Unit 6: File Handling

Prepared By:
Tanvi Patel
Asst. Prof. (CSE)

Contents

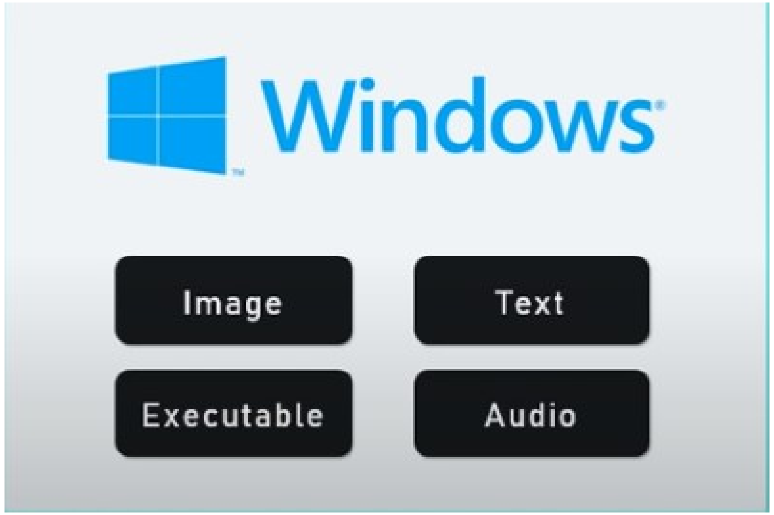
Why File Handling?

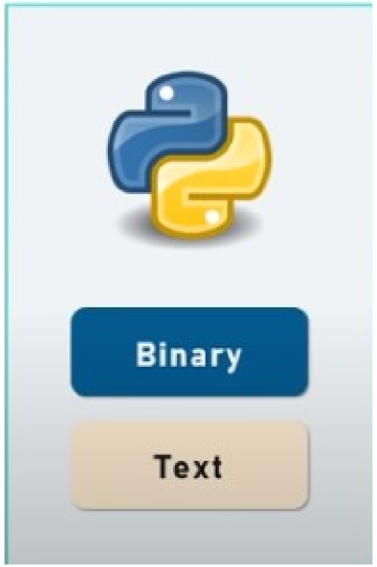
Importance of File Handling in Python

- Example: Suppose you want your python script to fetch data from the internet and then process that data. Now if data is small then this processing can be done every time you run the script but in case of huge data repetitive processing cannot be performed, hence the processed data needs to be stored. **This is where data storage or writing to a file comes into the picture.**
- An important component of an operating system is its file and directories.
- A file is a location on disk that stores related information and has a name. A hard-disk is non-volatile, and we use files to organize our data in different directories on a 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.

Types of Files

- Types of Files: Python file handling operations also known as Python I/O deal with two types of files.



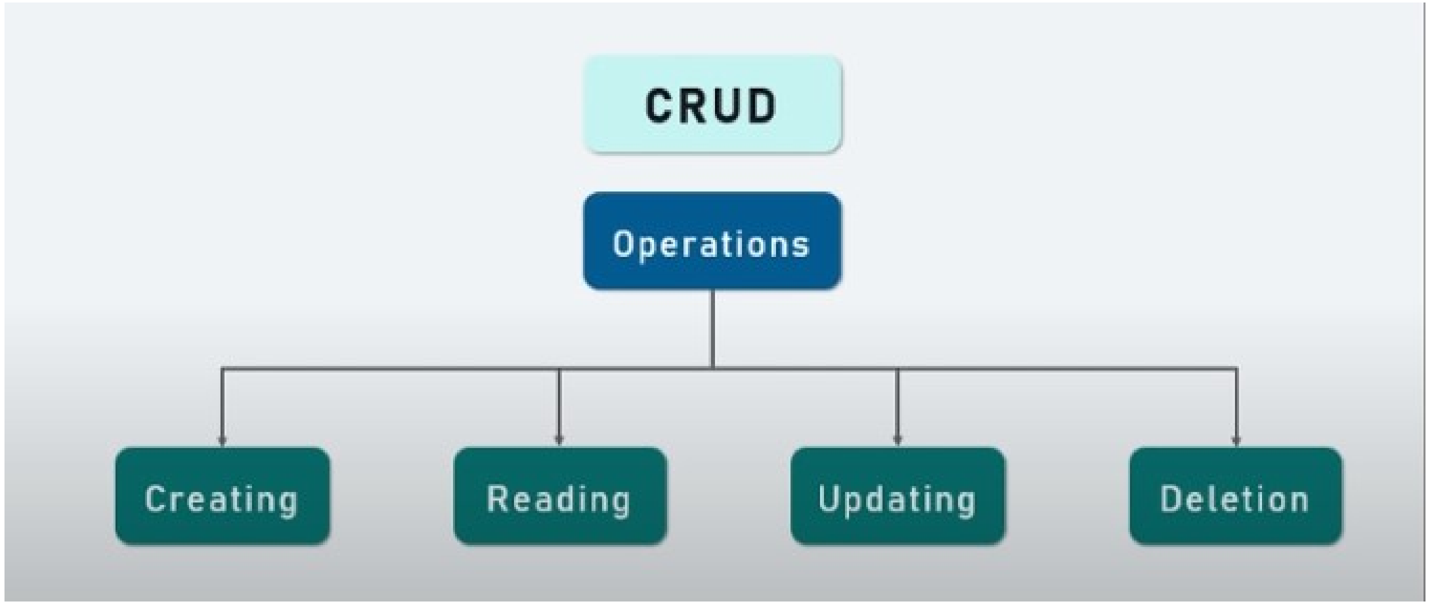


Types of Files (Cont.)

- Even though the two file types may look the same on the surface, they encode data differently.
- A text file is structured as a sequence of lines. And, each line of the text file consists of a sequence of characters. Termination of each line in a text file is denoted with the end of line (EOL). There are a few special characters that are used as EOL, but comma {,} and newlines are the most common ones.
- Image files such as .jpg, .png, .gif, etc., and documents such as .doc, .xls, .pdf, etc., all of them constitute binary files.
- **Whatever is not text is a binary.**

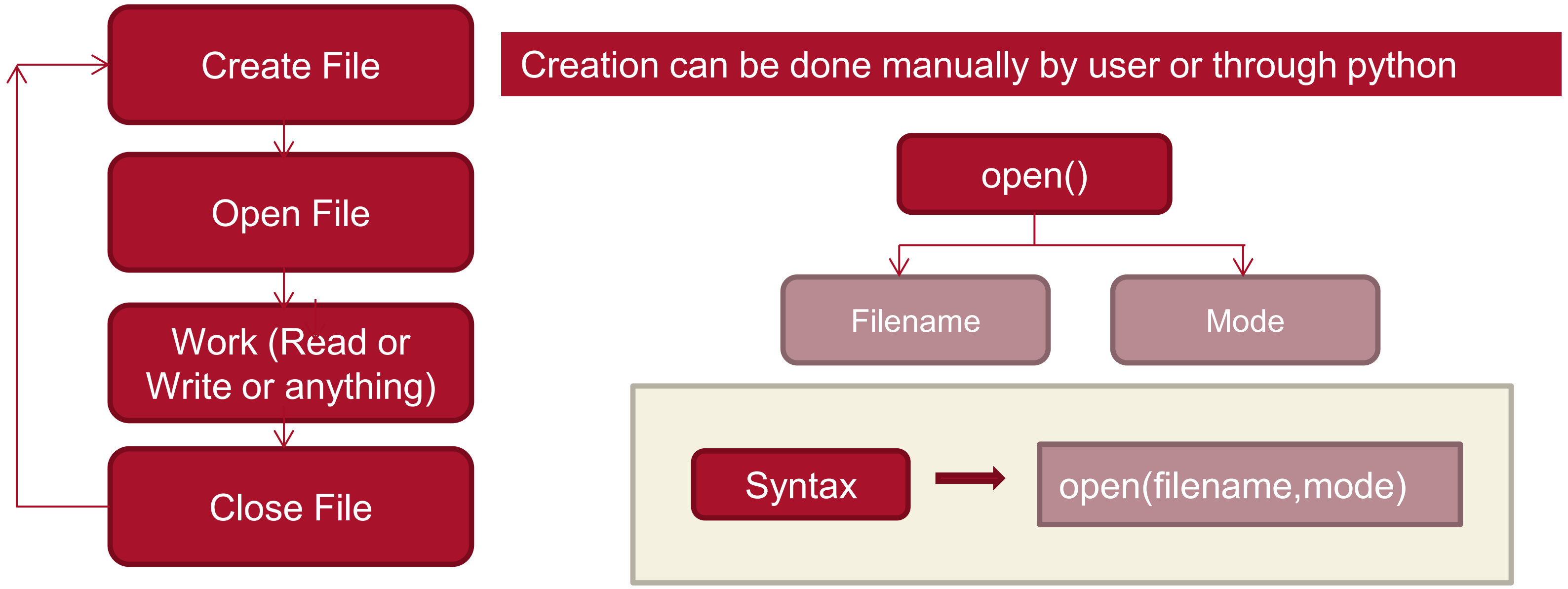
What is File Handling?

- File handling is an important part of any web application.



Python File Handling System

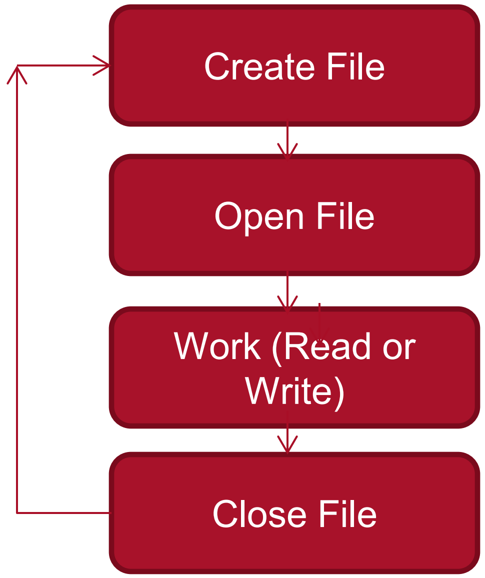
- The key function for working with files in Python is the open() function.

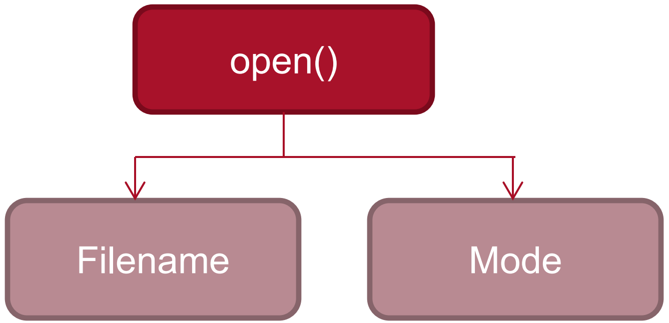


Create File
Open File
Work (Read or Write or anything)
Close File
Creation can be done manually by user or through python
open()
Filename
Mode
Syntax
open(filename,mode)

Python File Handling System

- The key function for working with files in Python is the open() function.

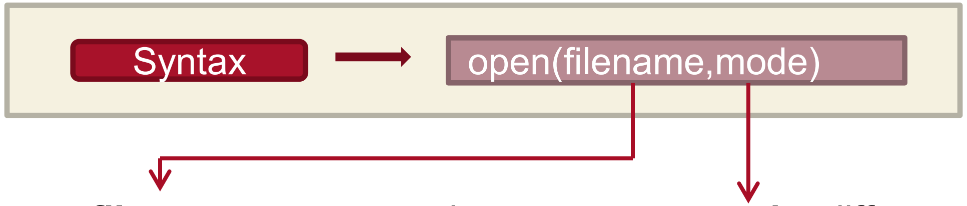




Syntax:
open(filename,mode)

Opening Files in Python

- The key function for working with files in Python is the **open()** function.
- 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.



filename: any name that you want **mode:** different modes for opening a file

"r" – **Read** – **Default mode**. Opens a file for reading, **error if the file does not exist**.

"a" – **Append** – Opens a file for appending, creates a file if does not exist.

"w" – **Write** – Opens a file for writing, creates the file if it does not exist.

"x" – **Create** – Creates the specified file, returns an error if the file exists.

Opening Files in Python (Cont.)

- 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.**

  “t” – **Text**– Default value. Text mode

  “b” – **Binary**– Binary mode (e.g., images)
- 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.

Opening Files in Python (Cont.)

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)

Opening Files in Python (Cont.)

- 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
- Unlike other languages, the character a does not imply the number 97 until it is encoded using ASCII (or other equivalent encodings).
- Moreover, the default encoding is platform dependent. In windows, it is cp1252 but utf-8 in Linux.
- So, we must not also rely on the default encoding or else our code will behave differently in different platforms.
- Hence, when working with files in text mode, it is highly recommended to specify the encoding type.
- f = open("test.txt", mode='r', encoding='utf-8')

Reading Files in Python

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

file.read()

- There are various methods available for this purpose.

All characters

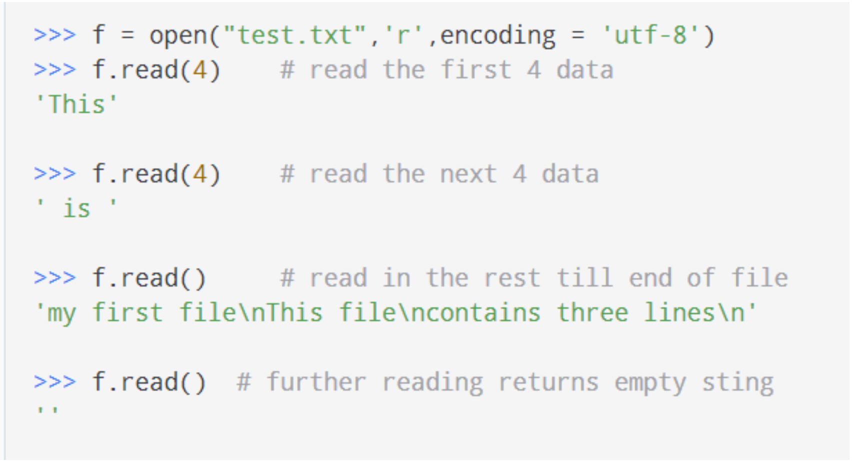
Some characters

Example

```
f = open(“test.txt”, “r”)
print(f.read())
```

Reading Files in Python (Cont.)

- 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
''
```

Reading Files in Python (Cont.)

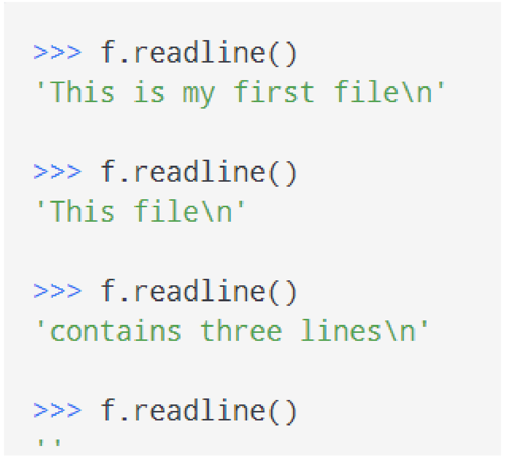
- We can see that the read() method returns a newline as '\n'. Once the end of the file is reached, we get an empty string on further reading.
- We can change our current file cursor (position) using the seek() method. Similarly, the tell() method returns our current position (in number of bytes).

```
>>> 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
```

Reading Files in Python (Cont.)



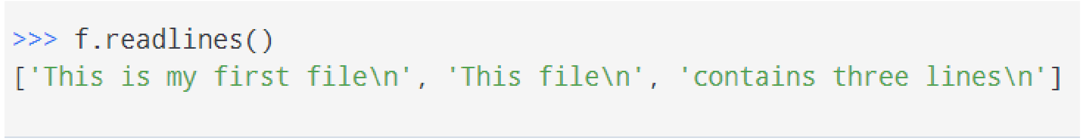
```
>>> f.readline()
'This is my first file\n'

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

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

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

- Lastly, the readlines() method returns a list of remaining lines of the entire file. All these reading methods return empty values when the end of file (EOF) is reached.



```
>>> f.readlines()
['This is my first file\n', 'This file\n', 'contains three lines\n']
```

Reading Files in Python (Cont.)

Example

```
f = open("test.txt", "r")
print(f.readline())
```

Line by line output

Example

```
f = open("test.txt", "r")
print(f.readline(3))
```

Read **third** line only

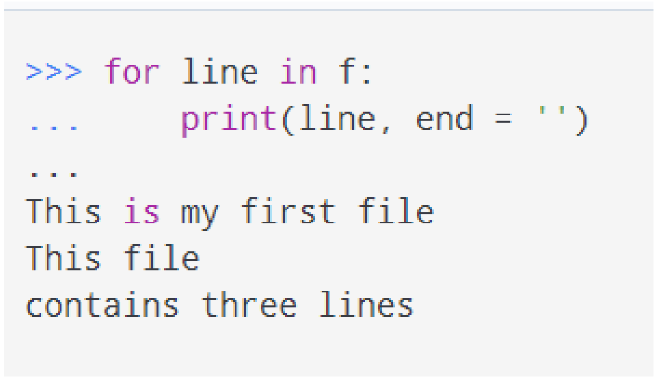
Example

```
f = open("test.txt", "r")
print(f.readlines())
```

Read lines **separately**

Looping over a File Object

- 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.

Writing to Files in Python

- To write to an existing file, we must add a parameter to the open() function.

  “a” – **Append** – will append to the end of the file.

  “w” – **Write** – will overwrite any existing content.

Example →

```
f = open(“test.txt”, “a”)
f.write(“We love Python”)
```

Example →

```
f = open(“test.txt”, “w”)
f.write(“We love Python”)
```

Note: the “w” method will overwrite the entire file.

Writing to Files in Python (Cont.)

- 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")
- This program will create a new file named test.txt in the current directory if it does not exist. If it does exist, it is overwritten.
- We must include the newline characters ourselves to distinguish the different lines.

Creating a new file

- To create a new file in Python, use the open() method with one of the following parameters:

  “x” – **Create** – Creates the specified file, returns an error if the file exists.

  “a” – **Append** – Opens a file for appending, creates a file if does not exist.

  “w” – **Write** – Opens a file for writing, creates the file if it does not exist.

```
f = open(“test.txt”, “x”)
f = open(“test.txt”, “w”)
```

Deleting a file

- To delete a file, we must import the os module and run its os.remove() function

Example ⟶

```
import os
os.remove(“demo.txt”)
```

Check if file exist

```
import os
if os.pathexists(“demo.txt”):
   os.remove(“demo.txt”)
else:
   print(“File not exist”)
```

Delete a folder

```
import os
os. rmdir(“foldername”)
```

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.

Closing Files in Python (Cont.)

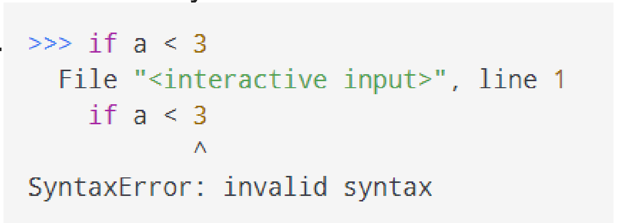
- A safer way is to use a try...finally block.
- try:
-     f = open("test.txt", encoding = 'utf-8')
-     # perform file operations
- finally:
-     f.close()
- This way, we are guaranteeing that the file is properly closed even if an exception is raised that causes program flow to stop.
- The best 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

Python Exception Handling

- We can make certain mistakes while writing a program that lead to errors when we try to run it. A python program terminates as soon as it encounters an unhandled error. These errors can be broadly classified into two classes:

1. Syntax errors
2. Logical errors (Exceptions)

- **Python Syntax Errors**
- Error caused by not following the proper structure (syntax) of the language is called **syntax error** or **parsing error**.
- Let's look at one example:
- As shown in the example, an arrow indicates where the parser ran into the syntax error.
- We can notice here that a colon : is missing in the if statement.



```
>>> if a < 3
  File "<interactive input>", line 1
    if a < 3
           ^
SyntaxError: invalid syntax
```

Python Exception Handling (Cont.)

- **Python Logical Errors (Exceptions)**
- Errors that occur at runtime (after passing the syntax test) are called **exceptions** or **logical errors**.
- For instance, they occur when we try to open a file(for reading) that does not exist (FileNotFoundError), try to divide a number by zero (ZeroDivisionError), or try to import a module that does not exist (ImportError).
- Whenever these types of runtime errors occur, Python creates an exception object. If not handled properly, it prints a traceback to that error along with some details about why that error occurred.
- Let's look at how Python treats these errors:

Python Exception Handling (Cont.)

```
>>> 1 / 0
Traceback (most recent call last):
 File "<string>", line 301, in runcode
 File "<interactive input>", line 1, in <module>
ZeroDivisionError: division by zero

>>> open("imaginary.txt")
Traceback (most recent call last):
 File "<string>", line 301, in runcode
 File "<interactive input>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'imaginary.txt'
```

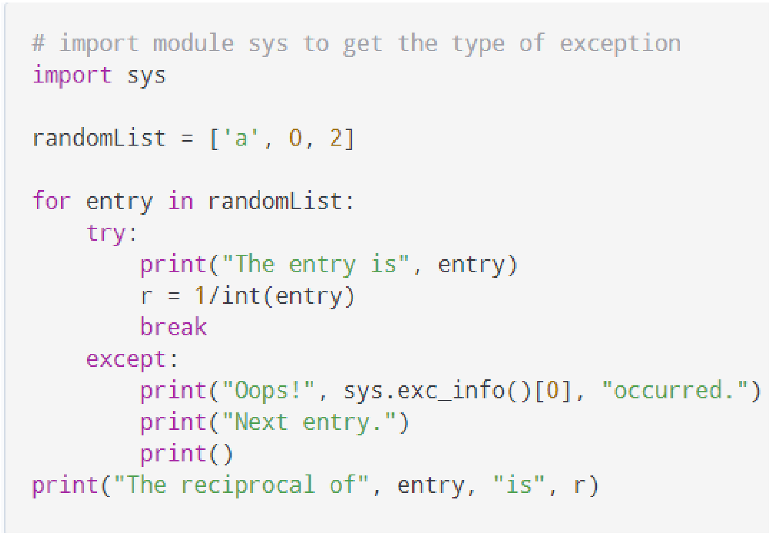
Python Exception Handling (Cont.)

- **Exceptions in Python**
- Python has many built-in exceptions that are raised when your program encounters an error (something in the program goes wrong).
- When these exceptions occur, the Python interpreter stops the current process and passes it to the calling process until it is handled. If not handled, the program will crash.
- For example, let us consider a program where we have a function A that calls function B, which in turn calls function C. If an exception occurs in function C but is not handled in C, the exception passes to B and then to A.
- If never handled, an error message is displayed and our program comes to a sudden unexpected halt.

Python Exception Handling (Cont.)

- **Catching Exceptions in Python**
- In Python, exceptions can be handled using a try statement.
- The critical operation which can raise an exception is placed inside the try clause. The code that handles the exceptions is written in the except clause.
- We can thus choose what operations to perform once we have caught the exception. Here is a simple example.

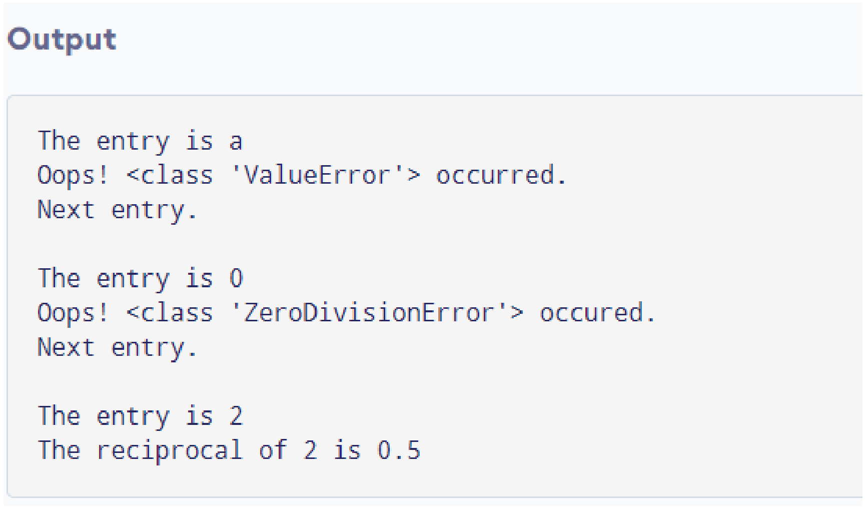
Python Exception Handling (Cont.)



```python
# import module sys to get the type of exception
import sys

randomList = ['a', 0, 2]

for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except:
        print("Oops!", sys.exc_info()[0], "occurred.")
        print("Next entry.")
        print()
print("The reciprocal of", entry, "is", r)
```



Output

```
The entry is a
Oops! <class 'ValueError'> occurred.
Next entry.

The entry is 0
Oops! <class 'ZeroDivisionError'> occured.
Next entry.

The entry is 2
The reciprocal of 2 is 0.5
```

Python Exception Handling (Cont.)

- In this program, we loop through the values of the randomList list. As previously mentioned, the portion that can cause an exception is placed inside the try block.
- If no exception occurs, the except block is skipped and normal flow continues(for last value). But if any exception occurs, it is caught by the except block (first and second values).
- Here, we print the name of the exception using the exc_info() function inside sys module. We can see that a causes ValueError and 0 causes ZeroDivisionError.

Python Exception Handling (Cont.)

- **Python try...finally**
- The try statement in Python can have an optional finally clause. This clause is executed no matter what, and is generally used to release external resources.
- For example, we may be connected to a remote data center through the network or working with a file or a Graphical User Interface (GUI).
- In all these circumstances, we must clean up the resource before the program comes to a halt whether it successfully ran or not. These actions (closing a file, GUI or disconnecting from network) are performed in the finally clause to guarantee the execution.

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

- This type of construct makes sure that the file is closed even if an exception occurs during the program execution.

Python Directory

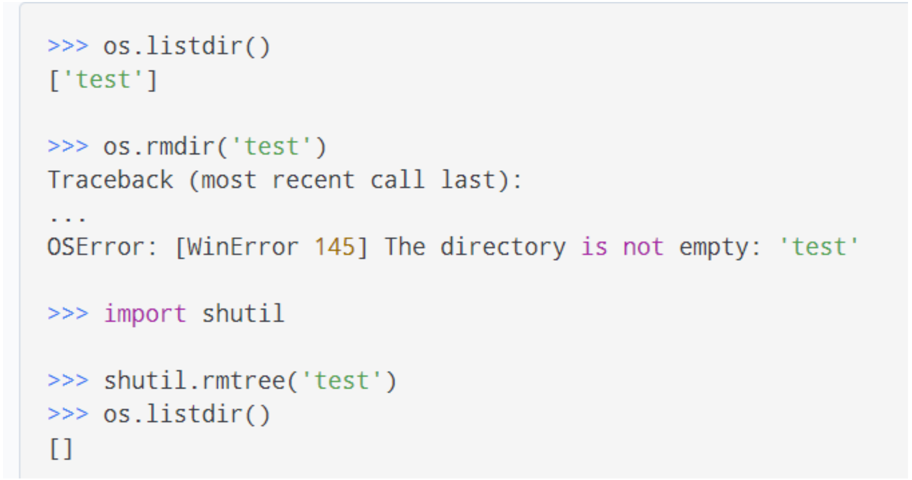
- If there are a large number of files to handle in our Python program, we can arrange our code within different directories to make things more manageable.
- A directory or folder is a collection of files and subdirectories. Python has the os module that provides us with many useful methods to work with directories (and files as well).
- **Get Current Directory**
- We can get the present working directory using the getcwd() method of the os module.
- This method returns the current working directory in the form of a string. We can also use the getcwdb() method to get it as bytes object.
- **Changing Directory**
- We can change the current working directory by using the chdir() method.
- The new path that we want to change into must be supplied as a string to this method. We can use both the forward-slash / or the backward-slash \ to separate the path elements.

Python Directory (Cont.)

- **List Directories and Files**
- All files and sub-directories inside a directory can be retrieved using the listdir() method.
- This method takes in a path and returns a list of subdirectories and files in that path. If no path is specified, it returns the list of subdirectories and files from the current working directory.
- **Making a New Directory**
- We can make a new directory using the mkdir() method.
- This method takes in the path of the new directory. If the full path is not specified, the new directory is created in the current working directory.
- **Renaming a Directory or a File**
- The rename() method can rename a directory or a file.
- For renaming any directory or file, the rename() method takes in two basic arguments: the old name as the first argument and the new name as the second argument.

Python Directory (Cont.)

- **Removing Directory or File**
- A file can be removed (deleted) using the remove() method.
- Similarly, the rmdir() method removes an empty directory.
- In order to remove a non-empty directory, we can use the rmtree() method inside the shutil module.



```
>>> os.listdir()
['test']

>>> os.rmdir('test')
Traceback (most recent call last):
...
OSError: [WinError 145] The directory is not empty: 'test'

>>> import shutil

>>> shutil.rmtree('test')
>>> os.listdir()
[]
```

File Methods

- **Python File Methods**
- There are various methods available with the file object. Some of them have been used in the above examples.
- Here is the complete list of methods in text mode with a brief description:

Method	Description
close()	Closes an opened file. It has no effect if the file is already closed.
detach()	Separates the underlying binary buffer from the TextIOBase and returns it.
fileno()	Returns an integer number (file descriptor) of the file.
flush()	Flushes the write buffer of the file stream.
isatty()	Returns True if the file stream is interactive.
read(n)	Reads at most n characters from the file. Reads till end of file if it is negative or None.

File Methods(Cont.)

Method	Description
readable()	Returns True if the file stream can be read from.
readline(n=-1)	Reads and returns one line from the file. Reads in at most n bytes if specified.
readlines(n=-1)	Reads and returns a list of lines from the file. Reads in at most n bytes/characters if specified.
seek(offset,from=SEEK_SET)	Changes the file position to offset bytes, in reference to from (start, current, end).
seekable()	Returns True if the file stream supports random access.
tell()	Returns the current file location.
truncate(size=None)	Resizes the file stream to size bytes. If size is not specified, resizes to current location.
writable()	Returns True if the file stream can be written to.
write(s)	Writes the string s to the file and returns the number of characters written.
writelines(lines)	Writes a list of lines to the file.

Thank You