1. What is the role of try and exception block?

The try and exception blocks in programming are used for error handling.

The try and exception blocks in programming are used for error handling. Here's a brief explanation of their roles:

1. **Try Block**: This block contains the code that might cause an error or exception. If an error occurs, the flow of control moves to the corresponding exception block.

2. **Exception Block (or Catch Block)**: This block contains the code that handles the error. It is executed only if an error occurs in the try block. You can specify different exception types to handle various errors accordingly.

2. What is the syntax for a basic try-except block?

The syntax for a basic try-except block in Python is as follows:

```
try:
    # Code that may cause an exception
    risky_operation()
except SomeException as e:
    # Code that runs if an exception occurs
    print(f"An error occurred: {e}")
```

3. What happens if an exception occurs inside a try block and there is no matching except block?

In Python, if an exception occurs inside a `try` block and there is no matching `except` block to handle that specific exception the following happens:

- The exception propagates up the call stack.

- **If no handler is found:** If the exception reaches the top level of the program (the main function or module) without being caught, the program terminates.

- **An error message is displayed:** The interpreter prints an error message to the console, including the exception type and a traceback, which shows the sequence of function calls that led to the exception.

4. What is the difference between using a bare except block and specifying a specific exception type?

Yes,can have nested `try-except` blocks in Python. This allows for more granular error handling, where you can handle specific exceptions at different levels of code execution.

```
def process_data(data):
    try:
        value = int(data)
        try:

            result = 10 / value
            print(f"Result: {result}")
        except ZeroDivisionError:
            print("Error: Cannot divide by zero.")
        except TypeError:
            print("Error: Type error occurred during calculation")
    except ValueError:
        print("Error: Invalid data format. Please provide a number.")
    except Exception as e: # Catch any other error
        print(f"An unexpected error occurred: {e}")


process_data("10")
process_data("0")
process_data("abc")
process_data([1,2])
```

5. Can you have nested try-except blocks in Python? If yes, then give an example.

Absolutely can use multiple `except` blocks with a single `try` block in Python. This is a crucial feature for handling different types of exceptions that might occur within the same block of code.

```
def calculate(num1, num2, operation):
    try:
        num1 = int(num1)
        num2 = int(num2)

        if operation == "+":
            result = num1 + num2
        elif operation == "-":
            result = num1 - num2
        elif operation == "*":
```

```
            result = num1 * num2
        elif operation == "/":
            result = num1 / num2
        else:
            raise ValueError("Invalid operation") # Raise exception if operation is invalid

        print(f"Result: {result}")

    except ValueError as e:
        print(f"ValueError: {e}") # Handle invalid input or operation
    except ZeroDivisionError:
        print("ZeroDivisionError: Cannot divide by zero.") # Handle division by zero
    except TypeError:
        print("TypeError: Invalid input types.") # Handle type errors
    except Exception as e: # Catch any other error that might occur
        print(f"An unexpected error occurred: {e}")


calculate("10", "5", "+")
calculate("10", "0", "/")
calculate("abc", "5", "+")
calculate("10", "5", "%")
calculate([1,2], 5, "+")
```

6. Can we use multiple exception blocks, if yes then give an example.

Yes, absolutely can use multiple `except` blocks with a single `try` block in Python. This is a crucial feature for handling different types of exceptions that might occur within the same block of code.

```
def calculate(num1, num2, operation):
    try:
        num1 = int(num1)
        num2 = int(num2)

        if operation == "+":
            result = num1 + num2
        elif operation == "-":
            result = num1 - num2
        elif operation == "*":
            result = num1 * num2
        elif operation == "/":
```

```
            result = num1 / num2
        else:
            raise ValueError("Invalid operation")

        print(f"Result: {result}")

    except ValueError as e:
        print(f"ValueError: {e}")
    except ZeroDivisionError:
        print("ZeroDivisionError: Cannot divide by zero.")
    except TypeError:
        print("TypeError: Invalid input types.")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")

# Test cases
calculate("10", "5", "+")
calculate("10", "0", "/")
calculate("abc", "5", "+")
calculate("10", "5", "%")
calculate([1,2], 5, "+")
```

7. Write the reason due to which following errors are raised:

a. EOFError

`EOFError` **(End of File Error)**

- **Reason:** This error is raised when input() function hits an end-of-file condition (EOF) without reading any data. This typically happens when a program expects more input from the user (or a file) but there is no more input available.

b. FloatingPointError

`FloatingPointError`

- **Reason:** This exception is raised when a floating-point operation fails. However, due to the way floating point numbers are handled by modern hardware, this exception is rarely actually raised. Floating point operations that would cause problems like division by zero

or overflows are usually handled by returning special floating-point values like `inf` (infinity) or `nan` (not a number) according to the IEEE 754 standard.

c. IndexError

`IndexError`

- **Reason:** This is raised when you try to access a sequence (like a list, tuple, or string) using an index that is out of range. This happens if the index is negative and smaller than the negative length of the sequence, or if it is greater than or equal to the length of the sequence.

d. MemoryError

`MemoryError`

- **Reason:** This exception is raised when an operation runs out of memory. This can occur when you try to create a very large data structure (like a huge list or dictionary) that exceeds the available memory on your system.

e. OverflowError

`OverflowError`

- **Reason:** This error occurs when the result of an arithmetic operation is too large to be represented within the numeric type being used. This is less common in Python 3 for integers, as they can grow dynamically to accommodate very large values. However, it can still occur with floating-point numbers or if you are using specific numeric types like `numpy.int32` which have fixed size.

f. TabError

`TabError`

- **Reason:** This error is raised when there is inconsistent use of tabs and spaces for indentation in your Python code. Python relies on consistent indentation to define code blocks, so mixing tabs and spaces can lead to ambiguity and this error. It is recommended to use spaces consistently (usually 4 spaces) for indentation.

g. ValueError

`ValueError`

- **Reason:** This exception is raised when a function receives an argument of the correct data type but an inappropriate value. This is different from `TypeError`, which occurs when the argument is of the wrong data type altogether.

8. Write code for the following given scenario and add try-exception block to it.

a. Program to divide two numbers

```
def divide_numbers(numerator, denominator):
    try:
        result = numerator / denominator
        print(f"The result of {numerator} / {denominator} is: {result}")
        return result
    except ZeroDivisionError:
        print("Error: Cannot divide by zero.")
        return None
    except TypeError:
        print("Error: Invalid input types. Please provide numbers (int or float).")
        return None
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
        return None

divide_numbers(10, 2)
divide_numbers(10, 0)
divide_numbers(10, "2")
divide_numbers("10", 2)
divide_numbers(10.5, 2)
divide_numbers(10, 2.5)
divide_numbers([1,2], 2)

result = divide_numbers(15, 3)
if result is not None:
    print(f"The calculation was successful. The result is: {result}")

result = divide_numbers(15, 0)
if result is not None:
    print(f"The calculation was successful. The result is: {result}")
```

b. Program to convert a string to an integer

```
def string_to_integer(string):
    try:
        integer = int(string)
        print(f"The integer value of '{string}' is: {integer}")
        return integer
    except ValueError:
        print(f"Error: Cannot convert '{string}' to an integer.")
        return None
    except TypeError:
        print("Error: Input must be a string.")
        return None
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
        return None

string_to_integer("123")
string_to_integer("abc")
string_to_integer(123)
```

c. Program to access an element in a list

```
def access_list_element(my_list, index):
    try:
        element = my_list[index]
        print(f"The element at index {index} is: {element}")
        return element
    except IndexError:
        print(f"Error: Index {index} is out of range for the list.")
        return None
    except TypeError:
        print("Error: First argument must be a list, second argument must be an integer.")
        return None
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
        return None

my_list = [10, 20, 30]
access_list_element(my_list, 1)
access_list_element(my_list, 5)
access_list_element(my_list, "1")
```

d. Program to handle a specific exception

```
def open_file(filename):
    try:
        with open(filename, 'r') as f:
            contents = f.read()
            print("File contents:")
            print(contents)
            return contents
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found.")
        return None
    except OSError as e: # More general file system errors
        print(f"OS Error with file '{filename}': {e}")
        return None
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
        return None

open_file("existing_file.txt")  # Create this file to test
open_file("nonexistent_file.txt")
```

e. Program to handle any exception

```
def do_something_risky(value):
    try:
        if value == 0:
            result = 10 / value
        elif isinstance(value, str):
            result = int(value)
        elif value < 0:
          raise ValueError("Value cannot be negative")
        else:
            result = value * 2
        print(f"The result is: {result}")
        return result
    except Exception as e:
        print(f"An error occurred: {type(e).__name__}: {e}")
        return None

do_something_risky(5)
do_something_risky(0)
do_something_risky("hello")
do_something_risky("123")
do_something_risky(-1)
do_something_risky([1,2])
```