Exception Handling in Python

Introduction

In any programming language, errors may occur during program development or execution. In Python, errors are broadly classified into **Syntax Errors** and **Runtime Errors**.

Types of Errors

1. Syntax Errors

Syntax errors occur due to invalid syntax in the program. These errors are detected by the Python interpreter before program execution.

Examples:

```
x = 10
if x == 10
   print("Hello")
```

Error: SyntaxError: invalid syntax

```
print "Hello"
```

Error: SyntaxError: Missing parentheses in call to 'print'

Note: The programmer is responsible for correcting syntax errors. Program execution starts only after all syntax errors are fixed.

2. Runtime Errors

Runtime errors occur while executing the program and are also known as **exceptions**. These errors arise due to invalid user input, programming logic, or memory problems.

Examples:

```
print(10/0)        # ZeroDivisionError
print(10/"ten")    # TypeError
x = int("ten")     # ValueError
```

What is an Exception?

An unwanted and unexpected event that disturbs the normal flow of a program is called an **exception**.

Common Exceptions:

- **ZeroDivisionError**
- **TypeError**
- **ValueError**
- **FileNotFoundError**
- **EOFError**

Core Components

The primary keywords used for exception handling are**:**

try: This block contains the code that might raise an exception.

except: This block catches and handles the exception if one occurs in the try block. You can specify the type of exception to catch for more granular control.

else: The code in this optional block is executed only if the try block runs without any exceptions.

finally: This optional block of code will run regardless of whether an exception occurred or was handled. It is often used for cleanup operations, such as closing files or network connections.

Different Types of Exceptions in Python

NameError: This Exception is raised when a name is not found in the local or global namespace.

IndexError: This Exception is raised when an invalid index is used to access a sequence.

KeyError: This Exception is thrown when the key is not found in the dictionary.

ValueError: This Exception is thrown when a built-in operation or function receives an argument of the correct type and incorrect value.

IOError: This Exception is raised when an input/output operation fails, such as when an attempt is made to open a non-existent file.

ImportError: This Exception is thrown when an import statement cannot find a module definition or a from ... import statement cannot find a name to import.

SyntaxError: This Exception is raised when the input code does not conform to the Python syntax rules.

TypeError: This Exception is thrown when an operation or function is applied to an object of inappropriate type.

AttributeError: occurs when an object does not have an attribute being referenced, such as calling a method that does not exist on an object.

ArithmeticError: A built-in exception in Python is raised when an arithmetic operation fails. This Exception is a base class for other specific arithmetic exceptions, such as ZeroDivisionError and OverflowError.

Floating point error: It is a type of arithmetic error that can occur in Python and other programming languages that use floating point arithmetic to represent real numbers.

ZeroDivisionError: This occurs when dividing a number by zero, an invalid mathematics operation.

FileExistsError: This is Python's built-in Exception thrown when creating a file or directory already in the file system.

PermissionError: A built-in exception in Python is raised when an operation cannot be completed due to a lack of permission or access rights.

```
try:
    # Code that might raise an exception
    numerator = 10
    denominator = int(input("Enter a denominator: "))
    result = numerator / denominator
except ZeroDivisionError:
    # Handle the specific ZeroDivisionError
    print("Error: You can't divide by zero!")
except ValueError:
    # Handle the specific ValueError (e.g., if input is not an integer)
    print("Error: Please enter a valid number.")
else:
    # This runs if no exception occurred in the try block
    print(f"Result is: {result}")
```

finally:

 # This block always runs

print("Execution finished.")

Need for Exception Handling

- Prevents abnormal termination of programs
- Ensures **graceful termination**
- Helps continue program execution using alternative logic
- Does not repair exceptions, but handles them

Default Exception Handling in Python

Every exception in Python is an object. For every exception type, a corresponding class exists.

If an exception occurs and no handling code is available, Python terminates the program abnormally and prints exception information.

Example:

```
print("Hello")
print(10/0)
print("Hi")
```

Exception Hierarchy

- Every exception in Python is a class
- All exception classes are child classes of **BaseException**
- Programmers usually deal with **Exception** and its child classes

Customized Exception Handling using try-except

Syntax:

```
try:
    Risky Code
except ExceptionName:
    Handling Code
```

- Risky code is placed inside try
- Handling code is placed inside except

Without try-except

```
print("stmt-1")
print(10/0)
print("stmt-3")
```

Result: Abnormal termination

With try-except

```
print("stmt-1")
try:
    print(10/0)
except ZeroDivisionError:
    print(10/2)
print("stmt-3")
```

Result: Normal termination

Control Flow in try-except

```
try:
    stmt-1
    stmt-2
    stmt-3
except Exception:
    stmt-4
stmt-5
```

Possible Cases:

- No exception → stmt-1,2,3,5 → Normal termination
- Exception matched → stmt-1,4,5 → Normal termination
- Exception not matched → Abnormal termination

Printing Exception Information

```
try:
    print(10/0)
except ZeroDivisionError as msg:
    print("Exception description:", msg)
```

Multiple except Blocks

Different exceptions require different handling logic.

```
try:
    x = int(input("Enter First Number: "))
    y = int(input("Enter Second Number: "))
```

```
    print(x/y)
except ZeroDivisionError:
    print("Can't divide by zero")
except ValueError:
    print("Please provide integer values only")
```

Note: Order of except blocks is important. Python checks from top to bottom.

Single except Block for Multiple Exceptions

```
except (ZeroDivisionError, ValueError) as msg:
    print("Invalid input:", msg)
```

Default except Block

```
except:
    print("Default exception")
```

Note: Default except block must be the last except block.

Finally Block

The finally block is used to maintain cleanup code such as resource deallocation.

Syntax:

```
try:
    Risky Code
except:
    Handling Code
finally:
    Cleanup Code
```

- finally block is always executed
- Except when os._exit() is used

else Block with try-except-finally

```
try:
    Risky Code
except:
    Exception Handling
else:
    Executes if no exception
finally:
    Always executes
```

Nested try-except-finally Blocks

Nested exception handling is possible.

```
try:
    try:
        print(10/0)
    except ZeroDivisionError:
        print("Inner except")
    finally:
        print("Inner finally")
except:
    print("Outer except")
finally:
    print("Outer finally")
```

Types of Exceptions

1. Predefined Exceptions

Also known as **in-built exceptions**. Raised automatically by Python.

Examples: - ZeroDivisionError - ValueError

2. User Defined Exceptions

Also known as **Customized Exceptions** or **Programmatic Exceptions**.

Programmers define and raise these exceptions explicitly using raise keyword.

Defining and Raising Customized Exceptions

Syntax:

```
class CustomException(Exception):
    def __init__(self, msg):
        self.msg = msg
```

Example:

```
class TooYoungException(Exception):
    def __init__(self, msg):
        self.msg = msg

class TooOldException(Exception):
    def __init__(self, msg):
        self.msg = msg

age = int(input("Enter Age:"))
```

```
if age > 60:
    raise TooYoungException("Please wait some more time")
elif age < 18:
    raise TooOldException("Not eligible")
else:
    print("You will get match details soon")
```

Important Note

- raise keyword is best suitable for **customized exceptions**
- Not recommended for predefined exceptions

Exception handling is a critical feature of Python that helps write robust, reliable, and error-resistant programs. Proper use of **try, except, else, finally, and custom exceptions** ensures smooth program execution and graceful error handling.

Important Definitions

- **Exception:** An unwanted and unexpected event that disturbs the normal flow of a program.
- **Runtime Error:** An error that occurs during program execution.
- **Graceful Termination:** Proper termination of a program without abrupt failure.
- **Custom Exception:** A user-defined exception created using a class that extends Exception.

Programs

Program 1: Handle Division by Zero

```
try:
    a = int(input("Enter a number:"))
    b = int(input("Enter another number:"))
    print(a/b)
except ZeroDivisionError:
    print("Cannot divide by zero")
```

Program 2: Handle Multiple Exceptions

```
try:
    x = int(input("Enter First Number:"))
    y = int(input("Enter Second Number:"))
```

```
    print(x/y)
except ZeroDivisionError:
    print("Zero is not allowed")
except ValueError:
    print("Please enter integer values only")
```

Program 3: Demonstrate finally Block

```
try:
    print("Inside try")
    print(10/2)
except:
    print("Inside except")
finally:
    print("Inside finally")
```

Program 4: User Defined Exception

```
class InvalidAgeException(Exception):
    def __init__(self, msg):
        self.msg = msg

age = int(input("Enter Age:"))
if age < 18:
    raise InvalidAgeException("Age must be 18 or above")
else:
    print("Eligible")
```