Module 9 : Error and Exception Handling

What is an Exception?

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

Error in Python can be of two types i.e., Syntax errors and Exceptions. Errors are the problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program.

In Python, there are several built-in exceptions that can be raised when an error occurs during the execution of a program. Here are some of the most common types of exceptions in Python:

- **SyntaxError:** This exception is raised when the interpreter encounters a syntax error in the code, such as a misspelled keyword, a missing colon, or an unbalanced parenthesis.
- **TypeError**: This exception is raised when an operation or function is applied to an object of the wrong type, such as adding a string to an integer.
- **NameError**: This exception is raised when a variable or function name is not found in the current scope.
- **IndexError**: This exception is raised when an index is out of range for a list, tuple, or other sequence type.
- **KeyError**: This exception is raised when a key is not found in a dictionary.
- **ValueError**: This exception is raised when a function or method is called with an invalid argument or input, such as trying to convert a string to an integer when the string does not represent a valid integer.
- **AttributeError**: This exception is raised when an attribute or method is not found on an object, such as trying to access a non-existent attribute of a class instance.
- **IOError**: This exception is raised when an I/O operation, such as reading or writing a file, fails due to an input/output error.

- **ZeroDivisionError**: This exception is raised when an attempt is made to divide a number by zero.
- **ImportError**: This exception is raised when an import statement fails to find or load a module.

Difference between Syntax Error and Exceptions

Syntax Error: As the name suggests this error is caused by the wrong syntax in the code. It leads to the termination of the program.

Example:

```
amount = 10000
if(amount > 2999)
print("You are eligible to purchase Dsa Self Paced")
```

There is a syntax error in the code . The ‘if' statement should be followed by a colon (:), and the ‘print' statement should be indented to be inside the ‘if' block.

OUTPUT:

```
  File "/home/ac35380186f4ca7978956ff46697139b.py", line 4
    if(amount>2999)
                  ^
SyntaxError: invalid syntax
```

Exceptions: Exceptions are raised when the program is syntactically correct, but the code results in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

Example:

Here in this code a s we are dividing the ‘marks’ by zero so a error will occur known as ‘ZeroDivisionError’

```
marks = 10000
a = marks / 0
print(a)
```

```
Traceback (most recent call last):
  File "/home/f3ad05420ab851d4bd106ffb04229907.py", line 4, in <module>
    a=marks/0
ZeroDivisionError: division by zero
```

In the above example raised the ZeroDivisionError as we are trying to divide a number by 0.

Note: Exception is the base class for all the exceptions in Python.

Example:

1) TypeError: This exception is raised when an operation or function is applied to an object of the wrong type. Here's an example:

Here a 'TypeError' is raised as both the datatypes are different which are being added.

```
x = 5
y = "hello"
z = x + y
```

OUTPUT:

```
Traceback (most recent call last):
  File
"7edfa469-9a3c-4e4d-98f3-5544e60bff4e.py", line
4, in <module>
    z = x + y
TypeError: unsupported operand type(s) for +: 'int'
and 'str'
```

Handling an Exception.

If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a try: block. After the try: block, include an except: statement, followed by a block of code which handles the problem as elegantly as possible.

TRY_EXCEPT: To handle possible exceptions, we use a try-except block. The try: block contains one or more statements which are likely to encounter an exception. If the statements in this block are executed without an exception, the subsequent except: block is skipped. If the exception does occur, the program flow is transferred to the except: block. The statements in the except: block are meant to handle the cause of the exception appropriately. For example, returning an appropriate error message. You can specify the type of exception after the except keyword. The subsequent block will be executed only if the specified exception occurs. There may be multiple except clauses with different exception types in a single try block. If the type of exception doesn't match any of the except blocks, it will remain unhandled and the program will terminate. The rest of the statements after the except block will continue to be executed, regardless if the exception is encountered or not. You can mention a specific type of exception in front of the except keyword. The subsequent block will be executed only if the specified exception occurs. There may be multiple except clauses with different exception types in a single try block. If the type of exception doesn't match any of the except blocks, it will remain unhandled and the program will terminate.

ELSE_FINALLY: There are two other clauses that we can add to a try-except block: else and finally. else will be executed only if the try clause doesn’t raise an exception. In Python, keywords else and finally can also be used along with the try and except clauses. While the except block is executed if the exception occurs inside the try block, the else block gets processed if the try block is found to be exception free. The finally block consists of statements which should be processed regardless of an exception occurring in the try block or not. As a consequence, the error-free try block skips the except clause and enters the finally block before going on to execute the rest of the code. If, however, there's an exception in the try block, the appropriate except block will be processed, and the statements in the finally block will be processed before proceeding to the rest of the code. If you can follow this as well as you would like to, don't worry I will demonstrate these practically soon.

RAISE: Python also provides the raise keyword to be used in the context of exception handling. It causes an exception to be generated explicitly. Built-in errors are raised implicitly. However, a built-in or custom exception can be forced during execution.you can define your custom exception type to be raised. What i mean by this is. If a condition does not meet our criteria but is correct according to the Python interpreter, we can intentionally

raise an exception using the raise keyword. We can use a customized exception in conjunction with the statement.

ASSERT: In Python, the assert statement is used to continue the execute if the given condition evaluates to True. If the assert condition evaluates to False, then it raises the Assertion Error exception with the specified error message. The syntax is assert condition [, Error Message]. The assert statement can optionally include an error message string, which gets displayed along with the Assertion Error. When we're finished verifying the program, an assertion is a consistency test that we can switch on or off.

Now that you have seen the theory aspect of it, lets look into more detailed examples:

```
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Oops!  That was no valid number.  Try
again...")
```

The try statement works as follows.

- First, the try clause (the statement(s) between the try and except keywords) is executed.
- If no exception occurs, the except clause is skipped and execution of the try statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then, if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try/except block.
- If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops with an error message.
- A try statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement. An except clause may name multiple exceptions as a parenthesized tuple, for example:

```
... except (RuntimeError, TypeError,
NameError):
...     pass
```

A class in an except clause matches exceptions which are instances of the class itself or one of its derived classes (but not the other way around — an except clause listing a derived class does not match instances of its base classes). For example, the following code will print B, C, D in that order:

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

Note that if the except clauses were reversed (with except B first), it would have printed B, B, B — the first matching except clause is triggered.

When an exception occurs, it may have associated values, also known as the exception's arguments. The presence and types of the arguments depend on the exception type.

The except clause may specify a variable after the exception name. The variable is bound to the exception instance which typically has an args attribute that stores the arguments. For convenience, builtin exception types define __str__() to print all the arguments without explicitly accessing .args.

```
try:
   raise Exception('spam', 'eggs')
except Exception as inst:
   print(type(inst))    # the exception type
   print(inst.args)     # arguments stored in .args
   print(inst)         # __str__ allows args to be printed
directly,
                       # but may be overridden in exception
subclasses
   x, y = inst.args     # unpack args
   print('x =', x)
   print('y =', y)

<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

The exception's __str__() output is printed as the last part ('detail') of the message for unhandled exceptions.

BaseException is the common base class of all exceptions. One of its subclasses, Exception, is the base class of all the non-fatal exceptions. Exceptions which are not subclasses of Exception are not typically handled, because they are used to indicate that the program should terminate. They include SystemExit which is raised by sys.exit() and KeyboardInterrupt which is raised when a user wishes to interrupt the program.

Exception can be used as a wildcard that catches (almost) everything. However, it is good practice to be as specific as possible with the types of exceptions that we intend to handle, and to allow any unexpected exceptions to propagate on.

The most common pattern for handling Exception is to print or log the exception and then re-raise it (allowing a caller to handle the exception as well):

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error:", err)
except ValueError:
    print("Could not convert data to an integer.")
except Exception as err:
    print(f"Unexpected {err=}, {type(err)=}")
    raise
```

The try … except statement has an optional else clause, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception. For example:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

The use of the else clause is better than adding additional code to the try clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the try … except statement.

Exception handlers do not handle only exceptions that occur immediately in the try clause, but also those that occur inside functions that are called (even indirectly) in the try clause. For example:

```
def this_fails():
    x = 1/0

try:
    this_fails()
except ZeroDivisionError as err:
    print('Handling run-time error:', err)

Handling run-time error: division by zero
```

Raising Exceptions

The raise statement allows the programmer to force a specified exception to occur. For example:

```
raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

The sole argument to raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from BaseException, such as Exception or one of its subclasses). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```
raise ValueError  # shorthand for 'raise ValueError()'
```

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the raise statement allows you to re-raise the exception:

```
try:
    raise NameError('HiThere')
except NameError:
    print('An exception flew by!')
    raise

An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: HiThere
```

Exception Chaining

If an unhandled exception occurs inside an except section, it will have the exception being handled attached to it and included in the error message:

```
try:
    open("database.sqlite")
except OSError:
    raise RuntimeError("unable to handle error")

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
FileNotFoundError: [Errno 2] No such file or directory:
'database.sqlite'

During handling of the above exception, another
exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: unable to handle error
```

To indicate that an exception is a direct consequence of another, the raise statement allows an optional from clause:

```
# exc must be exception instance or None.
raise RuntimeError from exc
```

This can be useful when you are transforming exceptions. For example:

```
def func():
   raise ConnectionError

try:
   func()
except ConnectionError as exc:
   raise RuntimeError('Failed to open database') from exc

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in func
ConnectionError

The above exception was the direct cause of the following
exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Failed to open database
```

It also allows disabling automatic exception chaining using the from None idiom:

```
try:
   open('database.sqlite')
except OSError:
   raise RuntimeError from None

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError
```

User-defined Exception

Programs may name their own exceptions by creating a new exception class. Exceptions should typically be derived from the Exception class, either directly or indirectly.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a few attributes that allow information about the error to be extracted by handlers for the exception.

To create a User-defined Exception, we have to create a class that implements the Exception class. Exceptions need to be derived from the Exception class, either directly or indirectly. Although not mandatory, most of the exceptions are named as names that end in “Error” similar to the naming of the standard exceptions in python.

Our Exception class should Implement Exceptions to raise exceptions. In this exception class, we can either put pass or give an implementation. we can define init function. class JustException(Exception):

```
def _init_(self, message):
    print(message)
```

To raise a User-defined Exception, we can do the following:

```
raise JustException("Raise an Exception")
```

The raise keyword raises the exception mentioned after it. In this case, JustException is raised. In the output Raise an Exception will be there with a Traceback (most recent call last) and the message we passed to the _init_ class will be displayed in the exception and output.

Example: Python User-Defined Exception

```
# define Python user-defined exceptions
class InvalidAgeException(Exception):
    "Raised when the input value is less than 18"
    pass

# you need to guess this number
number = 18

try:
    input_num = int(input("Enter a number: "))
    if input_num < number:
        raise InvalidAgeException
    else:
        print("Eligible to Vote")

except InvalidAgeException:
    print("Exception occurred: Invalid Age")
```

Output

If the user input input_num is greater than 18,

Enter a number: 45

Eligible to Vote

If the user input input_num is smaller than 18,

Enter a number: 14

Exception occurred: Invalid Age

In the above example, we have defined the custom exception InvalidAgeException by creating a new class that is derived from the built-in Exception class.

Here, when input_num is smaller than 18, this code generates an exception.

When an exception occurs, the rest of the code inside the try block is skipped.

The except block catches the user-defined InvalidAgeException exception and statements inside the except block are executed.

Customizing Exception Classes

We can further customize this class to accept other arguments as per our needs.

```
class SalaryNotInRangeError(Exception):
    """Exception raised for errors in the input salary.

    Attributes:
        salary -- input salary which caused the error
        message -- explanation of the error
    """

    def __init__(self, salary, message="Salary is not in
(5000, 15000) range"):
        self.salary = salary
        self.message = message
        super().__init__(self.message)


salary = int(input("Enter salary amount: "))
if not 5000 < salary < 15000:
    raise SalaryNotInRangeError(salary)
```

OUTPUT:

```
Enter salary amount: 2000
Traceback (most recent call last):
  File "<string>", line 17, in <module>
    raise SalaryNotInRangeError(salary)
__main__.SalaryNotInRangeError: Salary is not in
(5000, 15000) range
```

Here, we have overridden the constructor of the Exception class to accept our own custom arguments salary and message.Then, the constructor of the parent Exception class is called manually with the self.message argument using super().The custom self.salary attribute is defined to be used later.

The inherited __str__ method of the Exception class is then used to display the corresponding message when SalaryNotInRangeError is raised.

Defining Clean-up Actions

The try statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

```
try:
   raise KeyboardInterrupt
finally:
   print('Goodbye, world!')

Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
```

If a finally clause is present, the finally clause will execute as the last task before the try statement completes. The finally clause runs whether or not the try statement produces an exception. The following points discuss more complex cases when an exception occurs:

- If an exception occurs during execution of the try clause, the exception may be handled by an except clause. If the exception is not handled by an except clause, the exception is re-raised after the finally clause has been executed.
- An exception could occur during execution of an except or else clause. Again, the exception is re-raised after the finally clause has been executed.
- If the finally clause executes a break, continue or return statement, exceptions are not re-raised.
- If the try statement reaches a break, continue or return statement, the finally clause will execute just prior to the break, continue or return statement's execution.
- If a finally clause includes a return statement, the returned value will be the one from the finally clause's return statement, not the value from the try clause's return statement.

FOR EXAMPLE:

```
def bool_return():
    try:
        return True
    finally:
        return False

bool_return()
False
```

A more complicated example:

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")

divide(2, 1)
result is 2.0
executing finally clause
divide(2, 0)
division by zero!
executing finally clause
divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

As you can see, the finally clause is executed in any event. The TypeError raised by dividing two strings is not handled by the except clause and therefore re-raised after the finally clause has been executed.

In real world applications, the finally clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

Raising and Handling Multiple Unrelated Exceptions

There are situations where it is necessary to report several exceptions that have occurred. This is often the case in concurrency frameworks, when several tasks may have failed in parallel, but there are also other use cases where it is desirable to continue execution and collect multiple errors rather than raise the first exception.

The builtin ExceptionGroup wraps a list of exception instances so that they can be raised together. It is an exception itself, so it can be caught like any other exception.

```
def f():
    excs = [OSError('error 1'), SystemError('error 2')]
    raise ExceptionGroup('there were problems', excs)

f()
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  |   File "<stdin>", line 3, in f
  | ExceptionGroup: there were problems
  +-+---------------- 1 ----------------
    | OSError: error 1
    +---------------- 2 ----------------
    | SystemError: error 2
    +------------------------------------
try:
    f()
except Exception as e:
    print(f'caught {type(e)}: e')

caught <class 'ExceptionGroup'>: e
```

By using except* instead of except, we can selectively handle only the exceptions in the group that match a certain type. In the following example, which shows a nested exception group, each except* clause extracts from the group exceptions of a certain type while letting all other exceptions propagate to other clauses and eventually to be reraised.

```
def f():
    raise ExceptionGroup(
        "group1",
        [
            OSError(1),
            SystemError(2),
            ExceptionGroup(
                "group2",
                [
                    OSError(3),
                    RecursionError(4)
                ]
            )
        ]
    )

try:
    f()
except* OSError as e:
    print("There were OSErrors")
except* SystemError as e:
    print("There were SystemErrors")

There were OSErrors
There were SystemErrors
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  |   File "<stdin>", line 2, in f
  | ExceptionGroup: group1
  +-+---------------- 1 ----------------
    | ExceptionGroup: group2
    +-+---------------- 1 ----------------
      | RecursionError: 4
      +------------------------------------
```

Note that the exceptions nested in an exception group must be instances, not types. This is because in practice the exceptions would typically be ones that have already been raised and caught by the program, along the following pattern:

```
excs = []
for test in tests:
    try:
        test.run()
    except Exception as e:
        excs.append(e)

if excs:
   raise ExceptionGroup("Test Failures", excs)
```