UNIT-IV

4. ERROR / 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.

4.1 PYTHON - EXCEPTIONS

Errors or inaccuracies in a program are often called as bugs. The process of finding and removing errors is called debugging. Errors can be categorized into three major groups:

1. Compile Time error 2. Logical errors **3. Runtime errors**

1. Compile Time error

Syntax errors can also be called Compile Time Error. Some of the most common compile-time errors are syntax errors, library references, incorrect import of library functions and methods, uneven bracket pair(s), etc.

A syntax error is one of the most basic types of error in programming. Whenever we do not write the proper syntax of the python programming language (or any other language) then the python interpreter or parser throws an error known as a syntax error. The syntax error simply means that the python parser is unable to understand a line of code.

Example:

```
number = 100
if number > 50
    print("Number is greater than 50!")
```

Output:

```
File "test.py", line 3
    if number > 50
                 ^
SyntaxError: invalid syntax
```

2. Logical errors

Logical Errors are those errors that cannot be caught during compilation time. As we cannot check these errors during compile time, we name them Exceptions. Since we cannot check the logical errors during compilation time, it is difficult to find them.

3. Runtime errors

ZeroDivisionError is raised by the Python interpreter when we try to divide any number by zero. Let us take an example to understand the ZeroDivisionError.

```
number = 100
divided_by_zero = number / 0
print(divided_by_zero)
```

Output:

```
Traceback (most recent call last):
  File "d:\test.py", line 2, in <module>
    divided_by_zero = number / 0
ZeroDivisionError: division by zero
```

Some Examples of Python Runtime errors

- division by zero
- performing an operation on incompatible types
- using an identifier which has not been defined
- accessing a list element, dictionary value or object attribute which doesn’t exist
- trying to access a file which doesn’t exist

4.2 STANDARD EXCEPTION MODEL

Sr.No.	Exception Name & Description
1	**Exception** Base class for all exceptions
2	**StopIteration** Raised when the next() method of an iterator does not point to any object.

3	**SystemExit** Raised by the sys.exit() function.
4	**StandardError** Base class for all built-in exceptions except StopIteration and SystemExit.
5	**ArithmeticError** Base class for all errors that occur for numeric calculation.
6	**OverflowError** Raised when a calculation exceeds maximum limit for a numeric type.
7	**FloatingPointError** Raised when a floating point calculation fails.
8	**ZeroDivisionError** Raised when division or modulo by zero takes place for all numeric types.
9	**AssertionError** Raised in case of failure of the Assert statement.
10	**AttributeError** Raised in case of failure of attribute reference or assignment.
11	**EOFError** Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
12	**ImportError** Raised when an import statement fails.
13	**KeyboardInterrupt** Raised when the user interrupts program execution, usually by pressing Ctrl+c.
14	**LookupError** Base class for all lookup errors.

15	**IndexError** Raised when an index is not found in a sequence.
16	**KeyError** Raised when the specified key is not found in the dictionary.
17	**NameError** Raised when an identifier is not found in the local or global namespace.
18	**UnboundLocalError** Raised when trying to access a local variable in a function or method but no value has been assigned to it.
19	**EnvironmentError** Base class for all exceptions that occur outside the Python environment.
20	**IOError** Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
21	**IOError** Raised for operating system-related errors.
22	**SyntaxError** Raised when there is an error in Python syntax.
23	**IndentationError** Raised when indentation is not specified properly.
24	**SystemError** Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
25	**SystemExit** Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.

26	**TypeError** Raised when an operation or function is attempted that is invalid for the specified data type.
27	**ValueError** Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
28	**RuntimeError** Raised when a generated error does not fall into any category.
29	**NotImplementedError** Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

4.3 HANDLING MULTIPLE 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.

Syntax

Here is simple syntax of *try....except...else* blocks −

```
try:
   You do your operations here;
   ......................
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

Example

This example opens a file, writes content in the, file and comes out gracefully because there is no problem at all

Live Demo

```
#!/usr/bin/python
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print "Error: can\'t find file or read data"
else:
   print "Written content in the file successfully"
   fh.close()
```

This produces the following result −

```
Written content in the file successfully
```

4.4 EXCEPTION HIERARCHY

Exceptions occur even if our code is syntactically correct, however, while executing they throw an error. They are not unconditionally fatal, errors which we get while executing are called Exceptions. There are many Built-in Exceptions in Python let’s try to print them out in a hierarchy.

For printing the tree hierarchy we will use **inspect** module in Python. The **inspect** module provides useful functions to get information about objects such as modules, classes, methods, functions, and code objects. For example, it can help you examine the contents of a class, extract and format the argument list for a function.

For building a tree hierarchy we will use **inspect.getclasstree().**

Syntax: inspect.getclasstree(classes, unique=False)

inspect.getclasstree() arranges the given list of classes into a hierarchy of nested lists. Where a nested list appears, it contains classes derived from the class whose entry immediately precedes the list.

If the unique argument is true, exactly one entry appears in the returned structure for each class in the given list. Otherwise, classes using multiple inheritance and their descendants will appear multiple times.

Let’s write a code for printing tree hierarchy for built-in exceptions:

```
# import inspect module
import inspect

# our treeClass function
def treeClass(cls, ind = 0):

      # print name of the class
    print ('-' * ind, cls.__name__)

    # iterating through subclasses
    for i in cls.__subclasses__():
        treeClass(i, ind + 3)

print("Hierarchy for Built-in exceptions is : ")

# inspect.getmro() Return a tuple
# of class  cls’s base classes.

# building a tree hierarchy
inspect.getclasstree(inspect.getmro(BaseException))

# function call
treeClass(BaseException)
```

Output:

```
Hierarchy for Built-in exceptions is:
 BaseException
--- Exception
------ TypeError
------ StopAsyncIteration
------ StopIteration
------ ImportError
```

--------- ModuleNotFoundError

--------- ZipImportError

------ OSError

--------- ConnectionError

------------ BrokenPipeError

------------ ConnectionAbortedError

------------ ConnectionRefusedError

------------ ConnectionResetError

--------- BlockingIOError

--------- ChildProcessError

--------- FileExistsError

--------- FileNotFoundError

--------- IsADirectoryError

--------- NotADirectoryError

--------- InterruptedError

--------- PermissionError

--------- ProcessLookupError

--------- TimeoutError

--------- UnsupportedOperation

------ EOFError

------ RuntimeError

--------- RecursionError

--------- NotImplementedError

--------- _DeadlockError

------ NameError

--------- UnboundLocalError

------ AttributeError

------ SyntaxError

--------- IndentationError

------------ TabError

------ LookupError

--------- IndexError

--------- KeyError

--------- CodecRegistryError

```
------ ValueError
--------- UnicodeError
------------ UnicodeEncodeError
------------ UnicodeDecodeError
------------ UnicodeTranslateError
--------- UnsupportedOperation
------ AssertionError
------ ArithmeticError
--------- FloatingPointError
--------- OverflowError
--------- ZeroDivisionError
------ SystemError
--------- CodecRegistryError
------ ReferenceError
------ MemoryError
------ BufferError
------ Warning
--------- UserWarning
--------- DeprecationWarning
--------- PendingDeprecationWarning
--------- SyntaxWarning
--------- RuntimeWarning
--------- FutureWarning
--------- ImportWarning
--------- UnicodeWarning
--------- BytesWarning
--------- ResourceWarning
------ _OptionError
------ error
------ Verbose
------ Error
------ TokenError
------ StopTokenizing
------ EndOfBlock
```

--- GeneratorExit
--- SystemExit
--- KeyboardInterrupt

4.5 DATA STREAMS

A DataStream is similar to a regular Python Collection in terms of usage but is quite different in some key ways. They are immutable; meaning that once they are created you cannot add or remove elements.
We can use streaming data for predictive analysis or any other modelling procedure to get around critical procedures.

Using a Python library can assist you in learning the stream of data using various approaches. This library enables models to learn one data point at a time, allowing for updates as needed. This method aids in learning from large amounts of data that are not stored in the main memory.
As we all know, big data primarily focuses on data storage, which incurs significant costs, and storage has the tendency to make data unstructured. This is not beneficial to machine learning algorithms.

A data stream is a series of digitally encoded coherent signals that are used to transfer data during the data transfer process. A data stream is a collection of information extracted from the receiver and provided by the data provider.

Create from a list object

You can create a DataStream from a list object:

```
from pyflink.common.typeinfo import Types
from pyflink.datastream import StreamExecutionEnvironment

env = StreamExecutionEnvironment.get_execution_environment()
ds = env.from_collection(
    collection=[(1, 'aaa|bb'), (2, 'bb|a'), (3, 'aaa|a')],
    type_info=Types.ROW([Types.INT(), Types.STRING()]))
```

The parameter type_info is optional, if not specified, the output type of the
returned DataStream will be Types.PICKLED_BYTE_ARRAY().

4.6 FILE ACCESS MODES

Access modes govern the type of operations possible in the opened file. It refers to how the file will be used once its opened. These modes also define the location of the **File Handle** in the file. File handle is like a cursor, which defines from where the data has to be read or written in the file. There are 6 access modes in python.

1. **Read Only (‘r’) :** Open text file for reading. The handle is positioned at the beginning of the file. If the file does not exists, raises the I/O error. This is also the default mode in which a file is opened.
2. **Read and Write (‘r+’):** Open the file for reading and writing. The handle is positioned at the beginning of the file. Raises I/O error if the file does not exist.
3. **Write Only (‘w’) :** Open the file for writing. For the existing files, the data is truncated and over-written. The handle is positioned at the beginning of the file. Creates the file if the file does not exist.
4. **Write and Read (‘w+’)** : Open the file for reading and writing. For an existing file, data is truncated and over-written. The handle is positioned at the beginning of the file.
5. **Append Only (‘a’)**: Open the file for writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.
6. **Append and Read (‘a+’) :** Open the file for reading and writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

4.7 ACCESS MODE WRITING

Modes	**Description**
w	1. Opens a file for writing only. 2. Overwrites the file if the file exists. 3. If the file does not exist, creates a new file for writing.

wb	1. Opens a file for writing only in binary format. 2. Overwrites the file if the file exists. 3. If the file does not exist, creates a new file for writing.
w+	1. Opens a file for both writing and reading. 2. Overwrites the existing file if the file exists. 3. If the file does not exist, creates a new file for reading and writing.
wb+	1. Opens a file for both writing and reading in binary format. 2. Overwrites the existing file if the file exists. 3. If the file does not exist, creates a new file for reading and writing.

4.8 DATA TO A FILE READING AND DATA FROM A FILE

More specifically, opening a file, reading from it, writing into it, closing it, and various file methods that you should be aware of.

Files

Files are named locations on disk to store related information. They are used to permanently store data in a non-volatile memory (e.g. hard disk).

Since Random Access Memory (RAM) is volatile (which loses its data when the computer is turned off), we use files for future use of the data by permanently storing them.

When we want to read from or write to a file, we need to open it first. When we are done, it needs to be closed so that the resources that are tied with the file are freed.

Hence, in Python, a file operation takes place in the following order:

1. Open a file
2. Read or write (perform operation)

3. Close the file

Opening Files in Python

Python has a built-in open() function to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

```
>>> f = open("test.txt")    # open file in current directory
>>> f = open("C:/Python38/README.txt")  # specifying full path
```

We can specify the mode while opening a file. In mode, we specify whether we want to read r, write w or append a to the file. We can also specify if we want to open the file in text mode or binary mode.
The default is reading in text mode. In this mode, we get strings when reading from the file.

On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like images or executable files.

Mode	Description
r	Opens a file for reading. (default)
w	Opens a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
x	Opens a file for exclusive creation. If the file already exists, the operation fails.
a	Opens a file for appending at the end of the file without truncating it. Creates a new file if it does not exist.
t	Opens in text mode. (default)
b	Opens in binary mode.
+	Opens a file for updating (reading and writing)

```
f = open("test.txt")      # equivalent to 'r' or 'rt'
f = open("test.txt",'w')  # write in text mode
f = open("img.bmp",'r+b') # read and write in binary mode
```

Closing Files in Python

When we are done with performing operations on the file, we need to properly close the file. Closing a file will free up the resources that were tied with the file. It is done using the `close()` method available in Python.

Python has a garbage collector to clean up unreferenced objects but we must not rely on it to close the file.

```
f = open("test.txt", encoding = 'utf-8')
# perform file operations
f.close()
```

This method is not entirely safe. If an exception occurs when we are performing some operation with the file, the code exits without closing the file.

A safer way is to use a try...finally block.

```
try:
   f = open("test.txt", encoding = 'utf-8')
   # perform file operations
finally:
   f.close()
```

This way, we are guaranteeing that the file is properly closed even if an exception is raised that causes program flow to stop.

The best way to close a file is by using the `with` statement. This ensures that the file is closed when the block inside the `with` statement is exited.

We don't need to explicitly call the `close()` method. It is done internally.

```
with open("test.txt", encoding = 'utf-8') as f:
    # perform file operations
```

Writing to Files in Python

In order to write into a file in Python, we need to open it in write `w`, append `a` or exclusive creation `x` mode.

We need to be careful with the `w` mode, as it will overwrite into the file if it already exists. Due to this, all the previous data are erased.

Writing a string or sequence of bytes (for binary files) is done using the `write()` method. This method returns the number of characters written to the file.

```
with open("test.txt",'w',encoding = 'utf-8') as f:
   f.write("my first file\n")
   f.write("This file\n\n")
   f.write("contains three lines\n")
```

Reading Files in Python

To read a file in Python, we must open the file in reading `r` mode.

There are various methods available for this purpose. We can use the `read(size)` method to read in the `size` number of data. If the `size` parameter is not specified, it reads and returns up to the end of the file.

We can read the `text.txt` file we wrote in the above section in the following way:

```
>>> f = open("test.txt",'r',encoding = 'utf-8')
>>> f.read(4)    # read the first 4 data
'This'

>>> f.read(4)    # read the next 4 data
' is '

>>> f.read()     # read in the rest till end of file
'my first file\nThis file\ncontains three lines\n'
```

```
>>> f.read()  # further reading returns empty sting
''
```

We can see that the read() method returns a newline as '\n'. Once the end of the file is reached, we get an empty string on further reading.

We can change our current file cursor (position) using the seek() method. Similarly, the tell() method returns our current position (in number of bytes).

```
>>> f.tell()    # get the current file position
56

>>> f.seek(0)   # bring file cursor to initial position
0

>>> print(f.read())  # read the entire file
This is my first file
This file
contains three lines
```

We can read a file line-by-line using a for loop. This is both efficient and fast.

```
>>> for line in f:
...     print(line, end = '')
...
This is my first file
This file
contains three lines
```

In this program, the lines in the file itself include a newline character \n. So, we use the end parameter of the print() function to avoid two newlines when printing.

Alternatively, we can use the readline() method to read individual lines of a file. This method reads a file till the newline, including the newline character.

```
>>> f.readline()
```

```
'This is my first file\n'

>>> f.readline()
'This file\n'

>>> f.readline()
'contains three lines\n'

>>> f.readline()
''
```

4.9 ADDITIONAL FILE METHODS IN PYTHON

In python, there are specific pre-defined methods for working on the files and contents, such as open(), read(), readline(), next(), write(), writelines(), truncate(), seek() and close(). Open(), as the name says, used for open function, which along with other parameters like r, w, a, etc., expands the scope further. Similarly, write() can be combined with few parameters like r, a and w.

1. Open()

This function facilitates the user to open a file from a particular location and with a specific access mode.

Syntax:

```
open(address,access_mode)
```

“address” is your local system address where the file resides, and access mode is like: read-only, write-only, read and write both, etc.

2. Read()

This function facilitates reading the full file in the form of a string. However, one can restrict the reading to a certain limit by specifying the size in this function like read(size).

Example:

Let’s see example through small code,

```
f = open("C:/Users/Test/desktop/Hello.txt", "r")

print(f.read())
```

```
ff = open("C:/Users/Test/desktop/Hello.txt", "r")
print(ff.read(5))
```

Output:

3. Readline()

This helps the user reading only the first line of the file or reading line until it meets an EOF character in the file. Suppose EOF is met at first, then an empty string will be returned.

Example:

```
f = open("C:/Users/Test/desktop/Hello.txt", "r")
print(f.read())
ff = open("C:/Users/Test/desktop/Hello.txt", "r")
print(ff.readline())
```

Output:

As one can see, out of all three lines present in the file, the readline has printed only the first line.

4. Next()

file.next is helpful while iterating a file through a loop. Every time it's been called, it takes the next line.

Example:

Our file "Hello.txt" has 3 lines shown below,

Code:

```
ff = open("C:/Users/Test/desktop/Hello.txt", "r")
print(ff.next())
print(ff.next())
```

Output:

5. Write()

file.write() function is used to write the content in the output file.

Example:

As one can notice, a parameter is specified, i.e. "a". Here it means append to the content to this file. If the file exists, it will write further. If the file doesn't exist, it will be created and then written. These three types of parameters that can be used here:

- **“x”:** This is for creating a file. If the file exists with that name, an error will be thrown.
- **“a”:** This is for appending. If a file doesn’t exist, it will create that file.
- **“w”:** This will create that file if the file doesn’t exist and then write to it.

6. writelines()

Like the way readlines used to read string line by line, in iteration form. Similar way writelines is used to write string line if that’s in an iterable object form.

Code:

```
f = open("sample_EDUCBA.txt","w+")

iter_seq = ["This is good platform\n", "Datascience is buzzword"]

line = f.writelines( iter_seq )

f.close()
```

Output:

In the python console,

The generated file is,

7. truncate()

As the name suggests, this helps you shorten the file by chopping it off from anywhere required by the user.

Example: We have an input file,

Code:

```
ff = open("C:/Users/Test/desktop/sample_EDUCBA.txt", "r+")

print(ff.read())

ff = open("C:/Users/Test/desktop/sample_EDUCBA.txt", "w+")

ff.truncate()

ff = open("C:/Users/Test/desktop/sample_EDUCBA.txt", "r+")

print(ff.read())
```

Output:

Step1:

Step 2:

As one can notice, after getting truncated, there nothing printed on the console.

8. Seek()

This function helps users to set the offset position. By default, the value is 0. This is very useful when someone wants to adjust the position of the reading and writing pointer.

Example:

```
ff = open("C:/Users/i505860/sample_EDUCBA.txt", "r+")

ff.seek(0)

print(ff.readline())

ff = open("C:/Users/i505860/sample_EDUCBA.txt", "r+")

ff.seek(3)

print(ff.readline())
```

Output:

Step1:

Step2:

9. Close()

This function closes the file. A file, once it gets closed, can no more be used for reading or writing. File object created in reference to one file gets automatically closed when the same file object is assigned with another file; in python, it's always good to close the file after use. A close function might throw an error in some situations where one running out of disk space.

Example:

```
f = open("sample_EDUCBA.txt","w+")

iter_seq1 = ["This is good platform\n", "Datascience is buzzword"]

line = f.writelines( iter_seq1 )

f.close()

iter_seq2 = ["Analytics Insights\n", "Machine Learning"]

f.writelines( iter_seq2 )
```

4.10 WORKING WITH DIRECTORIES

All files are contained within various directories, and Python has no problem handling these too. The os module has several methods that help you create, remove, and change directories.

The mkdir() Method

You can use the *mkdir()* method of the os module to create directories in the current directory. You need to supply an argument to this method which contains the name of the directory to be created.

Syntax

```
os.mkdir("newdir")
```

Example

```
#!/usr/bin/python
import os
# Create a directory "test"
os.mkdir("test")
```

The chdir() Method

You can use the *chdir()* method to change the current directory. The chdir() method takes an argument, which is the name of the directory that you want to make the current directory.

Syntax

```
os.chdir("newdir")
```

Example

```
#!/usr/bin/python
import os
# Changing a directory to "/home/newdir"
os.chdir("/home/newdir")
```

The getcwd() Method

The *getcwd()* method displays the current working directory.

Syntax

```
os.getcwd()
```

Example

```
#!/usr/bin/python
import os
```

```
# This would give location of the current directory
os.getcwd()
```

The rmdir() Method

The *rmdir()* method deletes the directory, which is passed as an argument in the method.

Before removing a directory, all the contents in it should be removed.

Syntax

```
os.rmdir('dirname')
```

Example

```
#!/usr/bin/python
import os
# This would remove "/tmp/test" directory.
os.rmdir( "/tmp/test" )
```

~~~~~ The End Unit-IV ~~~~~
~~~~~