**Python Assignment 3**

Q1. Why are functions advantageous to have in your programs?

Ans. Functions are advantageous to have in programs for several reasons:

* **Modularity**: - Functions allow you to break your code into smaller, more manageable pieces. This makes it easier to read and understand, and also makes it easier to reuse code in other parts of your program or in other programs.
* **Abstraction**: - Functions allow you to hide the implementation details of a piece of code behind an interface. This means that you can use the function without needing to know how it works internally. This makes your code more modular and easier to reason about.
* **Code reus**e: - Functions can be reused in multiple parts of a program or in multiple programs. This saves time and effort because you don't need to rewrite the same code multiple times.
* **Testing**: - Functions make it easier to test your code because you can test each function in isolation. This means that if a function is not working correctly, you can isolate the problem and fix it without affecting the rest of the program.
* **Readability**: - Functions make your code more readable by giving meaningful names to blocks of code. This makes it easier to understand what the code does and how it fits into the overall program.

Overall, functions help make your code more organized, modular, reusable, testable, and readable, which in turn makes it easier to develop, maintain, and scale your programs.

Q2. When does the code in a function run: when it's specified or when it's called?

Ans. The code in a function runs when the function is called.

When you define a function, you are essentially creating a blueprint for a block of code that can be executed later. The code inside the function is not executed at the time of definition, but rather when the function is called.

When you call a function, the program jumps to the code inside the function, executes it, and then returns to the point where the function was called.

For example, consider the following function definition:

def greet(name):

print("Hello, " + name + "!")

The code inside the function, print("Hello, " + name + "!"), will not run until the function is called.

When you call the function like this:

greet("Vipin")

The output will be:

Hello, Vipin!

The code inside the function is executed only when the function is called with an argument.

Q3. What statement creates a function?

Ans. In Python, the def statement is used to create a function.

The basic syntax of a function definition is as follows:

def function\_name(parameters):

"""docstring"""

# code block

return value

Here, def is the keyword used to define a function, function\_name is the name you give to the function, parameters are the input arguments that the function accepts (optional), and docstring is an optional string that provides a brief description of what the function does.

The code block following the function definition contains the statements that are executed when the function is called. The return statement is optional, but if present, it specifies the value that the function returns when it is called.

For example, consider the following function definition:

def square(x):

"""Returns the square of a number."""

return x \*\* 2

This creates a function named square that takes one input argument, x, and returns its square.

Q4. What is the difference between a function and a function call?

Ans. A function is a block of code that performs a specific task and can be reused throughout a program. It is defined using the def keyword followed by a unique function name and any input parameters the function requires. The code inside the function is executed only when the function is called.

A function call, on the other hand, is the code that requests the execution of a particular function with specific arguments. A function call typically consists of the function name followed by parentheses, which may include any arguments that the function requires. When a function is called, the program jumps to the code inside the function, executes it, and then returns to the point where the function was called.

Here is an example that demonstrates the difference between a function and a function call:

# function definition

def add\_numbers(x, y):

"""Adds two numbers and returns the result."""

result = x + y

return result

# function call

sum = add\_numbers(3, 5)

print(sum) # output: 8

In this example, the add\_numbers function is defined with two parameters x and y. When called with the arguments 3 and 5, it returns the sum of these two numbers. The function call add\_numbers(3, 5) is where the function is actually executed, and the result is stored in the variable sum. The output of the program is 8, which is the result of the function call.

Q5. How many global scopes are there in a Python program? How many local scopes?

Ans. In a Python program, there is only one global scope, which is created when the program starts running. Any variable defined outside of a function or class is in the global scope and can be accessed from anywhere in the program.

On the other hand, there can be multiple local scopes in a Python program. A local scope is created whenever a function or method is called. Any variable defined inside a function or method is in the local scope and can only be accessed within the function or method where it is defined, unless it is explicitly declared as a global variable.

In addition to local and global scopes, Python also has nonlocal scopes, which are used to access variables in the enclosing scope of a nested function. However, nonlocal scopes are less commonly used than global and local scopes.

Q6. What happens to variables in a local scope when the function call returns?

Ans. When a function call returns, the local scope of the function is destroyed, and any variables defined within that scope are removed from memory. This means that the values assigned to those variables are no longer accessible or usable outside of the function.

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 local scope of the my\_function() function. When the function is called, the value of x (which is 10) is printed to the console. However, once the function call returns, the local scope of the function is destroyed and the variable x is no longer accessible. If you were to try to access x outside of the function, you would get a NameError because the variable is undefined.

def my\_function():

x = 10

print(x)

my\_function()

# This will raise a NameError because x is not defined

print(x)

It's worth noting that if a variable defined within a local scope is returned by the function, it can still be used outside of the function because it becomes part of the scope of the code that called the function. However, any other variables defined within the local scope of the function will still be destroyed when the function call returns.

Q7. What is the concept of a return value? Is it possible to have a return value in an expression?

Ans. In programming, a return value is the value that a function sends back to the code that called it. The return value is a way for the function to provide information to the calling code about the result of its execution.

When a function is called, it may perform some operations and then return a value. The returned value can then be used by the calling code to perform additional operations or make decisions. For example, a function that calculates the sum of two numbers might return the sum as its return value, which could then be used in other calculations.

Return values can be any type of data, including numbers, strings, booleans, lists, or even other functions.

It is possible to have a return value in an expression. This means that the return value of a function can be used as part of a larger expression. For example, consider the following code:

def square(x):

return x \* x

result = square(3) + square(4)

print(result)

In this code, the square() function returns the square of its input value. The return value of square(3) is 9, and the return value of square(4) is 16. These values are then added together using the + operator and assigned to the variable result. The final output of the code is 25, which is the sum of the squares of 3 and 4.

So, yes, return values can be used in expressions just like any other value or variable.

Q8. If a function does not have a return statement, what is the return value of a call to that function?

Ans. If a function does not have a return statement, the return value of a call to that function is None.

In Python, a function without a return statement automatically returns None by default. This means that if you call a function that doesn't explicitly return a value, the function will still return None, and you can assign that value to a variable or use it in an expression just like any other value.

For example, consider the following code:

def greet(name):

print("Hello, " + name + "!")

result = greet("Vipin")

print(result)

In this code, the greet() function takes a name as input, prints a greeting to the console, but does not have a return statement. When the function is called with the name "Vipin", the output will be "Hello, Vipin!". However, since the function does not return a value, the variable result will be assigned the value None. When the value of result is printed, the output will be None.

It's important to keep in mind that even though the return value of a function without a return statement is None, this does not mean that the function didn't perform any useful operations. The function may have modified variables or performed some other action that doesn't require a return value.

Q9. How do you make a function variable refer to the global variable?

Ans. In Python, if you want to use a global variable inside a function, you can use the global keyword to indicate that the variable you're using should refer to the global variable instead of creating a new local variable with the same name.

Here's an example:

x = 10 # global variable

def my\_function():

global x # use the global variable 'x'

x = 5 # modify the global variable 'x'

print(x)

my\_function() # prints 5

print(x) # prints 5 (the global variable has been modified)

In this code, the my\_function() function uses the global keyword to indicate that the x variable it is using should refer to the global variable with the same name, rather than creating a new local variable. Then, it modifies the value of the global x variable by assigning it a new value of 5. Finally, it prints the value of x, which is now 5.

Note that using global variables inside functions can make it harder to understand and reason about your code, since the function may modify the global variable in unexpected ways. It's generally better to avoid using global variables as much as possible and instead pass variables as parameters to functions or return values from functions.

Q10. What is the data type of None?

Ans. In Python, None is a special value that represents the absence of a value or a null value. It is often used to indicate that a function does not have a return value or that a variable has not been initialized.

None is a built-in constant of the NoneType data type, which is a singleton object that represents the absence of a value. The NoneType data type has only one value, which is None.

You can check the type of None using the type() function:

>>> type(None)

<class 'NoneType'>

Note that None is not the same as False or 0. None is a distinct value that represents the absence of a value, while False and 0 are values that represent logical falseness and numerical zero, respectively.

Q11. What does the sentence import areallyourpetsnamederic do?

Ans. The sentence import areallyourpetsnamederic does not have any built-in meaning in Python. It is simply a statement that attempts to import a module named areallyourpetsnamederic.

Python's import statement is used to import modules, which are files that contain Python code and define classes, functions, and variables that can be used in other parts of a program. When you import a module, Python searches for a file with the same name as the module and executes the code in that file.

However, there is no built-in module named areallyourpetsnamederic in Python, so attempting to import it will result in a ModuleNotFoundError exception:

>>> import areallyourpetsnamederic

ModuleNotFoundError: No module named 'areallyourpetsnamederic'

In general, it's good practice to only import modules that you actually need in your program, and to give your modules meaningful and descriptive names that reflect their purpose. Using a module name like areallyourpetsnamederic is not recommended, as it is unlikely to provide any useful functionality and may confuse other developers who read your code.

Q12. If you had a bacon() feature in a spam module, what would you call it after importing spam?

Ans. If you have a function named bacon() in a module named spam, and you import the spam module into your Python program, you can call the bacon() function using the dot notation to reference the function within the spam module.

Here's an example:

import spam

spam.bacon()

In this code, the spam module is imported using the import statement, and then the bacon() function within the spam module is called using the dot notation spam.bacon(). This will execute the code within the bacon() function.

If you want to avoid using the spam prefix every time you call the bacon() function, you can use the from ... import ... statement to import the bacon() function directly into your program's namespace:

from spam import bacon

bacon()

In this code, only the bacon() function is imported into the program's namespace using the from spam import bacon statement, so you can call the function directly without needing to use the spam prefix.

Q13. What can you do to save a programme from crashing if it encounters an error?

Ans. When a Python program encounters an error, it will often raise an exception and terminate the program. However, you can use exception handling to catch and handle exceptions, which can help prevent your program from crashing and provide more useful error messages to the user.

Here are some techniques you can use to save a program from crashing if it encounters an error:

Use try-except blocks: You can use the try-except statement to catch exceptions and handle them gracefully. In a try block, you can put the code that might raise an exception, and in an except block, you can specify how to handle the exception if it occurs.

Use if statements: You can use if statements to check for conditions that might cause errors before running code that might raise an exception. For example, you can check if a file exists before trying to read from it, or check if a variable is None before trying to use it.

Use default values: If a function can't perform its intended operation due to an error, you can return a default value instead of raising an exception. This can help prevent the program from crashing and provide a useful output to the user.

Use logging: You can use the Python logging module to log error messages and other information that can help you diagnose and fix problems in your program.

Here's an example of using a try-except block to catch an exception:

try:

x = int(input("Enter a number: "))

y = 10 / x

except ZeroDivisionError:

print("Error: division by zero")

except ValueError:

print("Error: invalid input")

else:

print("Result:", y)

In this code, we use a try block to attempt to read a number from the user and divide 10 by that number. If an exception occurs, we catch it in an except block and print an error message. If no exception occurs, we print the result.

Q14. What is the purpose of the try clause? What is the purpose of the except clause?

Ans. The try and except statements in Python are used for exception handling, which is a way to catch and handle errors that might occur during program execution. The try block contains the code that might raise an exception, and the except block contains the code that handles the exception if it occurs.

The purpose of the try clause is to enclose the code that might raise an exception. When the code in the try block encounters an error, it raises an exception, which Python can then catch and handle.

The except clause specifies how to handle the exception if it occurs. If an exception is raised in the try block, Python searches for an except block that matches the type of the exception, and executes the code in that block. If there is no matching except block, the exception is propagated up to the calling code, which can then handle it or let it propagate further up the call stack.

Here's an example of using a try and except block to catch a ValueError exception that might occur when converting user input to an integer:

try:

x = int(input("Enter a number: "))

except ValueError:

print("Error: invalid input")

else:

print("You entered:", x)

In this code, the try block attempts to convert user input to an integer using the int() function. If the user inputs a value that can't be converted to an integer, such as a string or a floating-point number, a ValueError exception is raised. The except block catches the exception and prints an error message, while the else block prints the integer value if no exception is raised.