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

Answer: Functions are advantageous in programming for several reasons:
Modularity: Functions allow you to break down a program into smaller, manageable parts. Each function can perform a specific task, making the code easier to understand, maintain, and debug.
Reusability: Once you define a function, you can use it multiple times throughout your program without having to rewrite the same code. This promotes code reuse and helps in avoiding redundancy.
Abstraction: Functions abstract away the implementation details of a certain task, allowing you to focus on using the function rather than understanding how it works internally. This abstraction makes the code more readable and understandable.
Encapsulation: Functions encapsulate a set of operations, meaning they can hide the details of how a task is performed. This can be particularly useful when working in teams, as it allows different team members to work on different functions independently without needing to understand the entire program.
Testing and Debugging: Functions make it easier to test and debug code because you can isolate specific parts of the program and test them individually. This makes it easier to identify and fix errors.
Scalability: As your program grows in complexity, functions allow you to organize and manage the codebase more effectively. You can add new functionality by simply defining new functions without having to modify existing code extensively.

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

Answer: The code inside a function runs when the function is called, not when the function is defined. Defining a function simply tells the program what tasks to perform when the function is eventually called. During the definition phase, you establish the function's structure, parameters, and the code block that executes when the function is invoked.
Here's a basic example to illustrate this:

```
def greet(name):
    print(f”Hello”,{name}!)
# defining the function does not output anything
greet(“Alice) 3 Calling the function executers the code inside it and output “Hello ,Alice!”
```

In this example, the greet function is defined first, but it doesn't output anything or perform any actions until it is explicitly called with greet("Alice"). At that point, the code inside the function runs and performs its task, which is to print a greeting message.

3. **What statement creates a function?**

Answer: In Python, you can create a function using the **def** statement followed by the function name and parameters, if any. Here's a simple example:

```
Def my_function():

    #Function body
```

Print(“This is my function!”)

In this example, **my_function** is the name of the function. You can then call this function elsewhere in your code by simply using **my_function()**.

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

Answer: A function is a block of code that performs a specific task when called. It is a reusable piece of code that can take input, process it, and optionally return output. Functions in programming serve to organize code, make it more readable, and promote code reusability.
On the other hand, a function call is the act of invoking or executing a function. When you call a function, you are instructing the program to run the code within that function's block. During a function call, you typically provide any required arguments or parameters that the function needs to perform its task. After the function executes, it may return a value, depending on its implementation.

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

Answer: In a Python program, there is typically one global scope, which is created when the program starts executing. This global scope includes variables and functions defined at the top level of the program, outside of any function.
Each time a function is called, a new local scope is created. Local scopes exist within the function and are destroyed when the function completes execution. Variables defined inside a function are local to that function and are not accessible outside of it.
So, the number of global scopes in a Python program is typically one, while the number of local scopes depends on the number of function calls made during program execution. Each function call creates its own local scope.

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

Answer: When a function call returns, the local scope associated with that function is destroyed. Any variables defined within that local scope cease to exist, and their memory is released. This means that variables defined inside the function are no longer accessible once the function finishes executing.

If the function has returned a value, that value can be used by the code that called the function. However, any variables that were local to the function are no longer available and cannot be accessed from outside the function. This behavior helps prevent variable name clashes and keeps the program organized by limiting the scope of variables to where they are needed.

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

Answer: The concept of a return value in programming refers to the value that a function provides back to the code that called it after the function has finished executing. When a function is called, it may perform some operations and optionally return a result to the caller. This returned value can then be used by the calling code for further computation, assignment to a variable, or any other purpose.

For example, consider a function add that adds two numbers:

```
Def add(a, b):
    Return a + b

Result = add(3, 4)

Print(result) # Output: 7
```

In this example, the add function returns the result of adding the two parameters a and b. The returned value, 7, is then assigned to the variable result and printed.

Yes, it is possible to have a return value in an expression. When a function call is used within an expression, the return value of the function becomes part of that expression. For example:

```
Total = add(2, 3) + add(4, 5)

Print(total) # Output: 14
```

In this example, the return values of the add function calls (5 and 9, respectively) are used in the expression add(2, 3) + add(4, 5) to calculate the value of total, which is then printed.

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

Answer: If a function does not have a return statement, it implicitly returns **None** when called. In Python, every function returns a value, even if no explicit return statement is provided. If the function reaches the end of its block without encountering a return statement, it automatically returns **None**.

For example:

```
def my_function():
   print("This function does not have a return statement")

result = my_function()
print(result)  # Output: None
```

In this example, the **my_function** does not have a return statement. When called, it executes the **print** statement but does not explicitly return any value. Therefore, the value of **result** is **None**.

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

Answer: In Python, if you want to use a global variable within a function and modify its value, you need to explicitly declare it as **global** within the function. This tells Python that the variable being referenced or modified is the one declared in the global scope.

Here's an example:

```
global_var = 10

def modify_global():
  global global_var
  global_var = 20

modify_global()
print(global_var)  # Output: 20
```

In this example, the **modify_global** function modifies the value of **global_var** to **20** by using the **global** keyword to specify that it's referring to the global variable, not creating a new local variable with the same name.

10. What is the data type of None?

Answer: In Python, **None** is a singleton object of type **NoneType**. This means that **None** is the sole instance of the **NoneType** class. It is commonly used to represent the absence of a value or to indicate that a variable or expression does not return any value.

We can check the type of **None** using the **type()** function:

```
print(type(None))  # Output: <class 'NoneType'>
```

11. What does the sentence import areallyourpetsnamederic do?

Answer: The sentence **import areallyourpetsnamederic** is syntactically valid in Python, but it doesn't have any built-in meaning or functionality in the Python language itself. It's essentially an import statement attempting to import a module named **areallyourpetsnamederic**.

In Python, the **import** statement is used to bring in functionality from other modules or packages into your current Python script or session. Typically, you would import modules that provide specific functionality or resources that you need to use in your program.

If there were a module named **areallyourpetsnamederic** available, the **import** statement would make its functionality accessible within your Python script or interactive session. However, as of my last update, there's no standard Python module with that name, so attempting to import it would result in an ImportError unless you've defined such a module yourself.

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

Answer: After importing the **spam** module, you would call the **bacon()** function by prefixing it with the module name followed by a dot. Here's how you would do it:

```
import spam
spam.bacon()
```

In this example, assuming the **bacon()** function is defined within the **spam** module, the **import** statement makes the **spam** module available in the current scope. By using **spam.bacon()**, you're accessing the **bacon()** function within the **spam** module and calling it.

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

Answer: To prevent a program from crashing when it encounters an error, you can use exception handling. Exception handling allows you to gracefully handle errors and recover from them without terminating the program abruptly. In Python, you can use the **try**, **except**, **finally**, and **else** blocks to implement exception handling.

Here's a basic example of how to use exception handling to prevent a program from crashing:
try:

```
    # Code that may raise an error
    result = 10 / 0  # This will raise a ZeroDivisionError
except ZeroDivisionError:
    # Handle the specific error
    print("Error: Division by zero occurred")
```

In this example, the **try** block contains the code that might raise an error. If an error occurs, such as a **ZeroDivisionError** in this case, it's caught by the **except** block, preventing the program from crashing. Instead, it executes the code inside the **except** block, allowing you to handle the error gracefully.

You can have multiple **except** blocks to handle different types of errors, or you can use a generic **except** block to catch all exceptions. Additionally, you can include an optional **finally** block for cleanup code that should be executed whether or not an exception occurs.

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

Answer: The **try** and **except** clauses are part of Python's exception handling mechanism, used to gracefully handle errors that might occur during the execution of a program.

The purpose of the **try** clause is to define a block of code in which you anticipate potential errors. Any code within the **try** block is monitored for exceptions. If an exception occurs within the **try** block, the execution of the **try** block is immediately halted, and the program jumps to the corresponding **except** block (if one exists).

The purpose of the **except** clause is to define the actions to be taken if a specific type of exception occurs within the associated **try** block. You can have one or more **except** blocks to handle different types of exceptions, allowing you to handle errors in a controlled and meaningful way.

Here's a basic example to illustrate the usage of **try** and **except**:

```
try:
    # Code that may raise an error
    result = 10 / 0  # This will raise a ZeroDivisionError
except ZeroDivisionError:
```

```
    # Handle the specific error
    print("Error: Division by zero occurred")
```

In this example, the **try** block contains code that may raise a **ZeroDivisionError**. If such an error occurs, the program jumps to the **except** block, which handles the **ZeroDivisionError** by printing an error message.