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

Ans

try block:

This is where you place the code that might raise an exception. Python will attempt to execute the code within this block.

except block:

If an exception occurs in the try block, Python will jump to the corresponding except block. Here, you can specify how to handle the exception, whether it's by displaying an error message, logging the issue, or taking some other corrective action.

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

Ans

try:

# Code that might raise an exception

# ...

except ExceptionType:

# Code to handle the exception

# …

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

Ans

If an exception occurs inside a try block, and there is no matching except block to handle that specific exception type, the program will terminate, and Python will display an error message known as an "unhandled exception" or "traceback." This message typically includes information about the exception type, the line of code where it occurred, and the call stack.

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

Ans

Using a bare `except` block and specifying a specific exception type in a try-except statement have significant differences in terms of error handling:

1. Specific Exception Type:

- When you specify a specific exception type (e.g., `except ValueError:`), the `except` block will only catch and handle exceptions of that exact type or its subclasses. This allows you to provide tailored error handling for specific errors while letting other exceptions propagate upward for more generalized handling or debugging.

- It's considered a best practice to catch and handle only the exceptions you expect and can handle effectively, rather than catching all exceptions indiscriminately.

2. Bare `except` Block:

- A bare `except:` block (without specifying a specific exception type) is also known as a generic or catch-all exception handler. It will catch and handle any exception that occurs in the try block. While this can prevent program crashes due to unhandled exceptions, it can also make it challenging to identify the cause of an error, as it lumps all exceptions together.

- Using a bare `except:` block is generally discouraged because it can lead to silent suppression of errors that should be handled differently or reported.

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

Ans Yes, you can have nested try-except blocks in Python. This means that you can place a try-except block inside another try block to handle exceptions at different levels of your code. This can be useful for more fine-grained error handling. Here's an example:

try:

# Outer try block

x = int(input("Enter a number: "))

y = int(input("Enter another number: "))

try:

# Inner try block

result = x / y

print("Result:", result)

except ZeroDivisionError:

print("Inner: Division by zero is not allowed.")

except ValueError:

print("Outer: Invalid input. Please enter valid numbers.")

except ZeroDivisionError:

print("Outer: Division by zero is not allowed.")

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

Ans Yes, you can use multiple `except` blocks to handle different exception types in a single try-except statement. This allows you to provide specific error handling for various types of exceptions that your code might encounter. Here's an example:

```python

try:

user\_input = input("Enter a number: ")

number = int(user\_input)

result = 10 / number

except ValueError:

# Handle a ValueError (e.g., invalid input)

print("Invalid input. Please enter a valid number.")

except ZeroDivisionError:

# Handle a ZeroDivisionError (e.g., division by zero)

print("Division by zero is not allowed.")

except Exception as e:

# Handle any other exceptions (generic)

print("An error occurred:", str(e))

```

1. Write the reason due to which following errors are raised:
   1. EOFError
   2. FloatingPointError
   3. IndexError
   4. MemoryError
   5. OverflowError
   6. TabError
   7. ValueError

Ans

Here are explanations for each of the listed Python errors:

1. `EOFError`:

- Raised when an input operation reaches the end of a file or input stream unexpectedly, and there is no more data to read.

2. `FloatingPointError`:

- Raised when a floating-point operation encounters an exceptional condition, such as division by zero or an invalid operation on a floating-point number.

3. `IndexError`:

- Raised when you try to access an index that is out of range in a sequence (e.g., a list or a string). This typically occurs when the index is negative or exceeds the length of the sequence.

4. `MemoryError`:

- Raised when your Python program runs out of available memory, indicating that it cannot allocate more memory for new objects or data structures.

5. `OverflowError`:

- Raised when a mathematical operation exceeds the limits of the data type, resulting in an overflow. For example, trying to calculate an integer that is too large for the platform's integer representation may trigger this error.

6. `TabError`:

- Raised when there is an issue with the indentation in your Python code. This error occurs when mixing tabs and spaces inconsistently, causing problems with the code's readability.

7. `ValueError`:

- Raised when an operation or function receives an argument of the correct data type but an inappropriate or invalid value. For example, trying to convert a non-numeric string to an integer can trigger a `ValueError`.

1. Write code for the following given scenario and add try-exception block to it.
   1. Program to divide two numbers
   2. Program to convert a string to an integer
   3. Program to access an element in a list
   4. Program to handle a specific exception
   5. Program to handle any exception

Ans Sure, here are code snippets for the scenarios you mentioned, each with a try-except block:

1. Program to divide two numbers:

```python

try:

num1 = float(input("Enter the first number: "))

num2 = float(input("Enter the second number: "))

result = num1 / num2

print("Result:", result)

except ZeroDivisionError:

print("Division by zero is not allowed.")

except ValueError:

print("Invalid input. Please enter valid numbers.")

except Exception as e:

print("An error occurred:", e)

```

2. Program to convert a string to an integer:

```python

try:

num\_str = input("Enter a number as a string: ")

num = int(num\_str)

print("Integer value:", num)

except ValueError:

print("Invalid input. Please enter a valid integer.")

except Exception as e:

print("An error occurred:", e)

```

3. Program to access an element in a list:

```python

try:

my\_list = [1, 2, 3, 4, 5]

index = int(input("Enter the index to access: "))

value = my\_list[index]

print("Value at index", index, "is:", value)

except IndexError:

print("Index out of range. Please enter a valid index.")

except Exception as e:

print("An error occurred:", e)

```

4. Program to handle a specific exception (Custom exception handling):

```python

class CustomException(Exception):

pass

try:

age = int(input("Enter your age: "))

if age < 18:

raise CustomException("You must be 18 or older.")

else:

print("You are eligible.")

except CustomException as ce:

print(ce)

except Exception as e:

print("An error occurred:", e)

```

5. Program to handle any exception:

```python

try:

# Your code that may raise exceptions

result = 10 / 0 # Example: Division by zero

except Exception as e:

print("An error occurred:", e)

```

These code snippets demonstrate how to handle different scenarios and exceptions using try-except blocks in Python.