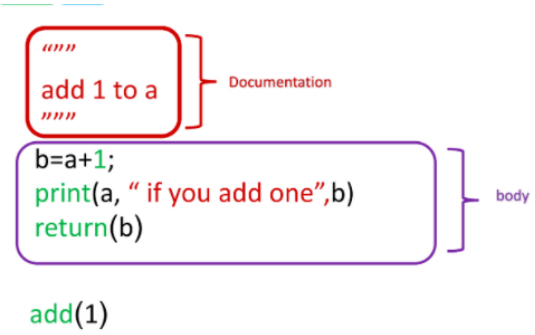
- Functions blocks begin `def` followed by the function `name` and parentheses `()`.
- There are input parameters or arguments that should be placed within these parentheses.
- You can also define parameters inside these parentheses.
- There is a body within every function that starts with a colon (`:`) and is indented.
- You can also place documentation before the body.
- The statement `return` exits a function, optionally passing back a value.

An example of a function that adds on to the parameter `a` prints and returns the output as `b`:

In [ ]:
```
# First function example: Add 1 to a and store as b

def add(a):
    b = a + 1
    print(a, "if you add one", b)
    return(b)
```

The figure below illustrates the terminology:





We can obtain help about a function :

In [ ]:
```
# Get a help on add function

help(add)
```

We can call the function:

In [ ]:
```
# Call the function add()

add(1)
```

If we call the function with a new input we get a new result:

In [ ]:
```
# Call the function add()

add(2)
```

We can create different functions. For example, we can create a function that multiplies two numbers. The numbers will be represented by the variables `a` and `b`:

In [2]:
```
# Define a function for multiple two numbers

def Mult(a, b):
    c = a * b
    return(c)
    print('This is not printed')

result = Mult(12,2)
print(result)
```

```
24
```

The same function can be used for different data types. For example, we can multiply two integers:

In [ ]:
```
# Use mult() multiply two integers

Mult(2, 3)
```

Note how the function terminates at the `return` statement, while passing back a value. This value can be further assigned to a different variable as desired.

The same function can be used for different data types. For example, we can multiply two integers:

Two Floats:

In [ ]:
```
# Use mult() multiply two floats

Mult(10.0, 3.14)
```

We can even replicate a string by multiplying with an integer:

In [3]:
```
# Use mult() multiply two different type values together

Mult(2, "Michael Jackson ")
```

Out[3]:
```
'Michael Jackson Michael Jackson '
```

Variables

The input to a function is called a formal parameter.

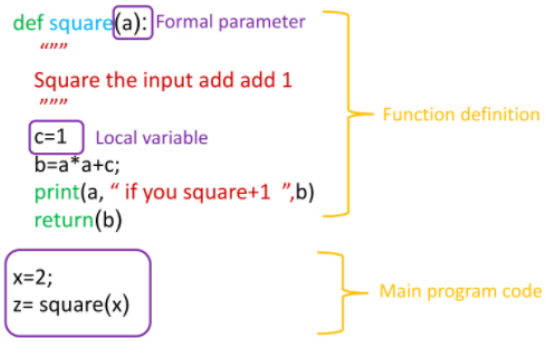
A variable that is declared inside a function is called a local variable. The parameter only exists within the function (i.e. the point where the function starts and stops).

A variable that is declared outside a function definition is a global variable, and its value is accessible and modifiable throughout the program. We will discuss more about global variables at the end of the lab.

In [ ]:
```
# Function Definition

def square(a):
```

```
    # Local variable b
    b = 1
    c = a * a + b
    print(a, "if you square + 1", c)
    return(c)
```

The labels are displayed in the figure:



We can call the function with an input of **3**:

In [ ]:
```
# Initializes Global variable

x = 3
# Makes function call and return function a y
y = square(x)
y
```

We can call the function with an input of **2** in a different manner:

In [ ]:
```
# Directly enter a number as parameter

square(2)
```

If there is no `return` statement, the function returns `None`. The following two functions are equivalent:

In [4]:
```
# Define functions, one with return value None and other without return value

def MJ():
    print('Michael Jackson')

def MJ1():
    print('Michael Jackson')
    return(None)
```

In [5]:
```
# See the output

MJ()
```

```
Michael Jackson
```

In [6]:
```
# See the output

MJ1()
```

```
Michael Jackson
```

Printing the function after a call reveals a **None** is the default return statement:

In [7]:
```
# See what functions returns are

print(MJ())
print(MJ1())
```

```
Michael Jackson
None
Michael Jackson
None
```

Create a function `con` that concatenates two strings using the addition operation:

In [8]:
```
# Define the function for combining strings

def con(a, b):
    return(a + b)
```

In [9]:
```
# Test on the con() function

con("This ", "is")
```

Out[9]:
```
'This is'
```

[Tip] How do I learn more about the pre-defined functions in Python?

We will be introducing a variety of pre-defined functions to you as you learn more about Python. There are just too many functions, so there's no way we

can teach them all in one sitting. But if you'd like to take a quick peek, here's a short reference card for some of the commonly-used pre-defined functions: Reference

Functions Make Things Simple

Consider the two lines of code in **Block 1** and **Block 2**: the procedure for each block is identical. The only thing that is different is the variable names and values.

Block 1:

In [ ]:
```
# a and b calculation block1

a1 = 4
b1 = 5
c1 = a1 + b1 + 2 * a1 * b1 - 1
if(c1 < 0):
    c1 = 0
else:
    c1 = 5
c1
```

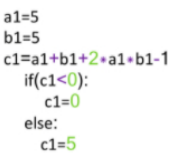
Block 2:

In [ ]:
```
# a and b calculation block2

a2 = 0
b2 = 0
c2 = a2 + b2 + 2 * a2 * b2 - 1
if(c2 < 0):
    c2 = 0
else:
    c2 = 5
c2
```

We can replace the lines of code with a function. A function combines many instructions into a single line of code. Once a function is defined, it can be used repeatedly. You can invoke the same function many times in your program. You can save your function and use it in another program or use someone else's function. The lines of code in code **Block 1** and code **Block 2** can be replaced by the following function:

In [ ]:
```
# Make a Function for the calculation above

def Equation(a,b):
    c = a + b + 2 * a * b - 1
    if(c < 0):
        c = 0
    else:
        c = 5
    return(c)
```

This function takes two inputs, a and b, then applies several operations to return c. We simply define the function, replace the instructions with the function, and input the new values of `a1`, `b1` and `a2`, `b2` as inputs. The entire process is demonstrated in the figure:



Code **Blocks 1** and **Block 2** can now be replaced with code **Block 3** and code **Block 4**.

Block 3:

In [ ]:
```
a1 = 4
b1 = 5
c1 = Equation(a1, b1)
c1
```

Block 4:

In [ ]:
```
a2 = 0
b2 = 0
c2 = Equation(a2, b2)
c2
```

Pre-defined functions

There are many pre-defined functions in Python, so let's start with the simple ones.

The `print()` function:

In [ ]:
```
# Build-in function print()

album_ratings = [10.0, 8.5, 9.5, 7.0, 7.0, 9.5, 9.0, 9.5]
print(album_ratings)
```

The `sum()` function adds all the elements in a list or tuple:

In [ ]:
```
# Use sum() to add every element in a list or tuple together

sum(album_ratings)
```

The `len()` function returns the length of a list or tuple:

In [ ]:
```
# Show the length of the list or tuple

len(album_ratings)
```

Using `if`/`else` Statements and Loops in Functions

The `return()` function is particularly useful if you have any IF statements in the function, when you want your output to be dependent on some condition:

In [ ]:
```
# Function example

def type_of_album(artist, album, year_released):

    print(artist, album, year_released)
    if year_released > 1980:
        return "Modern"
    else:
        return "Oldie"

x = type_of_album("Michael Jackson", "Thriller", 1980)
print(x)
```

We can use a loop in a function. For example, we can `print` out each element in a list:

In [ ]:
```
# Print the list using for loop

def PrintList(the_list):
    for element in the_list:
        print(element)
```

In [ ]:
```
# Implement the printlist function

PrintList(['1', 1, 'the man', "abc"])
```

Setting default argument values in your custom functions

You can set a default value for arguments in your function. For example, in the `isGoodRating()` function, what if we wanted to create a threshold for what we consider to be a good rating? Perhaps by default, we should have a default rating of 4:

In [ ]:
```
# Example for setting param with default value

def isGoodRating(rating=4):
    if(rating < 7):
        print("this album sucks it's rating is",rating)

    else:
        print("this album is good its rating is",rating)
```

In [ ]:
```
# Test the value with default value and with input

isGoodRating()
isGoodRating(10)
```

Global variables

So far, we've been creating variables within functions, but we have not discussed variables outside the function. These are called global variables. Let's try to see what `printer1` returns:

In [5]:
```
# Example of global variable

artist = "Michael Jackson"
def printer1(artist):
    internal_var1 = artist
    print(artist, "is an artist")

printer1(artist)
# try runningthe following code
#printer1(internal_var1)
```

```
Michael Jackson is an artist
```

We got a Name Error: `name 'internal_var' is not defined`. Why?

It's because all the variables we create in the function is a **local variable**, meaning that the variable assignment does not persist outside the function.

But there is a way to create **global variables** from within a function as follows:

In [ ]:
```
artist = "Michael Jackson"

def printer(artist):
    global internal_var
    internal_var= "Whitney Houston"
    print(artist,"is an artist")

printer(artist)
printer(internal_var)
```

Scope of a Variable

The scope of a variable is the part of that program where that variable is accessible. Variables that are declared outside of all function definitions, such as the `myFavouriteBand` variable in the code shown here, are accessible from anywhere within the program. As a result, such variables are said to have global scope, and are known as global variables. `myFavouriteBand` is a global variable, so it is accessible from within the `getBandRating` function, and we can use it to determine a band's rating. We can also use it outside of the function, such as when we pass it to the print function to display it:

In [ ]:
```
# Example of global variable

myFavouriteBand = "AC/DC"

def getBandRating(bandname):
    if bandname == myFavouriteBand:
        return 10.0
    else:
        return 0.0

print("AC/DC's rating is:", getBandRating("AC/DC"))
print("Deep Purple's rating is:",getBandRating("Deep Purple"))
print("My favourite band is:", myFavouriteBand)
```

Take a look at this modified version of our code. Now the `myFavouriteBand` variable is defined within the `getBandRating` function. A variable that is defined within a function is said to be a local variable of that function. That means that it is only accessible from within the function in which it is defined. Our `getBandRating` function will still work, because `myFavouriteBand` is still defined within the function. However, we can no longer print `myFavouriteBand` outside our function, because it is a local variable of our `getBandRating` function; it is only defined within the `getBandRating` function:

In [ ]:
```
# Example of local variable

def getBandRating(bandname):
    myFavouriteBand = "AC/DC"
    if bandname == myFavouriteBand:
        return 10.0
    else:
        return 0.0

print("AC/DC's rating is: ", getBandRating("AC/DC"))
print("Deep Purple's rating is: ", getBandRating("Deep Purple"))
print("My favourite band is", myFavouriteBand)
```

Finally, take a look at this example. We now have two `myFavouriteBand` variable definitions. The first one of these has a global scope, and the second of them is a local variable within the `getBandRating` function. Within the `getBandRating` function, the local variable takes precedence. **Deep Purple** will receive a rating of 10.0 when passed to the `getBandRating` function. However, outside of the `getBandRating` function, the `getBandRating` s local variable is not defined, so the `myFavouriteBand` variable we print is the global variable, which has a value of **AC/DC**:

In [ ]:
```
# Example of global variable and local variable with the same name

myFavouriteBand = "AC/DC"

def getBandRating(bandname):
    myFavouriteBand = "Deep Purple"
    if bandname == myFavouriteBand:
        return 10.0
    else:
        return 0.0

print("AC/DC's rating is:",getBandRating("AC/DC"))
print("Deep Purple's rating is: ",getBandRating("Deep Purple"))
print("My favourite band is:",myFavouriteBand)
```

Collections and Functions

When the number of arguments are unknown for a function, They can all be packed into a tuple as shown:

In [ ]:
```
def printAll(*args): # All the arguments are 'packed' into args which can be treated like a tuple
    print("No of arguments:", len(args))
    for argument in args:
        print(argument)
#printAll with 3 arguments
printAll('Horsefeather','Adonis','Bone')
#printAll with 4 arguments
printAll('Sidecar','Long Island','Mudslide','Carriage')
```

Similarly, The arguments can also be packed into a dictionary as shown:

In [10]:
```
def printDictionary(**args):
    for key in args:
        print(key + " : " + args[key])

printDictionary(Country='Canada',Province='Ontario',City='Toronto')
```

```
Country : Canada
Province : Ontario
City : Toronto
```

Functions can be incredibly powerful and versatile. They can accept (and return) data types, objects and even other functions as arguements. Consider the example below:

In [8]:
```
def addItems(list):
    list.append("Three")
    list.append("Four")

myList = ["One","Two"]

addItems(myList)

myList
```

Out[8]:
```
['One', 'Two', 'Three', 'Four']
```

Note how the changes made to the list are not limited to the functions scope. This occurs as it is the lists **reference** that is passed to the function - Any changes made are on the orignal instance of the list. Therefore, one should be cautious when passing mutable objects into functions.

Quiz on Functions

Come up with a function that divides the first input by the second input:

In [11]:
```
# Write your code below and press Shift+Enter to execute
def div(a,b):
    return(a/b)
```

Double-click **here** for the solution.

Use the function `con` for the following question.

In [12]:
```
# Use the con function for the following question

def con(a, b):
    return(a + b)
```

Can the `con` function we defined before be used to add to integers or strings?

In [ ]:
```
# Write your code below and press Shift+Enter to execute
#Yes, Function can take any values
```

Double-click **here** for the solution.

Can the `con` function we defined before be used to concatenate a list or tuple?

In [ ]:
```
# Write your code below and press Shift+Enter to execute
Yes
```

Double-click **here** for the solution.

Probability Bag

You have been tasked with creating a lab that demonstrates the basics of probability by simulating a bag filled with colored balls. The bag is represented using a dictionary called "bag", where the key represents the color of the ball and the value represents the no of balls. The skeleton code has been made for you, do not add or remove any functions. Complete the following functions -

- fillBag - A function that packs it's arguments into a global dictionary "bag".
- totalBalls - returns the total no of balls in the bucket
- probOf - takes a color (string) as argument and returns probability of drawing the selected ball. Assume total balls are not zero and the color given is a valid key.
- probAll - returns a dictionary of all colors and their corresponding probability

In [13]:
```
def fillBag(**balls):
    pass


def totalBalls():
    pass

def probOf(color):
    pass

def probAll():
    pass
```

Run this snippet of code to test your solution. Note: This is not a comprehensive test.

In [14]:
```
testBag = dict(red = 12, blue = 20, green = 14, grey = 10)
total =  sum(testBag.values())
prob={}
for color in testBag:
    prob[color] = testBag[color]/total;

def testMsg(passed):
    if passed:
       return 'Test Passed'
    else :
       return ' Test Failed'

print("fillBag : ")
try:
    fillBag(**testBag)
    print(testMsg(bag == testBag))
except NameError as e:
    print('Error! Code: {c}, Message: {m}'.format(c = type(e).__name__, m = str(e)))
except:
    print("An error occured. Recheck your function")



print("totalBalls : ")
try:
    print(testMsg(total == totalBalls()))
except NameError as e:
    print('Error! Code: {c}, Message: {m}'.format(c = type(e).__name__, m = str(e)))
except:
    print("An error occured. Recheck your function")

print("probOf")
try:
    passed = True
    for color in testBag:
           if probOf(color) != prob[color]:
                passed = False

    print(testMsg(passed) )
except NameError as e:
    print('Error! Code: {c}, Message: {m}'.format(c = type(e).__name__, m = str(e)))
except:
    print("An error occured. Recheck your function")

print("probAll")
try:
    print(testMsg(probAll() == prob))
except NameError as e:
    print('Error! Code: {c}, Message: {m}'.format(c = type(e).__name__, m = str(e)))
except:
    print("An error occured. Recheck your function")
```

```
fillBag :
Error! Code: NameError, Message: name 'bag' is not defined
totalBalls :
 Test Failed
probOf
 Test Failed
probAll
 Test Failed
```

Double-click **here** for the solution.

The last exercise!

Congratulations, you have completed your first lesson and hands-on lab in Python. However, there is one more thing you need to do. The Data Science community encourages sharing work. The best way to share and showcase your work is to share it on GitHub. By sharing your notebook on GitHub you are not only building your reputation with fellow data scientists, but you can also show it off when applying for a job. Even though this was your first piece of work,

it is never too early to start building good habits. So, please read and follow this article to learn how to share your work.

Author

Joseph Santarcangelo

Other contributors

Mavis Zhou

Change Log

Date (YYYY-MM-DD)	Version	Changed By	Change Description
2020-08-26	0.2	Lavanya	Moved lab to course repo in GitLab
2020 -09 -04	0.2	Arjun	Under What is a function, added code/text to further demonstrate the functionality of the return statement
2020 -09 -04	0.2	Arjun	Under Global Variables, modify the code block to try and print 'internal_var' - So a nameError message can be observed
2020 -09 -04	0.2	Arjun	Added section Collections and Functions
2020 -09 -04	0.2	Arjun	Added exercise "Probability Bag"