Functions and Scoping

Function

The function is a block of effectively grouped related codes which is utilized to execute a single, connected action and can be reused multiple times. For example, print() is a python built-in function, which we use multiple times by calling it with its name. Similar to this one, we can make custom functions and these are called “user-defined functions”.

Defining a function

Basic structure of a Function

```
def function_name (parameter_list):
        """docstring"""
        Body_of_function

        return [expression]
```

Example of function

```
def greetings():
     #function body starts
     """This function prints greetings"""
     print("Hello Students")
     print("I hope you guys are loving Python")
     #function body starts
```

```
def makeCube(number):
     #function body starts
     '''This function makes cube of the
     given number in the parameter'''
     print(number**3)
     #function body ends
```

1. **Def**: It is a Python reserved keyword which is used to inform Python that a function is being defined or started.

2. **Name_of_function:** It is used for referring or calling to the function. For example, in the above code, `makeCube` is the function name. Function names can be anything except the Python

reserved keywords. Similar to variable naming, it has to maintain certain rules.

Function names-

- can have letters (A-Z and a-z), digits(0-9), and underscores.
- cannot begin with digits.
- cannot have whitespace and special signs (e.g.: +, -, !, @, $, #, %.)
- are case sensitive meaning ABC, Abc, abc are three different variables.
- cannot be a reserved keyword for Python.

Note: For details read from the link below.
Link: https://www.python.org/dev/peps/pep-0008/#function-and-variable-names

3. **Parameters (arguments):** We can have an empty parameter (no variables within the parentheses) or multiple parameters separated with commas. Parameter_list is used to pass values(or inputs) in the function. These are optional. Here, in the above example, number is a parameter.

4. **Docstring:** This triple quoted string is written in the first statement of the function and contains documentation or the purpose of the function. It is **optional**. But maintaining docstring is a convention. Python has built-in functions to access these docstrings.

5. **Body_of_function:**Within the function, codes related to producing a certain result are written in an indented manner in the function body. It can have multiple sequential or conditional statements.

Function call/invocation

If we run the previous two examples, we will get no output or result. In order to invoke the function, only defining it is not enough. We need to call the function with its function name, accompanied by the parentheses. While invoking/calling, the parentheses can have no parameters or multiple parameters separated by commas depending on the structure of the defined function. Functions can be called directly or from other functions or straight from the Python prompt.

Code	Output

`def greetings():`
`  """This function prints greetings"""`
`  print("Hello Students")`
`  print("I hope you are loving Python")`
 	
`#driver code`
`#calling the function with an empty parameter`
`greetings()`
	`Hello Students`
	`I hope you are loving Python`

def makeCube(number):
  '''This function makes cube of the given number in the parameter'''
  print(number**3)
 	
#driver code
#calling the function with input 2
makeCube(2)
#calling the function with input 3
makeCube(3)
	8
	27

Parameters (arguments):

1. **Positional arguments:** During the function call, all the arguments must be provided and they have to be in the same order as the function definition. Here position or order is important.

Code: Function definition
def printInfo(name, age, height):   print("Name:", name, "Age:", age,"Height:", height, "cm")

Code: Function call	**Output**
#calling with proper order printInfo("John", 20, 167)	Name: John Age: 20 Height: 167
Here, "John" is being mapped to name, 20 is being mapped to age, and 167 is being mapped to height.	

Code: Function call	**Output**

#calling with wrong order
printInfo(20, 167, "John")
	Name: 20 Age: 167 Height: John

Here, 20 is being mapped to name, 167 is being mapped to age, and "John" is being mapped to height. Since the order of position during the function call was wrong, so our mapping was also wrong. So, finally the output of the code was wrong.	

2. **Default arguments:**

When a function has one or multiple arguments with default values in the function definition,

then few of arguments (with default values) can be excluded during function call. Then, the excluded arguments by default take the values provided in the function definition. If no default values are set in the function for the excluded arguments, then Python raises an error, TypeError.

Code: Function with default argument

```
def greetings(quizzes, name = "students" ):
     """This function prints greetings"""
     print("Hello " + name + "!")
     print("Hope you are loving Python.")
     print("You will have " + str(quizzes) + " this term")
     print("==========================")
```

Code: Function call	Output
#1)calling the function with values john and 10 greetings(10, "John")	Hello John! Hope you are loving Python. You will have 10 this term ==========
Here, 10 is being mapped to the quizzes variable and "John" is being mapped to the name variable, overwriting the default value,"students".	

Code: Function call	Output
#2)calling the function with only 10, skipping the name greetings(10)	Hello students! Hope you are loving Python. You will have 10 this term ==========
Here, 10 is being mapped to the quizzes variable and since no value has been provided for the name variable, the default value "students" is used for the name variable.	

Code: Function call	Output
#3)calling the function with an empty parameter greetings()	TypeError: greetings() missing 1 required positional argument: 'quizzes'
Here, the function had been tried to call with an empty parameter. But, since quizzes do not have any default values defined in the function, the Python raises an error.	

Another thing to keep in mind, the non-default argument(parameter with no defined values) must come before the default argument (parameter with defined values). Otherwise, Python will raise an error, SyntaxError.

Code	Output
def greetings(name = "students", quizzes): """This function prints greetings""" print("Hello " + name + "!") print("Hope you are loving Python.") print("You will have " + str(quizzes) + " this term") print("==========")	SyntaxError: non-default argument follows default argument

3. **Keyword arguments:** During the function call, all the arguments must be provided, but they can be in random order. Here position or order is not important.

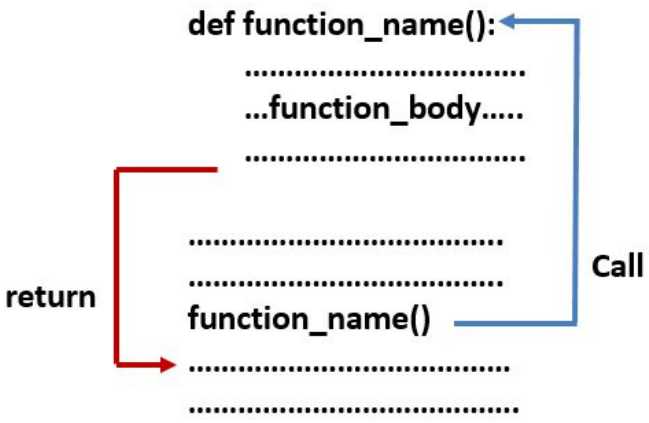
Code: Function definition
def greetings(quizzes, name = "students"): """This function prints greetings""" print("Hello " + name + "!") print("Hope you are loving Python.") print("You will have " + str(quizzes) + " this term") print("==========")

Code	Output
#1)calling with keyword argument greetings(name = "John", quizzes = 10)	Hello John! Hope you are loving Python. You will have 10 this term ==========
#2)calling without keyword argument greetings("John", 10)	TypeError: can only concatenate str (not "int") to str Here, "John" is being mapped to the quizzes variable and 10 is being mapped to the name variable, overwriting the default value,"students".

The *return* Statement

When a function is called, before exiting the function body, it returns a value to the point where the function was called from. If no values are returned from the function body, then by default it returns None (no values at all). In order to see the default value, we need to print the function.

By the return statement, results from the function can be passed to the main program. This results provided by the function can be saved for later use in the program, or it can be used to make a more complex expression. While being called, a function can only execute the return statement once. Printing inside a function is mainly done for human consumption (people to see the output).



Code	Output

def checkEven(num):
  if num%2 == 0:
  print("even")
  else:
  print("Odd")
 	
print(checkEven(3))
	Odd
	None

def checkEven(num):
  if num%2 == 0:
  print("even")
  else:
  print("Odd")
 	
  return
 	
answer = checkEven(3)
print(answer)
	Odd
	None

def checkEven(num):
  if num%2 == 0:
  return "even"
  else:
  return "Odd"
 	
answer = checkEven(3)
print(answer)
	Odd

def checkEven(num):
  if num%2 == 0:
  return "even"
  return "2nd return in if"
  else:
  return "Odd"
  return "return in function body"
 	
print(checkEven(2))
	even

def checkEven(num):
  return "return in function body"
  if num%2 == 0:
  return "even"
  else:
  return "Odd"
 	
print(checkEven(2))
	return in function body
	
	Here, the first return statement is being executed.

Necessity of Functions

1) Code reusability: Suppose we want to calculate the area of 3 different lands. What you have learned so far, for calculating the area, we need to write the same code 3 times. But using a function we can do it only once and reuse it.

Code	Output

`def calculateArea(length, width):`
`  '''This function calculates area of a land`
`  with the given parameters length and`
`  width'''`
 	
`  print(length*width)`
 	
`calculateArea(100, 50)`
`calculateArea(150, 40)`
`calculateArea(200, 30)`
	`5000`
	`6000`
	`6000`

2) Minimize writing identical code within a program.
3) Better modularity: Large programs might have hundreds or thousands lines of codes. Function helps us divide those codes into small easily manageable chunks of codes. So, our code becomes very easy to debug (correcting problems within a code).
4) Easy to manage: The more modular our code is, more manageable and organized it is.
5) Easily used inside a program: Functions made is possible to reuse codes easily inside complex computations without making it more lengthy.
6) Provides decomposition(breaking a huge program into small related reusable code chunks) and abstraction(hides unnecessary information, for example: a function is calculating roots. Everytime we use it, we do not need to know or see 100 lines of codes)

Anonymous Functions

Anonymous functions are written with the lambda keyword. If you are interested to learn more about Lamba read details from the provided link.

Link: https://docs.python.org/3/reference/expressions.html#lambda

Scope (Global and local scope)

Scope

The scope of a variable means the area of the code where that variable can be accessed and modified.

Local Scope & Local Variable

When a variable is created within a function body, then scope of that variable is the function body and the variable is called “Local variable”. Local variables can be modified and accessed within the function body where it was created (its scope). Outside that function, local variables do not exist.

1. Example1: Variable inside a function

Code	Output

`def printHi():`
`  a = "Hi local"`
`  print(a)`	

`printHi()`	`Hi local` Here, a is a local variable of the function, so it can be accessed within that function

`#trying to access a local variable, outside function`
 	
`print(a)`
	`NameError: name 'a' is not defined`
	
	Since local variables cannot be accessed from outside the function, it raises an error.

2. Example2: Using Global and Local variables in the same code

Code	Output

`a = "Global Hi"`
`def printHi():`
`  global a`
`  a = a + " Bye"`
`  b = "Local Hi"`
`  print("a inside: " + a)`
`  print("b inside: " + b)`
 	
`printHi()`
	`a inside: Global Hi Bye`
	`b inside: Local Hi`

2. Example3: Global variable and Local variable with same name

Code	Output

a = "Global Hi"
def printHi():
  global a
  a = "Local Hi"
  print("Local a: " + a)
 	
printHi()
print("Global a: " + a)
	Local a: Local Hi
	Global a: Local Hi

Global Scope & Global variable

When a variable is created within the main body of the Python code, then scope of that variable is the main body of the Python code and the variable is called "Global variable". Global variables can be modified and accessed by any functions and from any scope (global & local).

1. Example1: Global variables can be accessed from anywhere.

Code	Output
a = "Global Hi" def printHi(): print("a inside: " + a) printHi() print("a outside: " + a)	a inside: Global Hi a outside: Global Hi

2. Example 2: When we try to change the global variable inside a function, Python treats it as a local variable.

Code	Output

a = "Global Hi"
def printHi():
  a = a + "Bye"
  print("a inside: " + a)
 	
printHi()
print("a outside: " + a)
	UnboundLocalError: local variable 'a' referenced before assignment
	
	Here, Python is considering 'a' to be a local variable. Since 'a' has not been initialized with a value yet, its value cannot be updated.

Global Keyword

When Global variables are modified inside a function. It creates a local copy of that global variable with the updated value and the values of the global variable is never updated.

Code	Output

a = "Global Hi"
def printHi():
  a = "Local Hi" # local variable a
  print("a inside: " + a)
 	
printHi()
print("a outside: " + a)
	a inside: Local Hi
	a outside: Global Hi

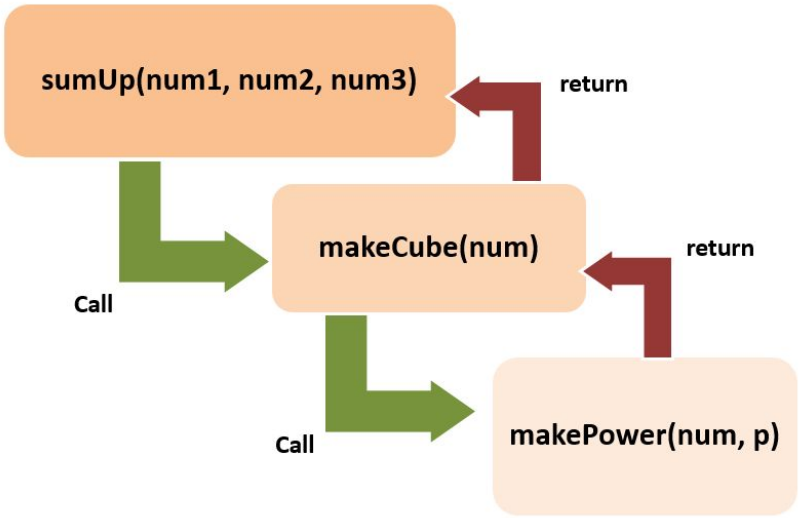
For solving this problem, before updating the global variable, the **global** keyword is used inside the function. The **global** keyword makes the variable work as a global variable. Then, instead of creating a local copy like the previous code, the global variable's value is updated.

Code	Output

a = "Global Hi"
def printHi():
  global a
  a = "Local Hi"
  print("a inside: " + a)
 	
printHi()
print("a outside: " + a)
	a inside: Local Hi
	a outside: Local Hi

Functional decomposition:

When a large problem is broken down into smaller sub-problems, it is called Functional decomposition.
In Python, all the functions can be called from other functions. The function calling another function is known as "Calling function", whereas the function which is being called, is known as "Called function" Here, in the example our problem has been solved using multiple functions and inner function calling.



Code	Output

def makePower(num, p):
  result = num ** p
  return result
 	
def makeCube(num):
  result = makePower(num, 3)#calling the makePower
  return result
 	
print(makeCube(5))
	125

def makePower(num, p):
  result = num ** p
  return result
 	
def makeCube(num):
  result = makePower(num, 3)
  return result
 	
def sumUp(num1, num2, num3):
  s1 = makeCube(num1)
  s2 = makeCube(num2)
  s3 = makeCube(num3)
  return s1 + s2 + s3
 	
print(sumUp(1, 2, 3))
	36

Style Guide for Python Code

For every programming language, there are few coding conventions followed by the coding community of that language. All those conventions or rules are stored in a collected document manner for the convenience of the coders, and it is called the “Style Guide” of that particular programming language. The provided link gives the style guidance for Python code comprising the standard library in the main Python distribution.

Python style guide link: https://www.python.org/dev/peps/pep-0008/