Assignment No-8

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

Solution: In Python, a built-in function is a function that is pre-defined in the Python language and is readily available for use without requiring any additional code or modules. These functions are part of the Python standard library and cover a wide range of functionalities. Examples of built-in functions include `print()`, `len()`, `input()`, `range()`, etc.

A user-defined function, on the other hand, is a function created by the user to perform specific tasks or calculations that are not available as built-in functions. These functions are defined by the user using the `def` keyword followed by a function name and a block of code that specifies what the function should do. Examples of user-defined functions could be:

```
def calculate_area(length, width):
    area = length * width
    return area

def check_prime(number):
    if number < 2:
        return False
    for i in range (2, int(number ** 0.5) + 1):
        if number % i == 0:
        return False
    return True
```

In the above examples, `calculate_area()` is a user-defined function that calculates the area of a rectangle given its length and width, while `check_prime()` is a user-defined function that checks if a number is prime or not.

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

Solution: In Python, you can pass arguments to a function in two ways: positional arguments and keyword arguments.

Positional arguments are passed to a function by their position in the function call. The order in which they are written in the function call must match the order in which the function parameters are defined. For example:

```
def multiply (a, b):
    return a * b
result = multiply (2, 3)
print(result) # Output: 6
```

In this example, the arguments 2 and 3 are passed to the `multiply` function as positional arguments. The `a` parameter inside the function will have the value 2, and the `b` parameter will have the value 3.

Keyword arguments, on the other hand, are passed to a function by explicitly specifying the parameter name and corresponding value in the function call. The order in which they are written does not matter. For example:

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

result = add (a=2, b=3)
print(result) # Output: 5
```

In this example, the arguments 2 and 3 are passed to the `add` function as keyword arguments. The function's parameter names `a` and `b` are matched with the corresponding keyword arguments.

Keyword arguments can also be passed in any order:

```
result = add (b=3, a=2)
print(result) # Output: 5
```

we can even mix positional and keyword arguments in a function call. However, all positional arguments must be specified before any keyword arguments:

```
result = add (2, b=3)
print(result) # Output: 5
```

In summary, positional arguments are passed to a function based on their position in the function call, whereas keyword arguments are passed by explicitly specifying the parameter name and corresponding value.

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

Solution: The purpose of the return statement in a function is to specify the value that the function should "return" or output when it has finished executing. It allows the function to "send back" a value or result to the code that called the function.

Yes, a function can have multiple return statements. The function can have conditional logic to determine which return statement to execute based on certain conditions. Only one return statement will be executed and the function will terminate at that point.

For example, consider a function that checks if a number is positive, negative, or zero:

```
def check_number(num):
   if num > 0:
      return "Positive"
   elif num < 0:
      return "Negative"
   else:
      return "Zero"
```

In this example, the function takes a number as an argument and checks if it is positive, negative, or zero. It uses multiple return statements to return different strings based on the condition. If the number is positive, the first return statement is executed and the function terminates. If the number is negative, the second return statement is executed and the function terminates. If the number is zero, the third return statement is executed and the function terminates.

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

Solution: Lambda functions, also known as anonymous functions, are small and anonymous functions in Python. They are defined without a name and are created using the lambda keyword. Lambda functions can take any number of arguments, but can only have one expression.

The main difference between lambda functions and regular functions is that lambda functions are used when a small, one-time function is needed, and it does not require a separate definition. Regular functions are defined using the def keyword and can have multiple expressions.

Here is an example to show the difference between a regular function and a lambda function:

Regular function:
```
def square(num):
    return num ** 2

print(square(5))  # Output: 25
```

Lambda function:
```
square = lambda num: num ** 2

print(square(5))  # Output: 25
```

In this example, both the regular function and lambda function compute the square of a number. The lambda function is more concise and does not require the use of the def keyword or a separate name. It can be used in situations where a small, one-time function is needed, such as passing a function to another function or using it as an argument in a higher-order function.

For instance, lambda functions can be useful when you want to sort a list of numbers based on specific criteria. Here's an example where a lambda function is used with the `sorted()` function to sort a list of tuples based on the second element:

```
numbers = [(3, 9), (2, 5), (4, 0), (1, 7)]
sorted_numbers = sorted(numbers, key=lambda x: x[1])

print(sorted_numbers)  # Output: [(4, 0), (2, 5), (1, 7), (3, 9)]
```

In this example, the lambda function `lambda x: x[1]` specifies that the sorting should be based on the second element of each tuple.

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

Solution: In Python, the "scope" of a variable or function defines where it can be accessed and used within the code. The concept of scope becomes particularly important when dealing with functions.

A "local scope" refers to the variables and functions that are defined within a specific function. These variables and functions are only accessible from within that function. They cannot be accessed or used outside of the function. The local scope is created when the function is called and destroyed when the function completes its execution.

For example, consider the following code:

```
def my_function():
   x = 10
   print(x)

my_function()
```

In this code, the variable `x` is defined within the `my_function()` function. It has a local scope and can only be accessed within that function. The output of the code will be `10` because `x` is printed within the function.

On the other hand, a "global scope" refers to the variables and functions that are defined outside of any specific function. These variables and functions are accessible from anywhere within the code, including inside functions.

For example, consider the following code:

```
x = 10

def my_function():
   print(x)

my_function()
```

In this code, the variable `x` is defined outside of any function, making it a global variable. It can be accessed and used within the `my_function()` function as well as any other part of the code. The output of the code will be `10`.

It is important to note that if a variable is defined as both a local variable and a global variable, the local variable takes precedence within the local scope.

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

Solution: In Python, you can use a tuple to return multiple values from a function. You can separate the values using commas and enclose them in parentheses to create a tuple. Then, you can use the `return` statement to return the tuple.

Here's an example:

```
def return_multiple_values():
    value1 = 10
    value2 = 'Hello'
    value3 = [1, 2, 3]
    return value1, value2, value3

result = return_multiple_values()
print(result) # Output: (10, 'Hello', [1, 2, 3])

# You can also unpack the returned tuple into separate variables:
val1, val2, val3 = return_multiple_values()
print(val1) # Output: 10
print(val2) # Output: 'Hello'
print(val3) # Output: [1, 2, 3]
```

In this example, the function `return_multiple_values()` returns three values: an integer, a string, and a list. These values are packed into a tuple `(value1, value2, value3)` and returned using the `return` statement. You can then assign the returned tuple to a variable and access individual values using indexing or unpack the returned tuple into separate variables.

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

Solution: In Python, the concepts of "pass by value" and "pass by reference" are slightly different due to how objects are handled in the language. Python uses a combination of both concepts, which can be a bit confusing.

In "pass by value," the value of the variable (or argument) itself is copied and passed to the function. Any modifications made to the variable inside the function do not affect the original variable outside the function. This is the case for immutable objects in Python, such as numbers and strings. When you pass an immutable object to a function in Python, it behaves like a "pass-by value."

Example:

```
def change_value(num):
num = 10
value = 5
change_value(value)
print(value) # Output: 5
```

In the above example, the `change_value()` function receives a copy of the value of `value`, which is 5. It modifies the copied value to 10 inside the function, but the original variable `value` remains unchanged because the function operates on a copy.

On the other hand, in "pass by reference," the memory address (or reference)

8. Create a function that can intake integer or decimal values and do the following operations: a. Logarithmic function (log x)
b. Exponential function (exp(x))
c. Power function with base 2 (2^x)
d. square root

Solution: Here's a Python function that can handle the requested operations:

```
import math
def math_operations(value):
    # Logarithmic function (log x)
    logarithm = round(math.log(value), 2)
    # Exponential function (exp(x))
    exponential = round(math.exp(value), 2)
    # Power function with base 2 (2^x)
    power = round(math.pow(2, value), 2)
    # Square root
    square_root = round(math.sqrt(value), 2)
    return logarithm, exponential, power, square_root
```

To use this function, simply pass an integer or decimal value to it:

```
result = math_operations(3.7)
print(result)
```

Output:

```
(1.31, 40.41, 13.37, 1.92)
```

In this example, the value 3.7 is passed to the function, and the output is a tuple containing the input value's logarithm, exponential, power, and square root (rounded to 2 decimal places).

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

Solution:
```
def split_name(full_name):
    first_name, last_name = full_name.split(" ", 1)
    return first_name, last_name
# Example usage:
name = "John Doe"
first_name, last_name = split_name(name)
print("First name:", first_name)
print("Last name:", last_name)
```