FUNCTIONS

In this video, we will cover functions. You will learn how to use some of Python's built-in functions, as well as how to build your own functions.

General recipe for calling functions:

Output = function\_name(input)

# Create variables var1 and var2

var1 = [1,2,3,4)

var2 = True

#print out type of var1

print(type(var1))

#print out length of var1

print(len(var1))

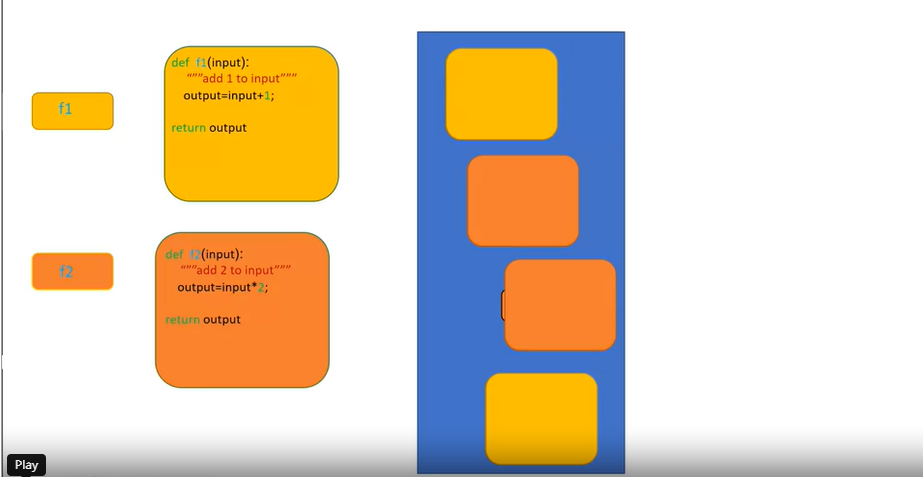
#Convert var2 to an integer: out2

Out2 = int(var2)

Functions take some input, then produce some output or change. The function, it's just a piece of code you can reuse.

You can implement your own function, but in many cases, you use other people's functions. In this case, you just have to know how the function works, and in some cases, how to import the functions.

Let the orange and yellow squares represent similar blocks of code. We can run the code using some input and get an output. If we define a function to do the task, we just have to call the function.



Let the small squares represent the lines of code used to call the function. We can replace these long lines of code by just calling the function a few times. Now we can just call the function. Our code is much shorter. The code performs the same task.

def function(a):

""add 1 to a""

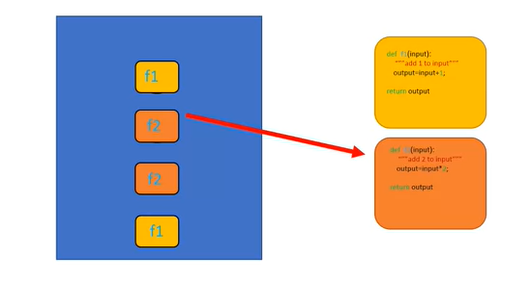
b=a+1;

print(a, "+1 = ",b)

return b

You can think of the process like this. When we call the function F1, we pass an input to the function. These values are passed to all those lines of code you wrote. This returns a value. You can use the value.

For example, you can input this value to a new function F2.



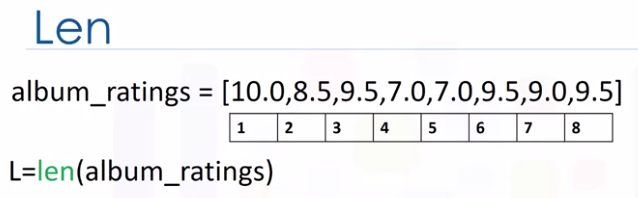
When we call this new function F2, the value is passed to another set of lines of code. The function returns a value. The process is repeated, passing the values to the function you call. You can save these functions and reuse them or use other people's functions.

PYTHON BUILT IN FUNCTIONS

Python has many built-in functions. You don't have to know how those functions work internally, but simply what task those functions perform.

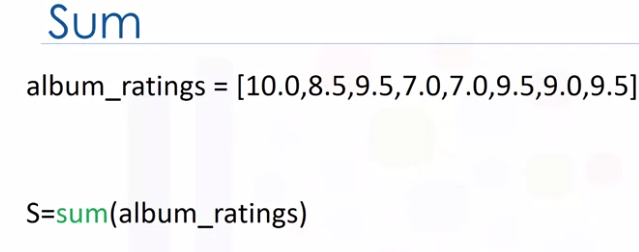
LEN

The function LEN takes in an input of type sequence such as a string, or list, or type collection such as a dictionary or set and returns the length of that sequence or collection. Consider the following list. The LEN function takes this list as an argument, and we assign the result to the variable L.



The function determines there are eight items in the list, then returns the length of the list. In this case, eight. The function sum takes in an iterable like a tuple or list and returns the total of all the elements.

Consider the following list.



We passed the list into the sum function and assign the result to the variable S. The function determines the total of all the elements, then returns it. In this case, the value is 70.

Another example:

sum([0,0,1])

= 1

len([sum([0,0,1])])

= 1

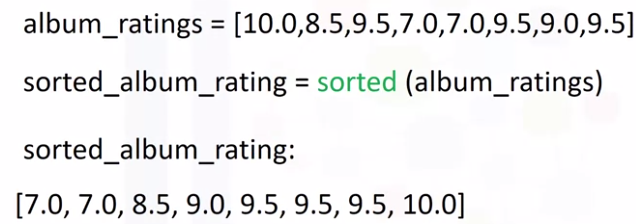
SORTED VS SORT

There are two ways to sort a list. The first is using the function sorted. We can also use the list method sort.

Methods are similar to functions. Let's use this as an example to illustrate the difference.

SORTED

**The function sorted returns a new sorted list or tuple**. Consider the list album\_ratings.



album\_ratings=[10.0,8.5,9,5,7.0,7.0,9.5,9.0,9.5]

sorted\_album\_rating=sorted(album\_ratings)

sorted\_album\_rating

= [5, 7.0, 7.0, 8.5, 9, 9.0, 9.5, 9.5, 10.0]

We can apply the function sorted to the list album\_ratings and get a new list sorted\_album\_rating. **The result is a new sorted list.**

If we look at the list album\_ratings, nothing has changed.

What is the value of list L after the following code segment is run :

L=[1,3,2]

sorted(L)

Answer = L=[1,3,2] because **sorted** is a function and returns a new list, it does not change the list **L**

Generally, functions take an input. In this case, a list. They produce a new output. In this instance, a sorted list.

SORT

If we use the method sort, the list album\_ratings will change and **no new list will be created**.

Let's use the diagram to help illustrate the process.

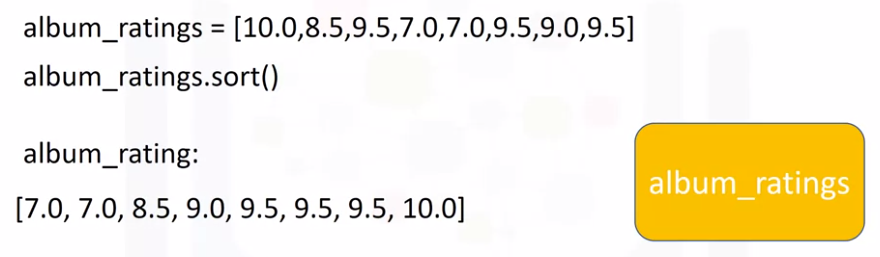
album\_ratings=[10.0,8.5,9,5,7.0,7.0,9.5,9.0,9.5]

album\_ratings.sort()

Album\_ratings

=[5, 7.0, 7.0, 8.5, 9, 9.0, 9.5, 9.5, 10.0]

In this case, the rectangle represents the list album\_ratings. When we apply the method sort to the list, the list album\_rating changes. Unlike the previous case, we see that list album\_rating has changed. In this case, no new list is created.



Now that we've gone over how to use functions in Python, let's see how to build our own functions.

MAKING FUNCTIONS

We will now get you started on building your own functions in Python.

This is an example of a function in Python that returns its input value plus one. To define a function, we start with the keyword def.

def add1(a):

b=a+1

return b

#call function

add1(5)

= 6

The name of the function should be descriptive of what it does. We have the function formal parameter “a” in parentheses, followed by a colon. We have a code block with an indent. For this case, we add one to a and assign it to b. We return or output the value for b. After we define the function, we can call it. The function will add one to five and return a six.

We can call the function again. This time assign it to the variable c. The value for c is 11. Let's explore this further.

def add1(a):

b=a+1

return b

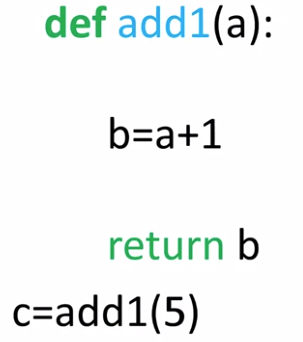
c=add1(10)

c

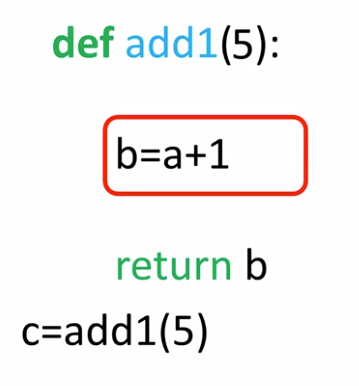
= 11

I.E. 10 + 1 = 11

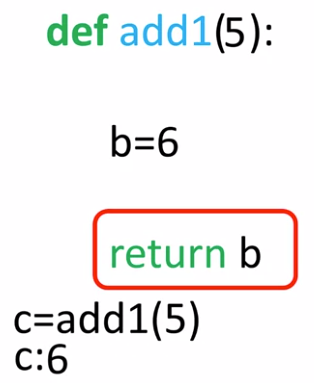
Let's go over an example when you call a function. It should be noted that this is a simplified model of Python, and Python does not work like this under the hood.



We call the function giving it an input five. It helps to think of the value of five is being passed to the function. Now the sequences of commands are run.

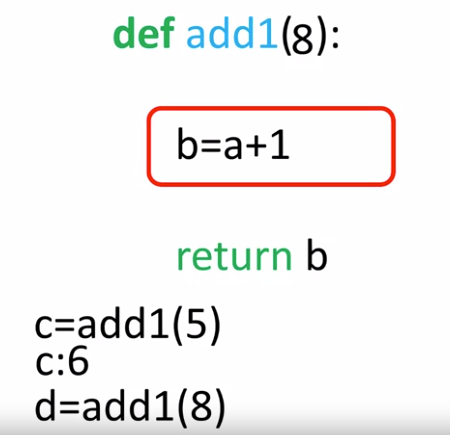


The value of a is five. B would be assigned a value of six. We then return the value of b. In this case, as b was assigned a value of six, the function returns a six.

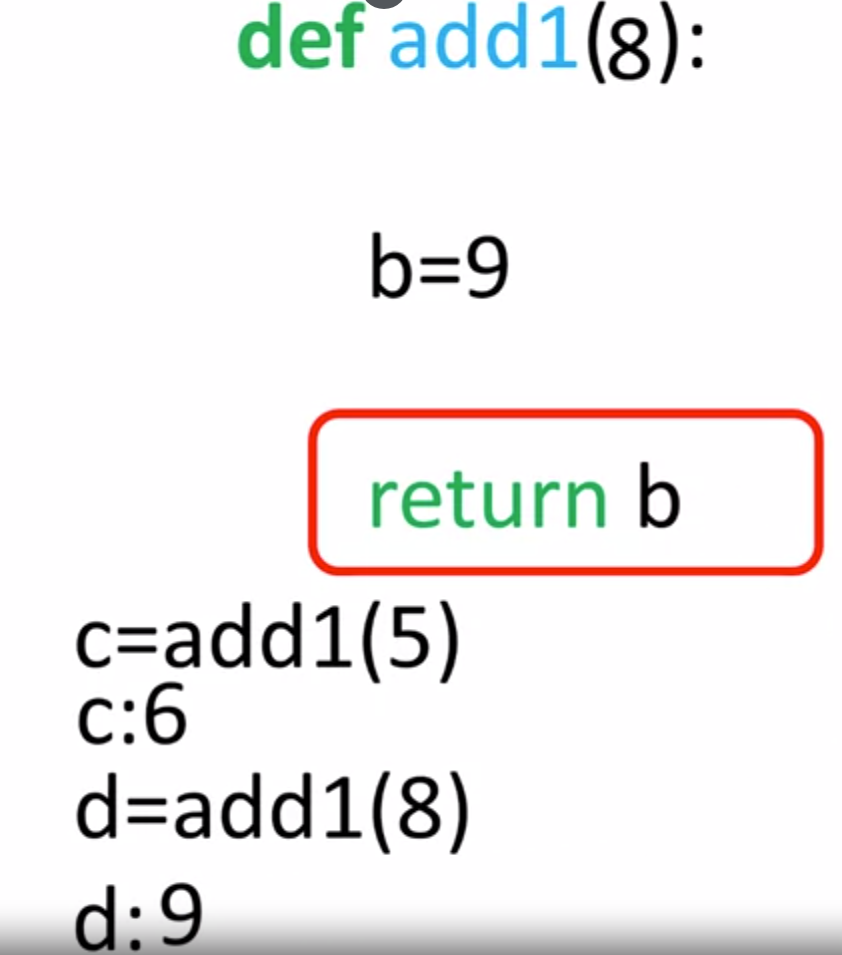


If we call the function again, the process starts from scratch.

We pass in an eight. The subsequent operations are performed.

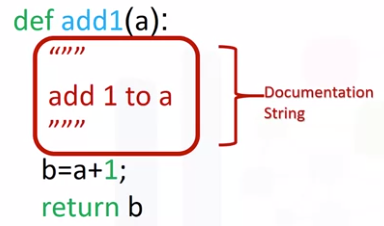


Everything that happened the last call will happen again with a different value of a. The function returns a value. In this case, nine. Again, this is just a helpful analogy.

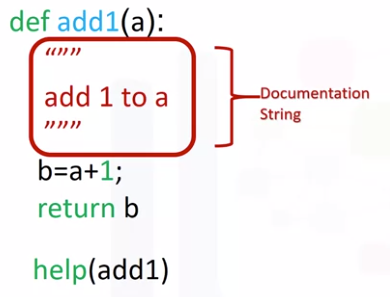


Let's try and make this function more complex.

It's customary to document the function on the first few lines. This tells anyone who uses the function what it does. This documentation is surrounded in triple quotes.



You can use the help command on the function to display the documentation as follows.



This will print out the function name and the documentation. We will not include the documentation in the rest of the examples.

MULT

A function can have multiple parameters. The function mult, multiplies two numbers. In other words, it finds their product.

def Mult(a,b):

c=a\*b

return c

Mult(2,3)

= 6

Mult(10,3.14)

= 31.4

If we pass the integers two and three, the result is a new integer. If we pass the integer 10 and the float 3.14, the result is a float 31.4.

If we pass in the integer two and the string "Michael Jackson," the string "Michael Jackson" is repeated two times. This is because the multiplication symbol can also mean repeat a sequence.

Mult(2,'Michael Jackson')

= 'Michael JacksonMichael Jackson'

If you accidentally multiply an integer with a string instead of two integers, you won't get an error. Instead, you will get a string and your program will progress, potentially failing later because you have a string where you expected an integer.

This property will make coding simpler, but you must test your code more thoroughly. In many cases, a function does not have a return statement. In these cases, Python will return the special none object.

Practically speaking, if your function has no return statement, you can treat it as if the function returns nothing at all. The function MJ simply prints the name Michael Jackson. We call the function. The function prints Michael Jackson.

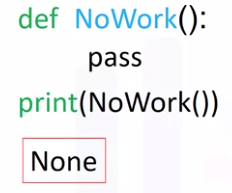
def MJ():

print('Michael Jackson')

MJ()

NONE

Let's define the function NoWork that performs no task. Python doesn't allow a function to have an empty body. So we can use the keyword pass, which doesn't do anything, but satisfies the requirement of a none empty body.

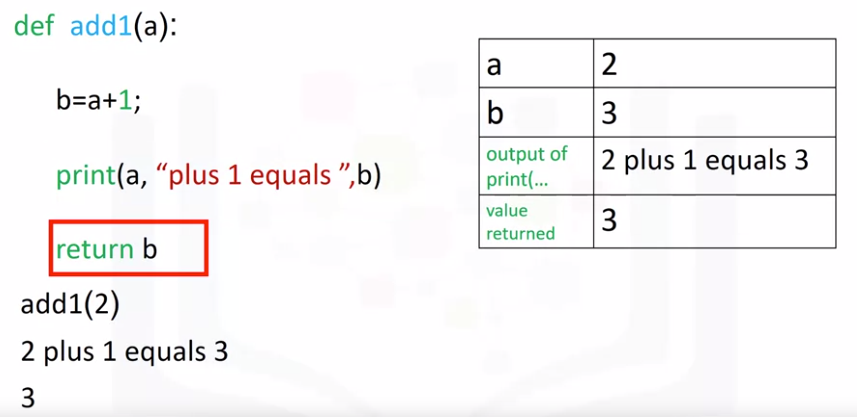


If we call the function and print it out, the function returns a none. In the background, if the return statement is not called, Python will automatically return a none. It is helpful to view the function NoWork with the following return statement.



Usually, functions perform more than one task. This function prints a statement, then returns a value. Let's use this table to represent the different values as the function is called.

We call the function with an input of two. We find the value of b. The function prints the statement with the value of a and b. Finally, the function returns the value of b. In this case, three.



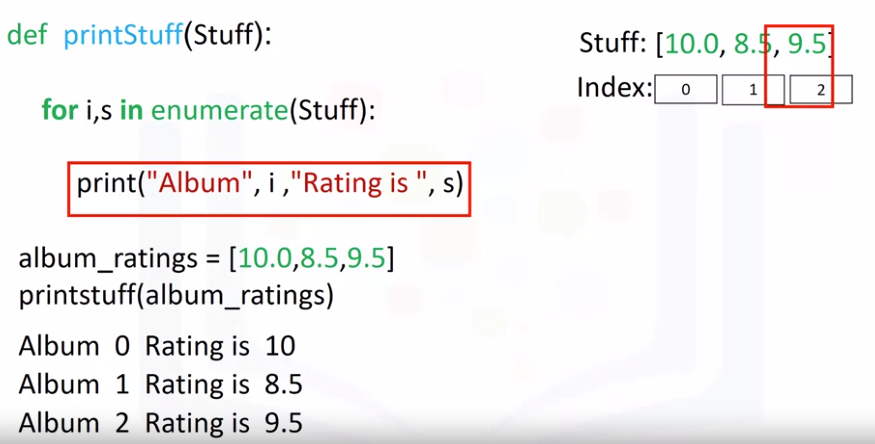
We can use loops in functions. This function prints out the values and indexes of a loop or tuple.

We call the function with a list album\_ratings as an input. Let's display the list on the right with its corresponding index.

Stuff is used as an input to the function enumerate. **This operation will pass the index to i and the value in the list to s**. The function would begin to iterate through the loop. The function will print the first index and the first value in the list. We continue iterating through the loop. The values of i and s are updated. The print statement is reached.

Similarly, the next values of the list and index are printed. The process is repeated. The values of i and s are updated.

We continue iterating until the final values in the list are printed out.



Examples:

def Print(A):

for a in A:

print(a+'1')

Print(['a','b','c'])

=a1

b1

c1

def Print(A):

for c in A:

print(c+'1')

Print(['a','b','c'])

= a1

b1

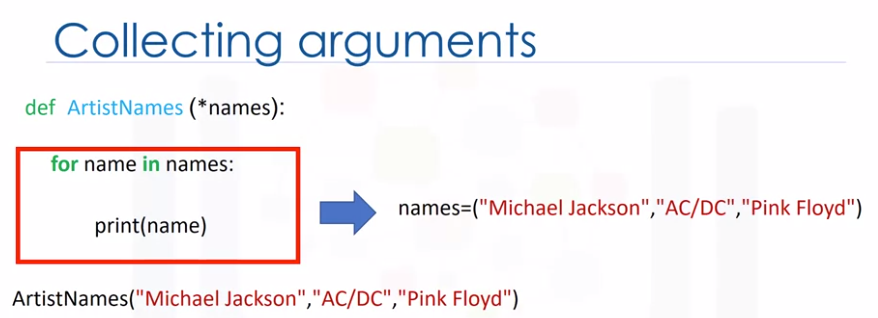
C1

Not sure what is happening here

VARIADIC PARAMETERS

Variadic parameters allow us to input a variable number of elements. Consider the following function.

The function has an asterisk on the parameter names. When we call the function, three parameters are packed into the tuple names. We then iterate through the loop. The values are printed out accordingly.



ArtistNames("mike jackson", "ac/dc","pink floyd")

def ArtistNames(\*names):

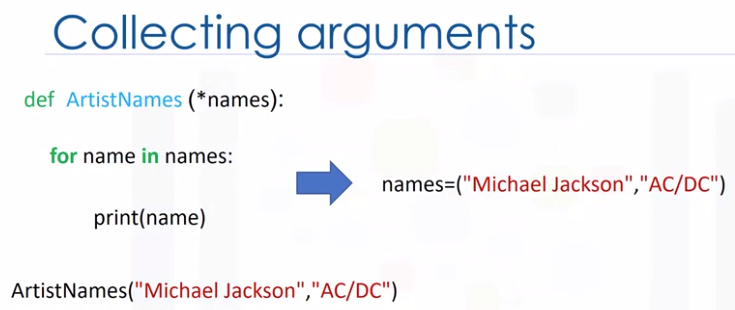
for name in names:

names=("mike jackson", "ac/dc","pink floyd")

print(name)

ArtistNames

If we call the same function **with only two parameters as inputs**, the variable names only contain two elements. The result is only two values are printed out.



ArtistNames("mike jackson", "ac/dc")

def ArtistNames(\*names):

for name in names:

names=("mike jackson", "ac/dc","pink floyd")

print(name)

ArtistNames

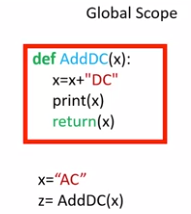
= mike jackson

ac/dc

SCOPE

The scope of a variable is the part of the program where that variable is accessible.

Variables are defined outside of any function are said to be within the global scope, meaning they can be accessed anywhere after they are defined.



def AddDC(x):

x=x+"DC"

print(x)

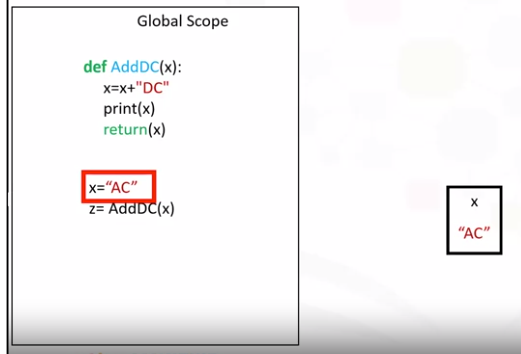
return(x)

x="AC" #\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* global variable

z=AddDC(x)

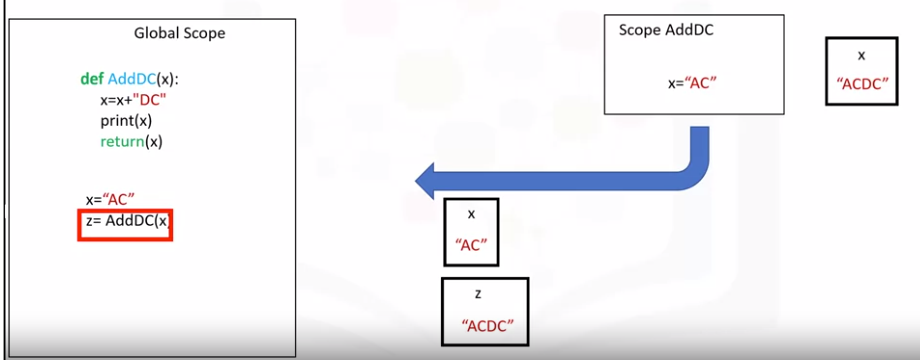
= ACDC

Here we have a function that adds the string "DC" to the parameter x. When we reach the part where the value of x is set to "AC", this is within the global scope, meaning X is accessible anywhere after it is defined.



**Global Scope**

A variable defined in the global scope is called a global variable. When we call the function, we enter a new scope or the scope of AddDC. We pass as an argument to the AddDC function. In this case, "AC". Within the scope of the function, the value of x is set to "ACDC". The function returns the value and is assigned to z.



Within the global scope, the value z is set to "ACDC." After the value is returned, the scope of the function is deleted.

**Local Variables**

Local variables only exist within the scope of a function.

def Thriller():

Date=1982

return Date

Thriller()

= 1982

Consider the function thriller. The local variable date is set to 1982. When we call the function, we create a new scope. Within that scope of the function, the value of the date is set to 1982.



The value of date does not exist within the global scope.

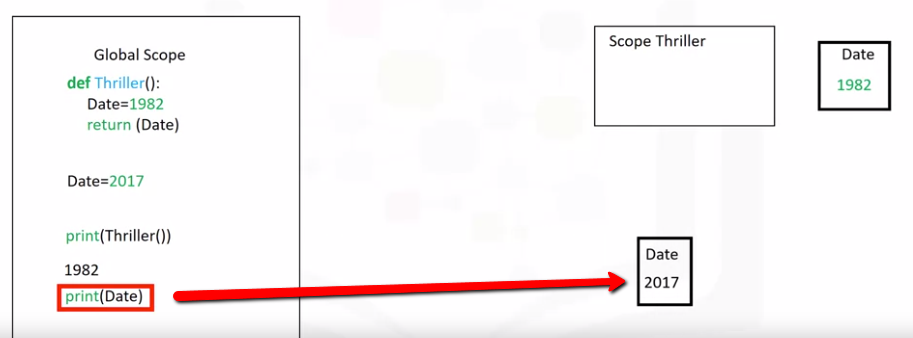
Variables inside the global scope can have the same name as variables in the local scope with no conflict.

Consider the function thriller. The local variable date is set to 1982. The global variable date is set to 2017.



When we call the function, we create a new scope. Within that scope, the value of the date is set to 1982. If we call the function, it returns the value of date in the local scope. In this case, 1982.

When we print in the global scope, we used the global variable value. The global value of the variable is 2017. Therefore, the value is set to 2017.



If a variable is not defined within a function, Python will check the global scope.

Consider the function "ACDC." The function has the variable rating with no value assigned.

#\*\*\*\* Global scope

def ACDC(y):

print(rating)

return(rating+y)

rating = 9

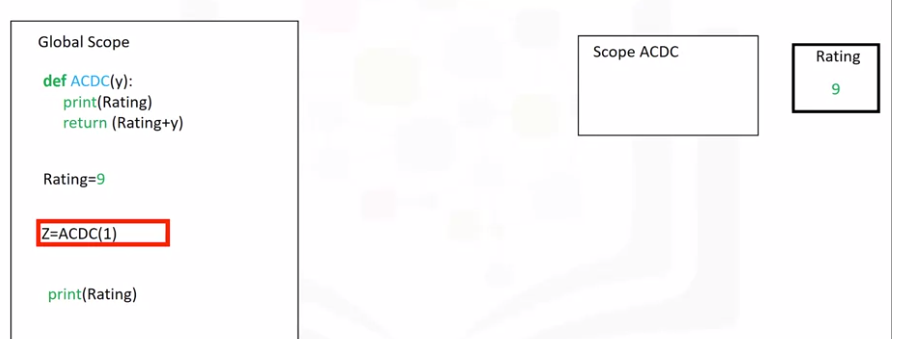
Z=ACDC(1)

print(rating)

= 9

9

If we define the variable rating in the global scope, then call the function, Python will see there is no value for the variable rating. As a result, Python will leave the scope and check of the variable ratings exists in the global scope. It will use the value of ratings in the global scope within the scope of "ACDC."



#\*\*\*\* Global scope

def ACDC(y):

print(rating)

return(rating+y)

rating = 9

Z=ACDC(1)

print(rating)

In the function, we'll print out a nine. The value of z in the global scope will be 10, as we added one. The value of rating will be unchanged within the global scope.

def ACDC(y):

print(rating)

return(rating+y)

rating = 9

Z=ACDC(1)

print(Z)

= 10



Consider the function PinkFloyd. If we define the variable ClaimedSales with the keyword global, the variable will be a global variable. We call the function PinkFloyd. The variable ClaimedSales is set to the string 45 million in the global scope. When we print the variable, we get a value of 45 million.

def PinkFloyd():

global ClaimedSales

ClaimedSales='45 million'

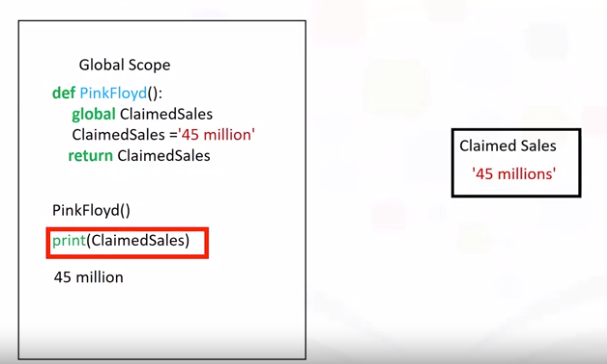
return ClaimedSales

PinkFloyd()

print(ClaimedSales)

= '45 million'

There is a lot more you can do with functions.



Check out the lab for more examples

Functions from exercise:

def div(a):

b = a /2

print(a, "if you divide by a by 2, you get ", b)

# return(b)

div(6)

From Lab:

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**

a1 = 4

b1 = 5

c1 = a1 + b1 + 2 \* a1 \* b1 -1

if(c1 < 0):

c1 = 0

else:

c1 = 5

c1

**Block 2**

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:

def equation(a,b):

c = a + b + 2 \* a \* b - 1

if(c < 0):

c = 0

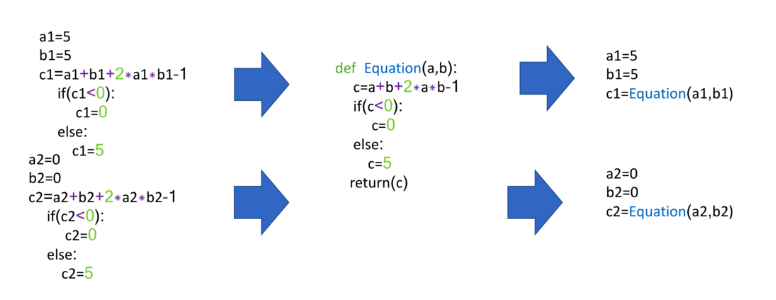
else:

c = 5

return(c)

equation(1,2)

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**

a1 = 4

b1 = 5

c1 = Equation(a1, b1)

c1

**Block 4**

a2 = 0

b2 = 0

c2 = Equation(a2, b2)

c2

Create **global variables** from within a function as follows:

artist = "Michale Jackson"

def printer(artist):

global internal\_var

internal\_var = "Whitney Houston - Global var"

print(artist, " is an artist - Local var")

printer(artist)

printer(internal\_var)

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:

**# 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)

AC/DC's rating is: 10.0

Deep Purple's rating is: 0.0

My favourite band is: AC/DC

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:

**# 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)

AC/DC's rating is: 10.0

Deep Purple's rating is: 0.0

My favourite band is: AC/DC

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**:

**# 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)

AC/DC's rating is: 0.0

Deep Purple's rating is: 10.0

My favourite band is: AC/DC