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

Functions are advantageous to have in programs for several reasons:

1. Reusability: Functions can be written once and reused multiple times in a program, reducing the amount of code duplication and making the code more modular and easier to maintain.

2. Readability: Functions allow complex tasks to be broken down into smaller, more manageable pieces. This makes the code easier to read and understand, and helps developers identify and isolate bugs or errors more easily.

3. Abstraction: Functions can be used to encapsulate complex logic and functionality, making it easier to reason about and work with. By hiding implementation details behind an abstraction, functions can make the code more flexible and easier to modify or extend.

4. Code organization: Functions can be used to organize code into logical units, grouping related functionality together and making it easier to navigate and understand.

Overall, functions are a powerful tool for improving the structure, readability, and maintainability of code, and are an essential component of modern programming.

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

The code in a function is executed when the function is called, not when it is defined or specified. When a function is defined, Python simply creates an object that represents the function, without executing the code inside the function. It is only when the function is called that the code inside the function is executed.

To call a function, you simply write the function name followed by parentheses, with any required arguments inside the parentheses. For example, if you have a function called `add\_numbers()` that takes two arguments, you could call it like this:

```python

result = add\_numbers(2, 3)

```

This would call the `add\_numbers()` function with the arguments 2 and 3, and assign the result to the variable `result`. The code inside the `add\_numbers()` function would be executed, and the result would be returned and assigned to `result`.

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

In Python, the `def` statement is used to create a function. The basic syntax for defining a function is as follows:

```python

def function\_name(parameters):

# code block

# code block

# ...

return value

```

Here, `function\_name` is the name of the function, and `parameters` is a comma-separated list of input parameters that the function takes. The code block under the function definition contains the instructions that the function will execute when called, and the `return` statement specifies the value that the function should return when it completes.

For example, here is a simple function that takes two arguments and returns their sum:

```python

def add\_numbers(a, b):

return a + b

```

To use this function, you would call it with two arguments:

```python

result = add\_numbers(2, 3)

print(result) # Output: 5

```

This would call the `add\_numbers()` function with arguments 2 and 3, and return the sum 5, which would be assigned to the variable `result` and printed to the console.

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

A function is a block of reusable code that performs a specific task when called. It can take input parameters, perform some operations on them, and return a result (or perform other actions, such as printing output to the console or modifying global variables).

A function call, on the other hand, is the act of actually invoking or running the function. It is the process of passing input parameters to the function, causing the function to execute its code block, and receiving any output or other effects of the function's actions.

To illustrate the difference between a function and a function call, consider the following example:

```python

def add\_numbers(a, b):

return a + b

result = add\_numbers(2, 3)

```

Here, `add\_numbers()` is a function that takes two arguments, adds them together, and returns the result. The `result` variable is assigned the value returned by the function call `add\_numbers(2, 3)`, which is 5.

In this example, the function is the block of code defined by the `def` statement, while the function call is the actual invocation of the function with the arguments `(2, 3)`.

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

In a Python program, there is only one global scope, which is the scope in which global variables and functions are defined. The global scope is created when the program starts and lasts until the program terminates.

On the other hand, there can be multiple local scopes in a Python program, which are created whenever a function is called. Each function call creates a new local scope, which contains the local variables and function arguments that are defined within that function. When the function returns, the local scope is destroyed and any variables defined within it are deleted.

It is important to note that each local scope is isolated from all other scopes, including the global scope and other local scopes. This means that variables defined in one scope are not accessible in other scopes, unless they are explicitly passed as arguments or accessed through global variables or the `nonlocal` keyword.

In summary, a Python program has one global scope and any number of local scopes that are created and destroyed during function calls.

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

When a function call returns, the local scope in which the function was executing is destroyed, and any variables defined within that scope are deleted. This means that the values of those variables are no longer accessible outside of the function.

For example, consider the following function:

```python

def my\_function():

x = 10

print("The value of x inside the function is:", x)

my\_function()

```

In this function, a local variable `x` is defined and assigned the value 10. When the function is called, it prints the value of `x` to the console. However, once the function call returns, the local scope is destroyed, and the variable `x` is deleted.

If we were to try to access the value of `x` outside of the function, we would get a `NameError` indicating that the variable is not defined:

```python

my\_function()

print("The value of x outside the function is:", x) # NameError: name 'x' is not defined

```

This is because the variable `x` only exists within the local scope of the function, and is deleted when the function returns. If we want to access the value of `x` outside of the function, we would need to either define it as a global variable or return it as a value from the function.

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

In Python, a return value is the value that a function returns to the caller when it completes its execution. When a function is called, it can perform some computation and produce a result that can be used by the caller in further computations or assignments. The return value of a function is specified using the `return` statement followed by the value or expression that is to be returned.

For example, consider the following function that takes two arguments and returns their sum:

```python

def add\_numbers(a, b):

return a + b

```

In this function, the `return` statement returns the sum of the two arguments `a` and `b`. When the function is called, the sum is returned to the caller, who can use it in further computations:

```python

result = add\_numbers(2, 3)

print(result) # Output: 5

```

Yes, it is possible to have a return value in an expression. In Python, expressions are statements that produce a value. When a function is called inside an expression, the return value of the function can be used directly in the expression. For example, the following code calls the `add\_numbers` function inside an expression that multiplies the result by 2:

```python

result = 2 \* add\_numbers(2, 3)

print(result) # Output: 10

```

Here, the return value of `add\_numbers(2, 3)` is first computed (i.e., 5), and then multiplied by 2 to produce the final result of 10.

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

If a function does not have a `return` statement, the return value of a call to that function is `None`. In Python, `None` is a special built-in value that represents the absence of a value. If a function does not explicitly return a value using the `return` statement, Python automatically returns `None` at the end of the function execution.

For example, consider the following function that prints a message but does not return a value:

```python

def print\_message(message):

print("The message is:", message)

```

If we call this function and assign the result to a variable, we get `None` as the value of the variable:

```python

result = print\_message("Hello, world!")

print(result) # Output: None

```

This is because the function `print\_message` does not return a value, so the variable `result` is assigned the value of `None`.

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

To make a function variable refer to the global variable with the same name, you can use the `global` keyword inside the function. This tells Python to use the global variable instead of creating a new local variable with the same name.

Here is an example that shows how to use the `global` keyword to access a global variable inside a function:

```python

x = 10 # global variable

def my\_function():

global x # use the global variable x

x = 5 # modify the global variable

my\_function()

print(x) # Output: 5

```

In this example, the `my\_function` function modifies the value of the global variable `x` to 5. To access the global variable inside the function, the `global` keyword is used before the variable name. This tells Python to use the global variable instead of creating a new local variable with the same name.

After the function call, the value of the global variable `x` is 5, which is printed using the `print` statement.

**10. What is the data type of None?**

The data type of `None` in Python is called `NoneType`. It is a built-in type that represents the absence of a value. The `None` value is commonly used to indicate the lack of a return value from a function or the absence of a value in a variable.

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

```python

x = None

print(type(x)) # Output: <class 'NoneType'>

```

In this example, the variable `x` is assigned the value `None`, and the `type()` function is used to determine the data type of `x`, which is `NoneType`.

**11. What does the sentence import areallyourpetsnamederic do?**

The sentence `import areallyourpetsnamederic` attempts to import a Python module named `areallyourpetsnamederic`. If such a module exists and is located in a location where Python can find it, the import statement will succeed and any functions, classes, or variables defined in the module will become available in the current program.

However, it is unlikely that such a module named `areallyourpetsnamederic` exists, as it is a non-standard name and not a commonly used module in the Python community. Therefore, attempting to import it will result in an error.

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

If you have a `bacon()` function defined in a module named `spam`, you can call it after importing the `spam` module using the dot notation: `spam.bacon()`. This tells Python to look for the `bacon()` function inside the `spam` module and execute it.

Here is an example that demonstrates how to call the `bacon()` function after importing the `spam` module:

```python

import spam

spam.bacon() # call the bacon() function from the spam module

```

In this example, the `spam` module is imported using the `import` statement, and the `bacon()` function is called using the dot notation (`spam.bacon()`). This tells Python to look for the `bacon()` function inside the `spam` module and execute it.

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

There are several things you can do to save a program from crashing if it encounters an error:

1. Use Exception handling: You can use a `try-except` block to catch and handle any errors that may occur during program execution. By using exception handling, you can gracefully handle errors and prevent your program from crashing.

2. Validate inputs: You can validate inputs and ensure that they meet the required criteria before using them in your program. This can help prevent errors caused by invalid inputs.

3. Use defensive programming: You can use defensive programming techniques to anticipate and handle errors that may occur during program execution. This can include adding input checks, adding fallback mechanisms, and using default values.

4. Use logging: You can use logging to record and analyze errors that occur during program execution. This can help you identify and resolve issues more quickly.

5. Test your code thoroughly: You can test your code thoroughly to ensure that it is working correctly and that all possible error conditions are handled properly. This can help you catch errors before they become a problem in production.

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

The `try` and `except` clauses are used in Python for Exception Handling. The `try` block contains the code that may raise an exception, and the `except` block contains the code that handles the exception if it is raised.

The purpose of the `try` clause is to test a block of code for errors. If an error occurs in the `try` block, the code execution is immediately transferred to the `except` block.

The purpose of the `except` clause is to handle the exceptions that occur in the `try` block. It catches the raised exception and executes the code defined in the `except` block. The `except` block can be used to display an error message, log the error, or take any other action necessary to handle the exception.

Here's an example of how the `try` and `except` clauses are used in Python:

```python

try:

# code that may raise an exception

x = 10 / 0

except ZeroDivisionError:

# code to handle the exception

print("Cannot divide by zero")

```

In this example, the `try` block attempts to divide the number `10` by `0`, which raises a `ZeroDivisionError`. The `except` block catches this exception and displays an error message "Cannot divide by zero".