Assignment No. 8

Submitted By: Sachin Dodake

1. In Python, what is the difference between a `built-in function` and a `user-defined function`? Provide an example of each.

`Ans-1`

`Functions:`

- A function is a set of statements that take inputs, do some specific computation and produce output. 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 call the function.
- Functions that readily comes with Python are called `built-in functions`. We can also create your own functions, and these functions are known as `user defines functions`.

`i) Built-in-Functions:`

- The Built-in functions are already defined in python. A user has to remember the name and parameters of a particular function. Since these functions are pre-defined, there is no need to define them again.
- The Python interpreter has a number of functions that are always available for use. These functions are called built-in functions. For example, print() function prints the given object to the standard output device (screen) or to the text stream file.
- **Example:**

In [6]:
```
# Explaination of built-in-function

num_list = [2, 5, 19, 7, 43]

print(f"Length of string is: {len(num_list)}")                  # to check length of list
print(f"Maximum number in list is: {max(num_list)}")    # to check maximum num from list
print(f"Type is: {type(num_list)}")                                    # to check type of list
```

```
Length of string is: 5
Maximum number in list is: 43
Type is: <class 'list'>
```

`ii) User-in-Functions:`

- Functions that we define ourselves to do the certain specific task are referred to as user-defined functions. The way in which we define and call functions in Python are already discussed.
- Functions that readily come with Python are called built-in functions. If we use functions written by others in the form of the library, it can be termed as library functions.
- All the other functions that we write on our own fall under user-defined functions. So, our user-defined function could be a library function to someone else.
- **Example:**

In [10]:
```
# # Explaination of user-defined functions

def add_numbers(x,y):                 # defining a function
    '''This function will be used to add the two numbers'''      # docstring to explain working on function
    sum = x + y          #  defining a sum variable
    return sum       # returning value of  variable sum

num1 = 5
num2 = 6

print(f"The sum of '{num1}' & '{num2} is : {add_numbers(num1, num2)}")
```

```
The sum of '5' & '6 is : 11
```

2. How can you `pass arguments to a function` in Python? Explain the difference between `positional arguments` and `keyword arguments`.

`Ans-2:`

`Arguments:`

- When we define and call a Python function, the term `parameter` and `argument` is used to pass information to the function.

```
    Parameter: It is the variable listed inside the parentheses in the function
definition.
    Argument: It is a value sent to the function when it is called. It is data on which
function performs some action and returns the result.
```

- `Parameters` are inside functions or procedures, while `arguments` are used in procedure calls, i.e. the values passed to the function at run-time.
- It is not mandatory to use arguments in function definition. But if you need to process user data, you need arguments in the function definition to accept that data. Also, we use argument in function definition when we need to perform the same task multiple times with different data.
- If the function is defined with parameters, the arguments passed must match one of the arguments the function accepts when calling.
- There are various ways to use arguments in a function. In Python, we have the following 4 types of function arguments.

```
i) Default argument
ii) Keyword arguments (named arguments)
iii) Positional arguments
iv) Arbitrary arguments (variable-length arguments *args and **kwargs)
```

`Passing Arguements to Function:`

- We can pass multiple arguments to a python function by predetermining the formal parameters in the function definition.

`Positional arguments:`

- An argument is a variable, value or object passed to a function or method as input. Positional arguments are arguments that need to be included in the proper position or order.
- The first positional argument always needs to be listed first when the function is called. The second positional argument needs to be listed second and the third positional argument listed third, etc.
- **Syntax:**

```
function_name(value1, value2, value3,.....)
```

- **Example:** The best exapmple is python's `complex() function`. This function returns a complex number with a `real term` and `an imaginary term`. The order that numbers are passed to the `complex() function` determines which number is the `real term` and which number is the `imaginary term`. If the complex number 3 + 5j is created, the two positional arguments are the numbers 3 and 5. As positional arguments, 3 must be listed first, and 5 must be listed second.

In [19]:
```
def nameAge(name, age):
    print(f"Hi, I am {name}.")
    print(f"My age is {age}. ")

# We will get correct output because argument is given in order
print("Case-1:")
nameAge("Sachin", 20)

# We will get incorrect output because argument is not in order
print("\nCase-2:")
nameAge(30, "Vicky")
```

```
Case-1:
Hi, I am Sachin.
My age is 20.

Case-2:
Hi, I am 30.
My age is Vicky.
```

`Keyword arguments:`

- A keyword argument is an argument passed to a function or method which is preceded by a keyword and an equals sign.
- **Syntax:**

```
function_name(parameterName1 = value1, parameterName2 = value1)
```

- **Example:** Python's `complex() function` can also accept two `keyword arguments`. The two keyword arguments are real= and imag=. To create the complex number 3 + 5j the 3 and 5 can be passed to the function as the values assigned to the keyword arguments real= and imag=.

In [21]:
```
def nameAge(name, age):
    print(f"Hi, I am {name}.")
    print(f"My age is {age}. ")

print("Case-1:")        # there will o efect on output even if the order is different
nameAge(name="Sachin", age=20)

print("\nCase-2:")           # there will o efect on output even if the order is different
nameAge(age=30, name="Vicky")
```

```
Case-1:
Hi, I am Sachin.
My age is 20.

Case-2:
Hi, I am Vicky.
My age is 30.
```

3. What is the purpose of the return statement in a function? Can a function have multiple return statements? Explain with an example.

`Ans-3`

`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.
- 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`.
- **Example:** The examples with and without return statements are as below:

In [6]:
```
# Example on Function without return statement

def print_something(a):
    print(f"Printing: {a}")

output = print_something('Hello World !')

print(f"\nOutput without return statement:")
print(f"\tA function without return statement returns:  {output}")


# Example on Function without return statement

def add(x, y):
    result = x + y
    return result

output = add(5, 4)

print(f"\nOutput with return statement:")
print(f'\tThe result of `add(5, 4)` function is :  {output}')
```

```
Printing: Hello World !

Output without return statement:
        A function without return statement returns:  None

Output with return statement:
        The result of `add(5, 4)` function is :  9
```

`Multiple return-Statements:`

- Python functions are not restricted to having a `single return statement`. If a given function has more than one return statement, then the first one encountered will determine the end of the function's execution and also its return value.
- A common way of writing functions with multiple return statements is to use conditional statements that allow you to provide different return statements depending on the result of evaluating some conditions.
- **Example:** The example of multiple return statement is as below:

In [45]:
```
def type_of_int(num):
    if num % 2 == 0:
        return 'even'
    else:
        return 'odd'

result = type_of_int(7)

print(f"The given number is : {result}.")
```

```
The given number is : odd.
```

4. What are lambda functions in Python? How are they different from regular functions? Provide an example where a lambda function can be useful.

`Ans-4`

`Lambda Function:`

- A `lambda function` is an anonymous function (i.e., defined without a name) that can take any number of arguments but, unlike normal functions, evaluates and returns only one expression.
- Python `Lambda Functions` are anonymous function means that the function is without a name. As we already know that the def keyword is used to define a normal function in Python. Similarly, the lambda keyword is used to define an anonymous function in Python.
- **Syntax:**

```
lambda p1, p2: expression
            p1- parameter1 to pass in lambda function
            p2- parameter2 to pass in lambda function
```

- The anatomy of a lambda function includes three elements:

```
    The keyword lambda — an analog of def in normal functions
    The parameters — support passing positional and keyword arguments, just like
normal functions
    The body — the expression for given parameters being evaluated with the lambda
function
```

In [51]:
```
# Exaplaination on Lambda function to Multiply argument a with argument b and return the result:

multi = lambda a, b : a * b

print(f"The multiplication using `Lambda Function` is : {multi(5, 6)}")
```

```
The multiplication using `Lambda Function` is : 30
```

Regular Function-Vs-Lambda Function:

- `Syntax:` Lambda functions are written in a single line of code, whereas regular functions defined with def can span multiple lines.
- `Function Name:` Lambda functions do not have a name, whereas regular functions defined with def have a name.
- `Return Statement:` Lambda functions automatically return the result of the expression they evaluate, while regular functions defined with def require an explicit return statement to return a value.
- `Arguments:` Both types of functions can take any number of arguments, but lambda functions are typically used for simple, one-line expressions with one or two arguments.
- `Functionality:` Regular functions defined with def can include complex logic, including flow control statements (such as if and while), error handling, and more complex calculations. Lambda functions are typically used for simple operations, such as filtering, mapping, or reducing data.
- **Example:** The examples of `Lambda` & `Regular` function are as below:

In [49]:
```
#  Explaination of Lambda function that adds two numbers
add_lambda = lambda x, y: x + y

# Explaination of Regular function that adds two numbers
def add_def(x, y):
    return x + y

# Calling the functions
print(f"The result using 'Lambda' function is : {add_lambda(2, 3)}")
print(f"\nThe result using 'Regular' function is : {add_def(2, 3)}")
```

```
The result using 'Lambda' function is : 5

The result using 'Regular' function is : 5
```

5. How does the concept of "scope" apply to functions in Python? Explain the difference between local scope and global scope.

`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 [52]:
```
# 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 [52], in <cell line: 10>()
      6     print("Local variable value is:",num_2)           # accessing local variable num_2inside functio
n
      8 local_test()                        # calling function `local_test`
---> 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 [54]:
```
# 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
```

6. How can you use the "return" statement in a Python function to return multiple values?

`Ans-6`

- Python functions can return multiple values. These values can be stored in variables directly. A function is not restricted to return a variable, it can return zero, one, two or more values.
- For returning multiple values from a function, we can return tuple, list or dictionary object as per our requirement.
- In Python, we can return multiple values from a function in different ways as listed below:

```
i) Using Object,
ii) Using Tuple,
iii) Using a list,
iv) Using a Dictionary,
v) Using Data Class,
vi) Using generator 'yield'
```

`i) Using Object:`

- This is similar to C/C++ and Java, we can create a class (in C, struct) to hold multiple values and return an object of the class.
- **Example:**

In [8]:
```
# A Python program to return multiple values from a method using class
class Test:
    def __init__(self):
        self.str = "Python"
        self.x = 1111

# This function returns an object of Test
def fun():
    return Test()

# Driver code to test above method
t = fun()
print(t.str)
print(t.x)
```

```
Python
1111
```

`ii) Using Tuple:`

- A Tuple is a comma separated sequence of items. It is created with or without (). Tuples are immutable.
- **Example:**

In [9]:
```
# A Python program to return multiple values from a method using tuple

# This function returns a tuple
def fun():
    str = "Python"
    x = 1111
    return str, x # Return tuple, we could also write (str, x)

# Driver code to test above method
str, x = fun() # Assign returned tuple
print(str)
print(x)
```

```
Python
1111
```

`iii) Using List:`

- A list is like an array of items created using square brackets. They are different from arrays as they can contain items of different types. Lists are different from tuples as they are mutable.
- **Example:**

In [11]:
```
# A Python program to return multiple values from a method using list

# This function returns a list
def fun():
    str = "Python"
    x = 1111
    return [str, x]

# Driver code to test above method
list = fun()
print(list)
```

```
['Python', 1111]
```

`iv) Using Dictionary:`

- A Dictionary is similar to hash or map in other languages. See this for details of dictionary.
- **Example:**

In [13]:
```
# A Python program to return multiple values from a method using dictionary

# This function returns a dictionary
def fun():
    d = dict();
    d['str'] = "Python"
    d['x'] = 1111
    return d

# Driver code to test above method
d = fun()
print(d)
```

```
{'str': 'Python', 'x': 1111}
```

`v) Using Data Class:`

- In Python 3.7 and above the Data Class can be used to return a class with automatically added unique methods. The Data Class module has a decorator and functions for automatically adding generated special methods such as **init**() and **repr**() in the user-defined classes.
- **Example:**

In [24]:
```
from dataclasses import dataclass

@dataclass
class Book_list:
    name: str
    perunit_cost: float
    quantity_available: int = 0

    # function to calculate total cost
    def total_cost(self) -> float:
        return self.perunit_cost * self.quantity_available

book = Book_list("Introduction to Python.", 1000, 5)
x = book.total_cost()

# print the total cost of the book
print(x)

# print book details
print(book)

# 900
Book_list(name='Python Programming.',
        perunit_cost=2000,
        quantity_available=5)
```

```
5000
Book_list(name='Introduction to Python.', perunit_cost=1000, quantity_available=5)
```

Out[24]:
```
Book_list(name='Python Programming.', perunit_cost=2000, quantity_available=5)
```

`vi) Using Using generator 'yield':`

- One alternative approach for returning multiple values from a function in Python is to use the yield keyword in a generator function. A generator function is a special type of function that returns an iterator object, which generates a sequence of values on the fly, one value at a time.
- To return multiple values from a generator function, you can use the yield keyword to yield each value in turn. The generator function will then pause execution until the next value is requested, at which point it will resume execution and yield the next value. This process continues until the generator function completes execution or encounters a return statement.
- **Example:**

In [25]:
```
def get_values():
    yield 42
    yield 'hello'
    yield [1, 2, 3]

# Test code
result = get_values()
print(next(result)) # should print 42
print(next(result)) # should print 'hello'
print(next(result)) # should print [1, 2, 3]
```

```
42
hello
[1, 2, 3]
```

7. What is the difference between the "pass by value" and "pass by reference" concepts when it comes to function arguments in Python?

`Ans-7`

- The main difference between pass by value and pass by reference is that, in a pass by value, the parameter value copies to another variable while, in a pass by reference, the actual parameter passes to the function.
- When we define a custom or user defined function in Python we may, optionally we may need to specify parameter names between the function's parentheses.
- If we specify parameters in the function definition, then we need to pass argument values to the function's parameters while calling it.
- The function use that passed values during its execution by referencing it via parameter name.
- The `Callee` is a function called by another and the `caller` is a function that calls another function (the callee).

```
    The values that are passed in the function call are called the actual parameters.
    The values received by the function (when it is called ) are called the formal
parameters.
```

- Basically, there are two ways to pass argument values to the function's parameters. They are:

```
i) Call/Pass by Value
ii) Call/Pass by Reference
```

`i) Pass by Value:`

- In pass by value (also known as call by value), the argument passed to the function is the copy of of its original object. If we change or update the value of object inside the function, then original object will not change.
- If the argument is variable, the copy of the current value of the variable is passed to the function's parameter. The value of the variable in the function call is not affected by what happens inside the function.
- Some of the programming languages like C++, Java uses call by value or pass by value concept to pass argument values to the function's parameters.
- If the argument is passed by value, a copy of it is made and passed to the function. If the argument values being passed to the function is large, the copying can take up a lot of time and memory.
- That's why `Python language does not support the pass by value` concept.

`ii) Pass by Reference:`

- In pass by reference (also known as call by reference), the argument passed to the function's parameter is the original object. If we change the value of object inside the function, the original object will also change.
- The scripting language like JavaScript uses the pass by reference mechanism to pass an object as an argument to the function.
- That's why `Python language does not support the pass by reference` mechanism to pass argument to the function.

8. Create a function that can intake integer or decimal value and do following operations:

```
a. Logarithmic function (log x)
b. Exponential function (exp(x))
c. Power function with base 2 (2x)
d. Square root
```

`Ans-8`

In [26]:
```
# Logarithmic function (log x)

import math
def log_num(x, b):
    log_result  = math.log(x) // math.log(b)         # x = given number, b=logarithmic base
    return log_result

x = 100
b = 3

print(f"The logarithmic value of `{x}` with base `{b}` is :  {int(log_num(x, b))}")
```

```
The logarithmic value of `100` with base `3` is :  4
```

In [27]:
```
# Exponential function (exp(x))

import math
def exp_num(x):
    exp_result  = math.exp( x )         # x = given number
    return exp_result

x = 10

print(f"The exponential value of `{x}` is :  {int(exp_num(x))}")
```

```
The exponential value of `10` is :  22026
```

In [28]:
```
# Power function with base 2 (2x)

import math
def pow_num(x):
    pow_result  = math.pow(x, y)         # x = base, y=exponent
    return pow_result

x = 2
y=2

print(f"The powe of `{x}` with base `{x}` is :  {int(pow_num(x))}")
```

```
The powe of `2` with base `2` is :  4
```

In [29]:
```python
# Square root

import math
def sqrt_num(x):
    sqrt_result  = math.sqrt(x)          # x = number
    return sqrt_result

x = 25

print(f"The square root  of `{x}` is :  {int(sqrt_num(x))}")
```

```
The square root  of `25` is :  5
```

9. Create a function that takes a full name as an argument and returns first name and last name

Ans-9

In [46]:
```python
def full_name(name):
    f_name = name.split()[0]
    l_name  = name.split()[1]
    name_list = [f_name, l_name]

    return name_list[0], name_list[1]

my_name = full_name('Sachin Dodake')

print(f"The FIRST name is: {my_name[0]}")
print(f"The LAST name is: {my_name[1]}")
```

```
The FIRST name is: Vikas
The LAST name is: Tejas
```

In [ ]: