In [ ]:
```
1. Role of try and exception block
```

In [ ]:
```
The try block allows you to test a block of code for errors. The except block lets you handle the error
gracefully instead of causing the program to crash.
```

In [ ]:
```
2. Syntax for a basic try-except block
```

In [ ]:
```python
try:
    # Code that might raise an exception
except ExceptionType:
    # Code to handle the exception
```

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

In [ ]:
```
If there is no matching except block for the exception, the program will terminate and display a traceback message.
```

In [ ]:
```
4. Difference between a bare except block and specifying a specific exception type
```

In [ ]:
```
A bare except block catches all exceptions, which can make debugging harder.
Specifying an exception type catches only the intended errors, making the code more predictable and safe.
```

In [ ]:
```python
try:
    x = 10 / 0
except ZeroDivisionError:
    print("Division by zero is not allowed.")  # Specific exception
except:
    print("Some other error occurred.")  # Bare except (not recommended)
```

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

In [ ]:
```
Yes, nested try-except blocks are allowed in Python.
```

In [ ]:
```python
try:
    print("Outer try block")
    try:
        print(10 / 0)  # Error
    except ZeroDivisionError:
        print("Inner except block: Division by zero")
except:
    print("Outer except block")
```

In [ ]:
```
6. Can we use multiple exception blocks? If yes, then give an example
```

In [ ]:
```python
# Yes, multiple except blocks can be used to handle different exceptions.


try:
    x = int(input("Enter a number: "))
    print(10 / x)
except ValueError:
    print("Invalid input, please enter a number.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

In [ ]:
```
7. Reasons for specific errors
a. EOFError: Raised when input() hits an end-of-file condition.
b. FloatingPointError: Occurs during an invalid floating-point operation.
c. IndexError: Raised when a list index is out of range.
d. MemoryError: Occurs when an operation runs out of memory.
e. OverflowError: Raised when a calculation exceeds the maximum value for a numeric type.
f. TabError: Raised when inconsistent indentation involves tabs and spaces.
g. ValueError: Occurs when a function receives an argument of the right type but inappropriate value.
```

In [ ]:
```
8. Code with try-exception blocks
```

In [ ]:
```python
try:
    a = int(input("Enter numerator: "))
    b = int(input("Enter denominator: "))
    print("Result:", a / b)
except ZeroDivisionError:
    print("Cannot divide by zero.")
except ValueError:
    print("Invalid input, please enter numbers.")
```

In [ ]:
```python
# . Program to convert a string to an integer

try:
    s = input("Enter a number: ")
    print("Converted number:", int(s))
except ValueError:
    print("Invalid input. Please enter a valid integer.")
```

In [ ]:
```python
# c. Program to access an element in a list

try:
    lst = [10, 20, 30]
    index = int(input("Enter index to access: "))
    print("Element at index:", lst[index])
except IndexError:
    print("Index out of range.")
except ValueError:
    print("Invalid index, please enter a number.")
```

In [ ]:
```python
# d. Program to handle a specific exception

try:
    print(10 / 0)
except ZeroDivisionError:
    print("Caught a division by zero error.")
```

In [ ]:
```
e. Program to handle any exception

try:
    x = int(input("Enter a number: "))
    print(10 / x)
except Exception as e:
    print("An error occurred:", e)
```