**Python Basic Assignment – 17th June**

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

The "try" and "exception" block is a fundamental concept in programming languages that enables developers to handle errors and exceptions gracefully. It allows you to write code in a way that anticipates and manages unexpected situations or errors that might occur during the execution of a program.

* **Try Block**: The code that might raise an exception is placed inside the "try" block. This is the section of code where you anticipate the possibility of an error occurring. If an error occurs within this block, the program will immediately jump to the corresponding "except" block without executing the remaining code inside the "try" block.
* **Except Block**: The "except" block is where you handle the exception raised in the "try" block. It contains the code that will execute when an exception occurs. The "except" block allows you to provide a fallback plan or alternative code to run when an error happens.

The general syntax in many programming languages is as follows:

try:

# Code that might raise an exception

# ...

except SomeException:

# Code to handle the exception

# ...

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

If an exception occurs inside a try block and there is no matching except block to handle that specific type of exception, the program will terminate abruptly, and an error message will be displayed. This is known as an "unhandled exception."

Here's an example of what happens when an exception is not caught:

try:

x = 10 / 0 # This will raise a ZeroDivisionError

except ValueError:

print("This won't be executed for ZeroDivisionError.")

In this example, we attempted to divide the number 10 by zero, which raises a ZeroDivisionError. However, there is no "except" block for ZeroDivisionError. Therefore, the program will terminate with an unhandled exception.

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

The difference between using a bare except block and specifying a specific exception type lies in how they handle exceptions.

* **Bare Except Block**: When you use a bare "except" block, it catches all exceptions, regardless of their type. This means that any exception raised inside the "try" block will be caught and handled by the bare "except" block, regardless of whether it is a common built-in exception like ValueError, ZeroDivisionError, or even a more critical system-level exception. Using a bare except block is generally discouraged because it can hide errors and make debugging difficult. It is harder to identify the root cause of the issue when all exceptions are caught in a generic manner.
* **Specifying a Specific Exception Type**: In contrast, specifying a specific exception type in the "except" block allows you to catch and handle only the exceptions of that particular type. This approach is more precise and recommended because it allows you to handle different exceptions differently, based on the context of your program. By catching specific exceptions, you can provide appropriate error handling for each type of exception, making your code more robust and easier to debug.

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

Yes, we can have nested try-except blocks in Python. A nested try-except block is one "try" block placed inside another "try" block. This allows you to handle exceptions at different levels of your code, providing more granular error handling and allowing for more complex error management.

def process\_data(data\_list):

try:

for item in data\_list:

try:

value = int(item) # Try to convert the item to an integer

divide\_numbers(value, value + 2)

except ValueError:

print("Error: Cannot convert the item to an integer.")

except Exception as e:

print("Error occurred during data processing:", str(e))

# Example usage:

data = [10, 5, 0, "4", 8]

try:

process\_data(data)

except Exception as e:

print("Unexpected error in the main program:", str(e))

Nested try-except blocks can be powerful when dealing with complex situations where you want to handle exceptions at different levels of your code or when you need to handle different types of exceptions separately for specific parts of your program.

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

Yes, we can use multiple "except" blocks to handle different types of exceptions in Python. This allows you to provide specific error handling for each type of exception that might occur in your code.

def divide\_numbers(a, b):

try:

result = a / b

print("Result of division:", result)

except ZeroDivisionError:

print("Error: Cannot divide by zero.")

except TypeError:

print("Error: Invalid input type for division.")

except Exception as e:

print("Error:", str(e))

# Additional error handling for other exceptions if needed

try:

divide\_numbers(10, 2)

divide\_numbers(10, 0)

divide\_numbers(10, "2")

except Exception as e:

print("Unexpected error:", str(e))

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

Here are the reasons for each of the mentioned errors:

* **EOFError**: EOFError stands for "End of File Error." It is raised when a built-in function like `input()` reaches the end of a file while trying to read user input from the console. This error occurs when the user tries to input data when the program is expecting input but the input stream has been closed or ended.
* **FloatingPointError:** The FloatingPointError is raised when a floating-point operation cannot be performed accurately or encounters some issue. This usually occurs when attempting to perform illegal floating-point operations, such as division by zero or when the result of a floating-point calculation is too large to be represented.
* **IndexError:** IndexError is raised when you try to access an index of a sequence that is out of range. This happens when you attempt to access an element using an index that does not exist in the sequence.
* **MemoryError:** MemoryError is raised when an operation cannot be completed due to insufficient memory. This error occurs when a program tries to allocate more memory than the system has available.
* **OverflowError:** OverflowError is raised when the result of an arithmetic operation exceeds the maximum representable value for the given data type. For example, it can occur in cases of large integer calculations that go beyond the limits of the integer type.
* **TabError**: TabError is raised when there is an issue with the indentation of code using tabs and spaces inconsistently. In Python, indentation is significant for defining blocks of code, so this error is raised when Python encounters inconsistent indentation in a script or module.
* **ValueError:** ValueError is raised when an operation or function receives an argument of the correct data type but an inappropriate value. For example, it can occur when trying to convert a string to a number, and the string does not represent a valid number.

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**

**(a).**

def divide\_numbers(a, b):

try:

result = a / b

print("Result of division:", result)

except ZeroDivisionError:

print("Error: Cannot divide by zero.")

except Exception as e:

print("Error:", str(e))

try:

divide\_numbers(10, 2)

divide\_numbers(10, 0)

except Exception as e:

print("Unexpected error:", str(e))

**(b).**

def convert\_to\_integer(input\_str):

try:

num = int(input\_str)

print("Integer value:", num)

except ValueError:

print("Error: Invalid input. Could not convert to integer.")

except Exception as e:

print("Error:", str(e))

try:

convert\_to\_integer("123")

convert\_to\_integer("abc")

except Exception as e:

print("Unexpected error:", str(e))

**©.**

def access\_list\_element(my\_list, index):

try:

value = my\_list[index]

print("Element at index", index, ":", value)

except IndexError:

print("Error: Index out of range.")

except Exception as e:

print("Error:", str(e))

try:

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

access\_list\_element(my\_list, 2)

access\_list\_element(my\_list, 10)

except Exception as e:

print("Unexpected error:", str(e))

**(d).**

try:

# Some code that might raise a specific exception

x = 10 / 0 # This will raise a ZeroDivisionError

except ZeroDivisionError:

print("Error: Cannot divide by zero.")

except Exception as e:

print("Error:", str(e))

**( e ).**

try:

# Some code that might raise an exception

# ...

except Exception as e:

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