In [ ]:

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


A ) The try and except blocks are used in Python for error handling
and exception handling. They allow  to catch and handle specific types
of errors or exceptions that may occur during the execution of the code.

The general syntax of a try-except block is as follows:

    try:
    # Code that may raise an exception
    ...
except ExceptionType:
    # Code to handle the exception
    ...

    Here's how the try-except block works:

The code inside the try block is executed.
If an exception occurs within the try block, the code execution is
immediately transferred to the corresponding except block.

The except block specifies the type of exception that it can handle.

If the exception raised matches the specified type, the code inside
the except block is executed. If the exception doesn't match the specified
type, it will propagate up the call stack until it is caught by an
appropriate except block or, if not caught at all, it will cause the
program to terminate with an error message.

After the except block is executed (if an exception occurred),
the program continues with the code that follows the except block.
```

In [ ]:

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

try:
    # Code that may raise an exception
    ...
except ExceptionType:
    # Code to handle the exception
    ...

    Here's an example that demonstrates the basic
    usage of a try-except block:


try:
    numerator = 10
    denominator = 0
    result = numerator / denominator
    print("Result:", result)
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
```

In [ ]:

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

A ) If an exception occurs inside a try block and there is no matching
except block to handle that specific type of exception, the exception
will propagate up the call stack until it is caught by an appropriate
except block or, if not caught at all, it will cause the program to
terminate with an error message.

When an exception is not caught by a matching except block, it is called
an unhandled exception. When an unhandled exception occurs, Python displays
a traceback that shows the line of code where the exception occurred and
the call stack leading up to that point. The traceback includes the
exception type, the error message associated with the exception, and
the sequence of function calls that led to the exception.

Here's an example to illustrate this scenario:

try:
    numerator = 10
    denominator = 0
    result = numerator / denominator
    print("Result:", result)
except ValueError:
    print("Thiswillnot be executed since the exception type doesn't match.")

    In this example, the code inside the try block attempts to perform a
    division operation. However, since the denominator is zero, it will
    raise a ZeroDivisionError. The except ValueError block cannot handle
    this type of exception, so it doesn't catch the error. As a result,
    the exception propagates up the call stack, and Python displays a
    traceback like this:


    Traceback (most recent call last):
  File "<filename>", line <line number>, in <module>
ZeroDivisionError: division by zero
The traceback indicates that a ZeroDivisionError occurred at
a specific line in the code, and it wasn't caught by any
except block. The program terminates, and the error message is displayed.
```

In [ ]:

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

A )The difference between using a bare except block and specifying a
specific exception type in a try-except block lies in how exceptions
are handled.

Bare except block:

    try:
    # Code that may raise an exception
    ...
except:
    # Code to handle the exception
    ...

    When you use a bare except block without specifying any exception type,
    it acts as a catch-all for any exception that occurs within the try
    block. It will catch and handle all types of exceptions, regardless
    of their specific types.

Using a bare except block is generally discouraged because it can make
it difficult to understand and debug the code. It can inadvertently
catch and handle exceptions that may not have intended to catch,
potentially hiding errors or causing unexpected behavior.

Specific exception type:

    try:
    # Code that may raise an exception
    ...

except ExceptionType:
    # Code to handle the exception
    ...
if specified a specific exception type (e.g., ExceptionType) in
the except block, it will only catch and handle exceptions of that
particular type. It allows  to selectively handle specific exceptions
while letting other exceptions propagate up the call stack.

Using specific exception types in except blocks provides more control
over exception handling. It helps in distinguishing between different
types of errors and allows for targeted error handling and appropriate
actions based on the specific exception that occurred.

In general, it is recommended to be as specific as possible when catching
exceptions. Catching only the exceptions we expect and handling them
appropriately helps in better error handling, code clarity, and maintenance.
Using a bare except block should be avoided unless you have a compelling
reason to catch all exceptions.
```

```
In [ ]:
```

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

A ) Yes, it is possible to have nested try-except blocks in Python.
This means that we  can have a try-except block inside another
try or except block. This allows for handling exceptions at
different levels of code execution.

Here's an example of nested try-except blocks:

try:
    # Outer try block
    numerator = 10
    denominator = 0
    result = numerator / denominator
    print("Result:", result)
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
    try:
        # Inner try block
        value = int(input("Please enter a valid number: "))
        print("You entered:", value)
    except ValueError:
        print("Error: Invalid number entered.")
In this example, there are two try-except blocks.
The outer try block attempts to perform a division operation,
which may raise a ZeroDivisionError. If such an exception occurs,
the outer except block handles it by printing an error message.

Inside the outer except block, there is an inner try-except block.
The inner try block prompts the user to enter a number using input(),
and then attempts to convert the input to an integer using int().
If enters an invalid number (e.g., a non-numeric value), a ValueError will
be raised. The inner except block catches this specific exception and
handles it by printing an error message.

The nested try-except blocks allow for handling different types of
exceptions at different levels of the code. The inner block provides
more specific error handling related to user input, while the outer
block handles division-related errors.
```

In [ ]:

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

A )Yes, it is possible to use multiple except blocks in a
try-except statement to handle different types of exceptions.
This allows to provide specific exception handling for each type
of exception.

Here's an example of using multiple except blocks:


try:
    # Code that may raise exceptions
    file = open("nonexistent.txt", "r")
    number = int("abc")
    result = 10 / 0
except FileNotFoundError:
    print("Error: File not found.")
except ValueError:
    print("Error: Invalid value.")
except ZeroDivisionError:
    print("Error: Division by zero.")
In this example, the try block contains three lines of code that may
raise different types of exceptions:

Opening a file that doesn't exist will raise a FileNotFoundError.
Trying to convert a non-numeric string to an integer will raise
a ValueError.

Performing a division by zero will raise a ZeroDivisionError.
The except blocks following the try block handle each of these
exceptions specifically:

The FileNotFoundError is caught by the first except block and
prints an error message stating that the file was not found.

The ValueError is caught by the second except block and prints
an error message indicating that the value is invalid.

The ZeroDivisionError is caught by the third except block
and prints an error message stating that division by zero is not allowed.

When an exception occurs, Python checks each except block in
order from top to bottom until it finds a block that can handle
the specific exception. The code in that matching except block is
executed, and then the program continues with the code following the
except blocks.

Using multiple except blocks allows you to handle different types
of exceptions with specific error handling logic for each one,
providing more granular control over the exception handling process.
```

In [ ]:

```
7. Write the reason due to which following errors are raised:
a. EOFError
b. FloatingPointError
c. IndexError
d. MemoryError
e. OverflowError
f. TabError
g. ValueError

A ) Here are the reasons due to which the mentioned errors are raised:

a. EOFError: This error occurs when the input() function reaches the end
    of the file while trying to read input. It typically happens when
    there is no more data available to be read, and the program expects
    further input.

b. FloatingPointError: This error occurs when a floating-point operation,
    such as division or square root, encounters an exceptional condition.
    It can happen when attempting to divide a number by zero or when
    performing an illegal mathematical operation on floating-point numbers.

c. IndexError: This error occurs when you try to access an element
    from a sequence (e.g., a list or string) using an index that is out
    of range. It happens when you attempt to access an element at an
    invalid index that doesn't exist in the sequence.

d. MemoryError: This error occurs when the Python interpreter cannot
    allocate more memory for an object or operation. It typically happens
    when the program consumes all the available memory resources, and there
    is no more memory left for allocation.

e. OverflowError: This error occurs when the result of an arithmetic
    operation exceeds the maximum representable value for a numeric data
    type. It typically occurs in situations where the calculated result is
    too large to be stored within the available memory or the range of the
    data type.

f. TabError: This error occurs when there is an issue with the indentation
    of lines using tabs and spaces inconsistently. It happens when the
    interpreter encounters inconsistent or incorrect indentation in the
    code, such as mixing tabs and spaces or using an incorrect number of
    spaces for indentation.

g. ValueError: This error occurs when a function or operation receives
    an argument of the correct type but an inappropriate value. It happens
    when an invalid value is passed to a built-in function or method, or
    when a conversion between different data types fails due to an
    incorrect value.

These errors provide useful information for debugging and troubleshooting
code, helping developers identify specific issues in their programs.
Handling these errors appropriately through try-except blocks allows
for graceful error handling and exception recovery in Python programs.
```

In [ ]:

```
8. Write code for the following given scenario and add try-exception block
to it.
a. Program to divide two numbers
b. Program to convert a string to an integer
c. Program to access an element in a list
d. Program to handle a specific exception
e. Program to handle any exception

A )  Here's an example code that includes try-except blocks
for each scenario:

a. Program to divide two numbers:


try:
    numerator = int(input("Enter the numerator: "))
    denominator = int(input("Enter the denominator: "))
    result = numerator / denominator
    print("Result:", result)
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
except ValueError:
    print("Error: Invalid input. Please enter integers only.")

b. Program to convert a string to an integer:


try:
    string_num = input("Enter a number: ")
    number = int(string_num)
    print("Number:", number)
except ValueError:
    print("Error: Invalid input. Please enter a valid integer.")

c. Program to access an element in a list:


try:
    my_list = [1, 2, 3]
    index = int(input("Enter an index: "))
    value = my_list[index]
    print("Value:", value)
except IndexError:
    print("Error: Index is out of range.")
except ValueError:
    print("Error: Invalid input. Please enter a valid integer.")

d. Program to handle a specific exception:


try:
    num = int(input("Enter a number: "))
    if num < 0:
        raise ValueError("Number must be positive.")
    print("Number:", num)
except ValueError as err:
    print("Error:", err)

e. Program to handle any exception:
```

```
try:
    x = 10
    y = 0
    result = x / y
    print("Result:", result)
except Exception as err:
    print("An error occurred:", err)
In each scenario, the try-except block is used to handle specific types
of exceptions. The code within the try block attempts the desired operation,
and if an exception occurs, the corresponding except block is executed,
displaying an appropriate error message. The final example (e) uses the
generic Exception class to catch any type of exception.
```