Assignment 8 Exception handling-1

September 16, 2023

Q1. What is an Exception in python? Write the difference between Exceptions and syntax errors.

ANS:

In Python, an exception is an event that occurs during the execution of a program that disrupts the normal flow of the program's instructions. When an exception occurs, Python generates an exception object that contains information about the error, such as its type and details about where it occurred in the code. This exception object can be caught and handled by the program to prevent the program from crashing and to provide more meaningful error messages to the user.

Here are some common types of exceptions in Python:

->SyntaxError: These are errors that occur when the Python interpreter encounters invalid syntax in your code. They typically prevent the program from running at all. For example, if you forget to close a parenthesis or use an undefined variable, you'll get a syntax error.

->ZeroDivisionError: This exception occurs when you try to divide a number by zero.

->FileNotFoundError: Raised when an attempt to open a file fails because the file does not exist.

->IndexError: Raised when you try to access an index that is out of range in a list or other sequence.

->TypeError: Occurs when you perform an operation on a data type that is not supported, such as trying to add a string and an integer.

->ValueError: Raised when you provide an argument of the correct data type but with an invalid value, such as trying to convert a string that cannot be parsed into an integer.

Now, let's discuss the difference between exceptions and syntax errors:

0.0.1 Nature of Error:

->Exceptions: Exceptions are runtime errors that occur when the program is executed. They can happen even if the code's syntax is correct.

->Syntax Errors: Syntax errors are compile-time errors that occur when the code is being parsed by the Python interpreter before execution. These errors prevent the program from running at all.

0.0.2 Timing:

->Exceptions: They occur during program execution when a specific condition is met, causing the program to deviate from its normal flow.

->Syntax Errors: They are detected during the parsing phase before the program begins execution.

0.0.3 Handling:

->Exceptions: You can catch and handle exceptions using try-except blocks to gracefully deal with unexpected issues in your code.

->Syntax Errors: You need to fix syntax errors before you can even run the program; they cannot be caught and handled using try-except blocks.

[ ]:

Q2. What happens when an exception is not handled? Explain with an example

ANS:

When an exception is not handled in a Python program, it will propagate up the call stack until it is either caught and handled by an appropriate try-except block or until it reaches the top-level of the program. If the exception reaches the top-level without being caught and handled, it will result in the program terminating, and Python will display an error message that includes information about the unhandled exception.

[1]:
```
def divide(a, b):
    return a / b

result = divide(10, 0)
print("Result:", result)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[1], line 4
      1 def divide(a, b):
      2     return a / b
----> 4 result = divide(10, 0)
      5 print("Result:", result)

Cell In[1], line 2, in divide(a, b)
      1 def divide(a, b):
----> 2     return a / b

ZeroDivisionError: division by zero
```

[3]:
```
#Correct Method For Above Example

def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        print("Error: Division by zero is not allowed.")
        return None
```

```
result = divide(10, 0)
if result is not None:
    print("Result:", result)
else:
    print("Cannot calculate the result due to an error.")
```

```
Error: Division by zero is not allowed.
Cannot calculate the result due to an error.
```

[ ]:

Q3. Which Python statements are used to catch and handle exceptions? Explain with an example.

ANS:

In Python, the statements used to catch and handle exceptions are try and except. These statements are part of a try-except block, which allows you to specify a block of code to try and another block of code to execute if an exception is raised within the "try" block.

[5]:
```
def divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        print("Error: Division by zero is not allowed.")
        result = None
    return result

# Example 1: Division by zero
result1 = divide(10, 0)
if result1 is not None:
    print("Result 1:", result1)
else:
    print("Cannot calculate Result 1 due to an error.")
```

```
Error: Division by zero is not allowed.
Cannot calculate Result 1 due to an error.
```

[6]:
```
# Example 2: Valid division
result2 = divide(10, 2)
if result2 is not None:
    print("Result 2:", result2)
else:
    print("Cannot calculate Result 2 due to an error.")
```

```
Result 2: 5.0
```

[ ]:

Q4. Explain with an example:

a.try and else b.finally c.raise

ANS:

a. try and else:

The try and else blocks are used together in a try-except-else structure. The try block contains the code that might raise an exception, and the else block contains code that should execute if no exceptions are raised in the try block

b. finally:

The finally block is used in conjunction with a try block to specify code that should always be executed, whether an exception was raised or not. This block is often used for cleanup operations.

c. raise:

The raise statement is used to explicitly raise an exception in your code. You can use it when you want to trigger a specific exception condition based on certain criteria.

[9]:
```
# Example of try and else

try:
    num = int(input("Enter a number: "))
except ValueError:
    print("Invalid input! Please enter a valid number.")
else:
    print("You entered:", num)
```

```
Enter a number:  15
You entered: 15
```

[10]:
```
# Exapmle for finally

try:
    file = open("example.txt", "r")
    data = file.read()
except FileNotFoundError:
    print("File not found!")
else:
    print("File contents:", data)
finally:
    if 'file' in locals():
        file.close()
```

```
File not found!
```

[13]:
```
# Example for raise

def divide(a, b):
    if b == 0:
```

```
        raise ZeroDivisionError("Division by zero is not allowed.")
    return a / b

try:
    result = divide(10, 0)
except ZeroDivisionError as e:
    print("Error:", e)
else:
    print("Result:", result)
```

```
Error: Division by zero is not allowed.
```

[ ]:

Q5. What are Custom Exceptions in python? Why do we need Custom Exceptions? Explain with an example

ANS:

Custom exceptions in Python are user-defined exceptions that extend or inherit from the built-in exception classes. These exceptions allow you to define your own error conditions and provide meaningful error messages tailored to your specific application or module. You create custom exceptions by defining new classes that inherit from the Exception class or one of its subclasses.

1. Clarity and Readability: Custom exceptions make your code more readable and self-documenting. They provide context-specific information about errors, making it easier to understand what went wrong.
2. Modularity: Custom exceptions allow you to encapsulate error-handling logic within your modules or libraries. This separation of concerns makes your code more modular and maintainable.
3. Debugging: Custom exceptions can help you pinpoint issues more quickly during debugging because they provide specific information about the error, including where it occurred and what caused it.
4. Exception Hierarchy: You can create your own hierarchy of custom exceptions to reflect the structure of your application. This hierarchy allows you to catch and handle exceptions at different levels of granularity.

[14]:
```
class CustomError(Exception):
    """A custom exception for demonstrating custom exceptions."""
    def __init__(self, message):
        super().__init__(message)

def process_data(data):
    if data < 0:
        raise CustomError("Data cannot be negative.")
    return data * 2
```

```
try:
    result = process_data(-5)
except CustomError as e:
    print("Custom Error:", e)
else:
    print("Result:", result)
```

```
Custom Error: Data cannot be negative.
```

[ ]:

Q6. Create a custom exception class. Use this class to handle an exception.

ANS:

[24]:
```
# Custom exception class
class InvalidAgeError(Exception):
    def __init__(self, age):
        super().__init__(f"Invalid age: {age}. Age must be between 0 and 120.")

# Function that uses the custom exception
def register_user(name, age):
    try:
        if age < 0 or age > 120:
            raise InvalidAgeError(age)
        print(f"User {name} has been registered with age {age}.")
    except InvalidAgeError as e:
        print(f"Error: {e}")
```

[25]:
```
register_user("Alice", 25)
register_user("Bob", -5)
register_user("Charlie", 150)
```

```
User Alice has been registered with age 25.
Error: Invalid age: -5. Age must be between 0 and 120.
Error: Invalid age: 150. Age must be between 0 and 120.
```

[ ]:

[ ]: