Assignment No. 3

Submitted By: Sachin Dodake

1. Why are `functions` advantageous to have in your programs?

`Ans-1`

> - Python Functions is a block of statements that return the specific task. The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again.
> - Functions provide better modularity for your application and a high degree of code reusing.
> - We may call the function and reuse the code contained within it with different variables rather than repeatedly creating the same code block for different input variables.
> - **Syntax:** The syntax to declare a function is:

```
def function_name(arguments):
        """Docstring"""
        # function body
        return
```

> - **Parameters:**

```
def :- keyword used to declare a function
function_name :- any name given to the function
arguments :- any value passed to function
docstring :- short explaination about fucntion's working
return (optional) :- returns value from a function
```

> **`Rules for Defining a Function:`**
>
> - Function blocks begin with the keyword **`def`** followed by the **`function name`** and **`parentheses ( ( ) )`**.
> - Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
> - The first statement of a function can be an optional statement - **`the documentation string of the function or docstring`**.
> - The code block within every function starts with a **`colon (:)`** and is **`indented`**.
> - The statement **`return [expression]`** exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as **return None**.

> **`Advantages of Functions in Python:`**
>
> - By including functions, we can prevent repeating the same code block repeatedly in a program.
> - Python functions, once defined, can be called many times and from anywhere in a program.
> - If our Python program is large, it can be separated into numerous functions which is simple to track.
> - The key accomplishment of Python functions is we can return as many outputs as we want with different arguments.

2. When does the code in a `function run`: when it's specified or when it's `called`?

`Ans-2`

> - A function is defined by using the def keyword and giving it a name, specifying the arguments that must be passed to the function, and structuring the code block.
> - After a function's fundamental framework is complete, we can call it from anywhere in the program.
> - Following is the example to call function –

In [8]:
```python
# defining greet function without arguement
def wish():
    '''This function will greet student'''             # docstring to understand function working
    print(f"Good Morning Students!")    # fucntion body to print specific greet msg

wish()                    # calling a 'greet'()  function
```

```
Good Morning Students!
```

In [12]:
```python
# defining greet function with arguement
def greet(name):
    '''This function will greet student'''             # docstring to understand function working
    print(f"Welcome to iNeuron, {name} !")    # fucntion body to print specific greet msg

greet('Sachin')                  # calling a 'greet'()  function
greet('Virat')                  # calling a 'greet'()  function
greet('Rohit')                  # calling a 'greet'()  function
```

```
Welcome to iNeuron, Sachin !
Welcome to iNeuron, Virat !
Welcome to iNeuron, Rohit !
```

In [15]:
```python
# Example Python Code for User-Defined function
def square( num ):
    """
    This function computes the square of the number.
    """
    return num**2

result = square(6)

print( f"The square of the given number is:  {result}" )
```

```
The square of the given number is:  36
```

3. What statement creates a `function`?

`Ans-3`

> - In Python, a function can be created/defined by using the **`def`** keyword, then we can write the function identifier (name) followed by parentheses and a colon.
> - **Syntax:**

```
def function_name(arguments):
          """Docstring"""
          # function body
          return
```

> - **Example:** We can see in the example below how function is created using **def** keyword, with few other elements like function name, function body,

In [20]:
```python
# defining greet function without arguement
def hello():
    '''This function will greet student'''             # docstring to understand function working
    print(f"Hello World !")    # fucntion body to print specific greet msg

hello()                  # calling a 'greet'()  function
```

```
Hello World !
```

4. What is the difference between a `function` and a `function call`?

`Ans-4`

> **`Function:`**
>
> - `Def:-` A Function is block of code than accepts some values processes the desire task on it and return the result value.
> - A function is a piece of code which enhanced the reusability and modularity of your program. It means that piece of code need not be written again.
> - A function is something whic take parameters and do some calculations and operations and returns value.
> - **Example:** Let us assum one simple function 'add' which add's two integer numbers as below,

```
def add(x,y):        # defining a function 'add'
     return x + y      # returning result of function 'add'
```

> **`Function Call:`**
>
> - `Def:-` When a perticular task which is to be perform at any point in a program using a function then that process is called as function call.
> - A function call means invoking or calling that function. Unless a function is called there is no use of that function.
> - Calling a function means saying function, do this. You basically call a function, you don't return it.
> - **Example:** We can call the function 'add' as below,

```
add(2,3)       # calling function 'add' to get result
```

5. How many `global scopes` are there in a Python program? How many `local scopes`?

`Ans-5`

> **`Python Scope:`**
>
> - The scope defines the accessibility of the python **object**.
> - To access the particular variable in the code, the scope must be defined as it cannot be accessed from anywhere in the program. The particular coding region where variables are visible is known as scope.
> - Variables are not visible to the entire code; their visibility can be restricted. Scope verifies which variable can be 'Seen'.
> - The `scope` defines the set of rules which tell us how and where a variable can be searched. The variable is searched either to retrieve a value or for assigning value. The `namespace` is the unique identification of the variable or the method.
> - `Namespace` tells the python interpreter about the name of the object and the location from where it is trying to access it.
> - The various types of Python Scopes are listed below:

```
        i) Local Scope- The local, or function-level, scope, which exists inside functions.
        ii) Global Scope - The global scope, which exists at the module level
        iii) Enclosing Scope or NonLocal- The enclosing, or non-local, scope, which appears in nested functions
        iv) Built-in Scope - The built-in scope, which is a special scope for Python's built-in names
```

> - The `Namespaces` are searched for scope resolution according to the `LEGB` rule. The sequence of **LEGB** is important. The variable is first searched in Local, followed by Enclosed, then global and finally built-in.
> - The LEGB stands for `L` **: Local**, `E` **: Enclosed**, `G` **: Global**, `B` **: Built-in**.

> **`i) Local Scope:`**
>
> - The Variables which are defined in the function are a **`local scope of the variable`**.
> - The Local scope variables are defined in the function body.

> - **Example:** Let's us consider an example, we have taken one variables `num_1= 111` outside the function, and `num_2 = 222` inside the function, so it is not a local variable. As per our definition, the variables which are declared inside the function body is a local variable. Here, `num_2= 222` is a local variable that is declared and printed inside the function `local_test`. But when trying to access `num_2 = 222` and then print the same variable from outside the function ie print(num_2), it raised a NameError. This is because `num_2 = 222` is local to the function - thus, it cannot be reached from outside the function body.

In [125…
```python
# eample-1: explaination of local variable inside function

num_1= 111                          # defining global variable
def local_test():                   # defining function
    num_2 = 222                    #  defining local variable
    print("Local variable value is:",num_2)          # accessing local variable num_2inside function

local_test()                        # calling function `local_test`

print(num_2)                   # We can not access local variable outside function, it will show error as be
```

```
Local variable value is: 222
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [125], in <cell line: 10>()
      6     print("Local variable value is:",num_2)          # accessing local variable num_2inside functio
n
      8 local_test()                        # calling function
---> 10 print(num_2)

NameError: name 'num_2' is not defined
```

> **`ii) Global Scope:`**
>
> - The Variable which can be read from anywhere in the program is known as a global scope.
> - These variables can be accessed inside and outside the function. When we want to use the same variable in the rest of the program, we declare it as global.

> - **Example:** Let us consider an example, we have declared a variable Str, which is outside the function. The function demo is called, and it prints the value of variable Str. To use a global variable inside a function, there is no need to use the global keyword.

In [126…
```python
# explaination on global variable
num_1 = 1111                              # defining global variable
def global_test():
    num_2 = 2222                          # defining local variable
    print(f"Global Variable is: {num_1}")  # accessing local variable inside function
    print(f"Local Variable is: {num_2}")    # accessing global variable inside function

global_test()                                  # calling function

print(f"\nGlobal Variable is: {num_1}")   # accessing global variable outside function
```

```
Global Variable is: 1111
Local Variable is: 2222

Global Variable is: 1111
```

> **`iii) Enclosing Scope or NonLocal:`**
>
> - Nonlocal Variable is the variable that is defined in the `nested function`. It means the variable can be neither in the local scope nor in the global scope.
> - To create a nonlocal variable **`nonlocal`** keyword is used.
> - **Example:** In the following code, we created an outer function, and there is a `nested function inner()`. In the scope of `outer()` function `inner()` function is defined. If we change the nonlocal variable as defined in the inner() function, then changes are reflected in the outer function.

In [128…
```python
# explaination on non-local variable

def func_outer():           # defining outer function
    x = "local"                       # definng local variable
    def func_inner():       # defining inner function
        nonlocal x
        x = "nonlocal"                 # definng local variable
        print("inner:", x)    # accessing
    func_inner()
    print("outer:", x)
func_outer()
```

```
inner: nonlocal
outer: nonlocal
```

> **`iv) Built-in Scope:`**
>
> - If a Variable is not defined in local, Enclosed or global scope, then python looks for it in the built-in scope.
>
> **Example:** In the Following Example, 1 from math module pi is imported, and the value of pi is not defined in global, local and enclosed. Python then looks for the pi value in the built-in scope and prints the value. Hence the name which is already present in the built-in scope should not be used as an identifier.

In [130…
```python
# explaination  on built-in Scope

from math import pi
# pi = 'Not defined in global pi'
def func_outer():
    # pi = 'Not defined in outer pi'
    def inner():
        # pi = 'not defined in inner pi'
        print(pi)
    inner()

func_outer()
```

```
3.141592653589793
```

6. What happens to variables in a `local scope` when the function call `returns`?

`Ans-6`

> - A local variable retains its value until the next time the function is called. A local variable becomes undefined after the function call completes
> - When a function call returns, then the variables defined in its local scope are no longer accessible and their memory is released by the program.
> - However, if a variable is defined as a global variable, it will retain its value even after the function call returns.
> - Additionally, if a variable from the local scope is passed as an argument to another function, its value may be retained in that function's scope.

7. What is the concept of a `return value`? Is it possible to have a `return value` in an expression?

`Ans-7`

> **`return Value:`**
>
> - **`Def:`** A return is a value that a function returns to the calling function when it completes its task.
> - The Python `return statement` is a key component of `functions` and `methods`. A return statement consists of the `return keyword` followed by an optional `return value`. We can use the `return statement` to make your functions send Python objects back to the caller code. These objects are known as the function's **`return value`**.
> - Everything in Python is an `object`. So, our functions can return numeric values (int, float, and complex values), collections and sequences of objects (list, tuple, dictionary, or set objects), user-defined objects, classes, functions, and even modules or packages. The type of value your function returns depends largely on the task it performs.
> - We can omit the `return value` of a function and use a bare return without a return value. You can also omit the entire return statement. In both cases, the return value will be None. Using the return statement effectively is a core skill if you want to code custom functions that are Pythonic and robust.

In [88]:
```python
# Example-1: Explaination on return value

def return_value():       # defining a function

    return 42                     # returning simple int value

return_value()        # calling return_value function
```

Out[88]: 42

> **`return Value in Expression:`**
>
> - If you define a function with an explicit return statement that has an explicit return value, then you can use that return value in any expression.

In [87]:
```python
#  Example-2: Explaination on return value

def return_value():       # defining a function

    return 42 + 5                     # using return value in a expression

return_value()        # calling return_value function
```

Out[87]: 47

8. If a function does not have a `return statement`, what is the `return value` of a call to that function?

`Ans-8`

> - If you don't explicitly use a return value in a `return statement`, or if we totally omit the `return statement`, then Python will implicitly return a default value for us. That default return value will always be **`None`**.

> **`return statement`**
>
> - The Python `return statement` is a special statement that you can use inside a `function` or `method` to send the function's result back to the caller.
> - A return statement consists of the **`return keyword`** followed by an optional **`return value`**.
> - The `return value` of a Python function can be any Python `object`. Everything in Python is an object. So, your functions can return numeric values (int, float, and complex values), collections and sequences of objects (list, tuple, dictionary, or set objects), user-defined objects, classes, functions, and even modules or packages.
> - There are two types of `return statements` are as below:

```
i) Explicit return Statements,
ii) Implicit return Statements.
```

> **`i) Explicit return Statements:`**
>
> - An `explicit return statement` immediately terminates a function execution and sends the `return value` back to the caller code.
> - To add an explicit return statement to a Python function, you need to use `return` followed by an optional `return value`.
> - **`Syntax:`**

```
def function_name():
    '''docstring'''
    # function body
    return return_value
```

> - **Example:**

In [92]:
```python
# explaination of Explicit return statement
def hello():                              # defining function
    msg = "Hello World!"      # local variable with some massege

    return msg                            # explicitly returning  return value

hello()                                    # calling function
```

Out[92]: 'Hello World!'

> **`ii) Implicit return Statements:`**
>
> - A Python function will always have a `return value`. There is no notion of procedure or routine in Python.
> - So, if you don't `explicitly` use a `return value` in a return statement, or if you totally `omit` the `return statement`, then Python will `implicitly return` a default value for you. That default return value will always be `None`.
> - **Syntax**:

```
def function_name():
    '''docstring'''
    # function body
```

In [122…
```python
# explaination of Explicit return statement
def add(x):                              # defining function
    addition = x+2      # local variable with some massege

                                                  # NO return-statement used here

result=add(2)                                    # calling function

print(result)                          # it will return "None"
```

```
None
```

9. How do you make a `function variable` refer to the `global variable`?

`Ans-9`

> **`i) Local Variable:`**
>
> - The variables which are defined inside the function are known as a **`Local Variables/function variables`**.
> - We can access `local variables` in Python only `inside` the function.

> **`ii) Global Variable:`**
>
> - The variables which are defined outside the function are known as **`Global Variables`**.
> - We can access `global variables` in Python both `inside` and `outside` the function.

> **`Local variable as global variable:`**
>
> - If a function has a `local variable` and if we wanst to modify that local variable as a global variable inside function then we have to use **`global`** keyword before the variable name at start of function.
> - **Example:**

In [120…
```python
# function variable refer to the global variable

total = 100                   # defining global variable outside the function
def func():
    global total               # making to global variable 'total' inside function
    if total > 10:
        total = 15
        print(f"Printing Inside the Function: {total}")        # printing variable inside the function

func()

print(f"Printing Outside the Function: {total}")           #  printing local variable outside the functio
```

```
Printing Inside the Function: 15
Printing Outside the Function: 15
```

10. What is the data type of `None`?

`Ans-10`

> **`None:`**
>
> - The `None` keyword is used to define a null value, or no value at all.
> - The `None` is not the same as 0, False, or an empty string. None is a data type of its own (NoneType) and only None can be None.
> - Python uses the keyword `None` to define `null objects and variables`.
> - While `None` does serve some of the same purposes as null in other languages, it's another beast entirely. As the `null` in Python, `None` is not defined to be 0 or any other value.

> **`Data Type of None:`**
>
> - The datatype of `None` is **`NoneType`**.
> - None is a data type of its own (NoneType) and only None can be None.

11. What does the sentence `import areallyourpetsnamederic` do?

`Ans-11`

> - The sentence "import areallyourpetsnamederic" will import the module 'areallyourpetsnamederic'.

12. If you had a `bacon() feature` in a `spam module`, what would you call it after importing spam?

`Ans-12`

> - Let us assume, we have **`spam`** module which contains **`bacon()`** function/feature, then we can use it in following way:

In [ ]:
```python
# explaination
import spam                   # importing spam-module
spam.bacon()                  # calling bacon() feature of spam-module
```

13. What can you do to save a programme from crashing if it encounters an `error`?

`Ans-13`

> - Sometimes while executing a Python program, the program does not execute at all or the program executes but generates unexpected output or behaves abnormally. These occur when there are `syntax errors`, `runtime errors` or `logical errors` in the code.
> - `Error` in Python can be of two types i.e. `Syntax errors` and `Exceptions`. **`Errors`** are problems in a program due to which the program will stop the execution. On the other hand, **`exceptions`** are raised when some internal events occur which change the normal flow of the program.
> - Each and every exception has to be handled by the programmer to avoid the program from crashing abruptly. This is done by writing additional code in a program to give proper messages or instructions to the user on encountering an exception. This process is known as **`exception handling`**.
> - **Exceptions** are raised when the program encounters an error during its execution. They disrupt the normal flow of the program and usually end it abruptly. To avoid this, you can catch them and handle them appropriately using **`Try-Except-Statement.`**

14. What is the purpose of the `try clause`? What is the purpose of the `except clause`?

`Ans-14`

> - In Python, try and except are used to handle exceptions, which are errors detected during execution.
> - With try and except, even if an exception occurs, the process continues without terminating. Additionally, else and finally can be used to set the ending process.

> **`try:`**
>
> - The `try block` is used to check some code for errors i.e the code inside the `try block` will execute when there is no error in the program.

> **`except`**
>
> - The code in `except block` is only executed if an exception occured in the try block. The `except block` is required with a try block, even if it contains only the pass statement.

> - It may be combined with the `else` and `finally` keywords. **`else:`** Code in the else block is only executed if no exceptions were raised in the try block. **`finally:`** The code in the finally block is always executed, regardless of if a an exception was raised or not.