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

Advantages of functions:

* Code organization: Functions allow you to break down a program into smaller, manageable chunks of code. By encapsulating specific functionality within functions, you can organize and structure your codebase more effectively. This modularity makes your code easier to understand, debug, and maintain.
* Code Reusability: Functions promote code reuse. Once you define a function, you can call it multiple times from different parts of your program without having to rewrite the same code. This saves time and effort and reduces the chances of introducing errors.
* Abstraction: Functions help in abstracting complex operations. You can encapsulate a complex set of instructions within a function and provide a simpler interface for other parts of the program to interact with. This abstraction hides the implementation details, allowing other programmers to use the function without needing to understand its internal workings.
* Readability: Functions enhance the readability of your code. By giving meaningful names to functions, you can convey the purpose and functionality of the code in a more self-explanatory manner. This makes it easier for other programmers (including your future self) to understand and maintain the code.
* Customization: User-defined functions allow programmers to tailor their programs to specific requirements. By defining functions that perform specific tasks, programmers can design their programs to address unique needs and solve specific problems.

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

The function in a code runs when it is called. When you define a function, you provide the code that should be executed when the function is called. This code is often referred to as the function's body. However, the code inside the function's body is not executed immediately upon definition. It only runs when the function is explicitly called or invoked in your program.

def greet():

print("Hello, World!")

#Function defined, but code inside is not executed yet

greet() #Function call

3. What statement creates a function?

def function\_name(argument):

#This is the general syntax used

#The syntax is followed by a codeblock that provides instructions on how to execute function when called.

* def: This keyword is used to indicate the start of a function definition.
* function\_name: This is the name you choose for your function. It should be descriptive and follow naming conventions.
* argument: These are optional input variables that the function can accept. They are specified inside parentheses after the function name. If there are no parameters, empty parentheses are still required.
* ‘:’: The colon symbol marks the end of the function header and indicates the start of the function body.
* Function body: This is the code block where you define the operations and functionality of the function. It can contain any valid statements and can span multiple lines.
* return statement (optional): You can use the return statement to specify the value that the function should return. It is optional and can appear anywhere within the function body. If no return statement is used, the function returns None by default.

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

Function: defined using the def statement in most programming languages. They are like predefined procedures or subroutines that can be invoked or called from different parts of a program.

Function Call: A function call, also known as a function invocation, is the act of executing or using a function within your code. When you make a function call, you are instructing the program to jump to the function's body, execute the code inside, and potentially receive a return value. A function call is typically made by using the function name followed by parentheses ().

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

Global scope: It refers to the outermost level of the program, where variables and functions are defined outside of any function or class. The global scope is accessible from anywhere within the program. Variables and functions defined in the global scope can be accessed and modified throughout the program.

Local scope: It is created whenever a function or a block of code is executed. Each function has its own local scope, which is created when the function is called and destroyed when the function completes execution. Variables and parameters defined within a function are local to that function and cannot be accessed outside of it, unless explicitly returned or referenced.

Eg: def myfunc():  
  x = 300  
  print(x)  
  
 myfunc()

The variable x is not available outside the function, but it is available for any function inside the function. Hence, it is available in the local scope of that function.

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

When a function call returns, the local variables defined within that function's scope are typically destroyed, and their memory is released. This process is known as "variable scope" or "variable lifetime."

When a function is called, a new local scope is created for that function. Any variables declared within the function body are allocated memory within this local scope. These variables are only accessible within the function and cannot be directly accessed from outside the function or from other functions.

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

A return statement is used to end the execution of the function call and “returns” the result (value of the expression following the return keyword) to the caller. The statements after the return statements are not executed.

It is possible to have a return value in an expression. The return value of a function can be used directly in expressions, assigned to variables, passed as arguments to other functions, or used in any way that a value of that type would typically be used.

Eg:

def square(x):

return x \*\* 2

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

The return value of a call to that function is ‘None’ when a function has not specified a return statement.

In Python, ‘None’ is a special value that represents the absence of a value or the lack of a return value. It is often used to indicate that a function does not produce a meaningful result or when the result is not needed for the specific context.

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

We can use the ‘global’ keyword.

global\_var = 10 # Global variable

def my\_function():

global global\_var # Declare the variable as global within the function

global\_var += 5 # Modify the global variable

my\_function()

print(global\_var) # Output: 15

10. What is the data type of None?

The None keyword is used to define a null value, or no value at all. 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.

11. What does the sentence import areallyourpetsnamederic do?

The sentence "import areallyourpetsnamederic" does not have any specific meaning or functionality in Python. It is not a valid import statement and would result in a ModuleNotFoundError unless there is a custom module named areallyourpetsnamederic available.

In Python, the import statement is used to bring functionality from other modules into the current module. Valid import statements typically reference existing module names, such as built-in modules (e.g., math, random) or external libraries installed via pip (e.g., numpy, pandas).

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

import spam

spam.bacon()

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

Assertions and try-except-finally are error handling mechanisms in Python to save a programme if it encounters an error.

1. try-except-finally block

try:

x = int(input("Please enter a number: "))

y = 100/x

except ValueError:

print("Error: there was an error")

except ZeroDivisionError:

print("Error: 0 is an invalid number")

except Exception:

print("Error: another error occurred")

finally:

print("Cleanup can go here")

* 1. The statement inside the try block is executed.
  2. If the statement is successful, both except clauses are skipped and the code inside the finally clause is run.
  3. If the statement inside the try block fails, the code in the first except statement is executed. If the statement fails due to a ValueError (i.e. not being able to convert a non-digit to an int), the code in the except ValueError block is run.
  4. If the statement inside the try block fails and the error is not a ValueError, the second except statement is checked. If the statement fails due to a ZeroDivisionError (i.e. integer is being divided by zero), the code inside the except ZeroDivisionError block is run.
  5. The finally clause will always execute after the last task completes — regardless of whether the last task is in the try block or except block.

1. Assertions evaluate an expression to true or false. If the expression is false, python will raise an AssertionError exception. Assertions can serve as a powerful developer tool when testing your code. The syntax for assertions is assert Expression[, Arguments]:

def divide(a, b):

assert b != 0, "Cannot divide by zero"

return a / b

result = divide(10, 0) #output displays assertion error

Exceptions should be used when you want to gracefully exit a program, log data, and notify the user of why such an error occurred.

Assertions have a fail-fast approach and should be used to find errors in your code and detect bugs.

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

try-except-finally block

try:

x = int(input("Please enter a number: "))

y = 100/x

except ValueError:

print("Error: there was an error")

except ZeroDivisionError:

print("Error: 0 is an invalid number")

except Exception:

print("Error: another error occurred")

finally:

print("Cleanup can go here")

* 1. The statement inside the try block is executed.
  2. If the statement is successful, both except clauses are skipped and the code inside the finally clause is run.
  3. If the statement inside the try block fails, the code in the first except statement is executed. If the statement fails due to a ValueError (i.e. not being able to convert a non-digit to an int), the code in the except ValueError block is run.
  4. If the statement inside the try block fails and the error is not a ValueError, the second except statement is checked. If the statement fails due to a ZeroDivisionError (i.e. integer is being divided by zero), the code inside the except ZeroDivisionError block is run.
  5. The finally clause will always execute after the last task completes — regardless of whether the last task is in the try block or except block.