ADVANCED PYTHON PROGRAMMING

Exception Handling and File Manipulation

Peter Lo

Exception

How to handle Error

Error Handling

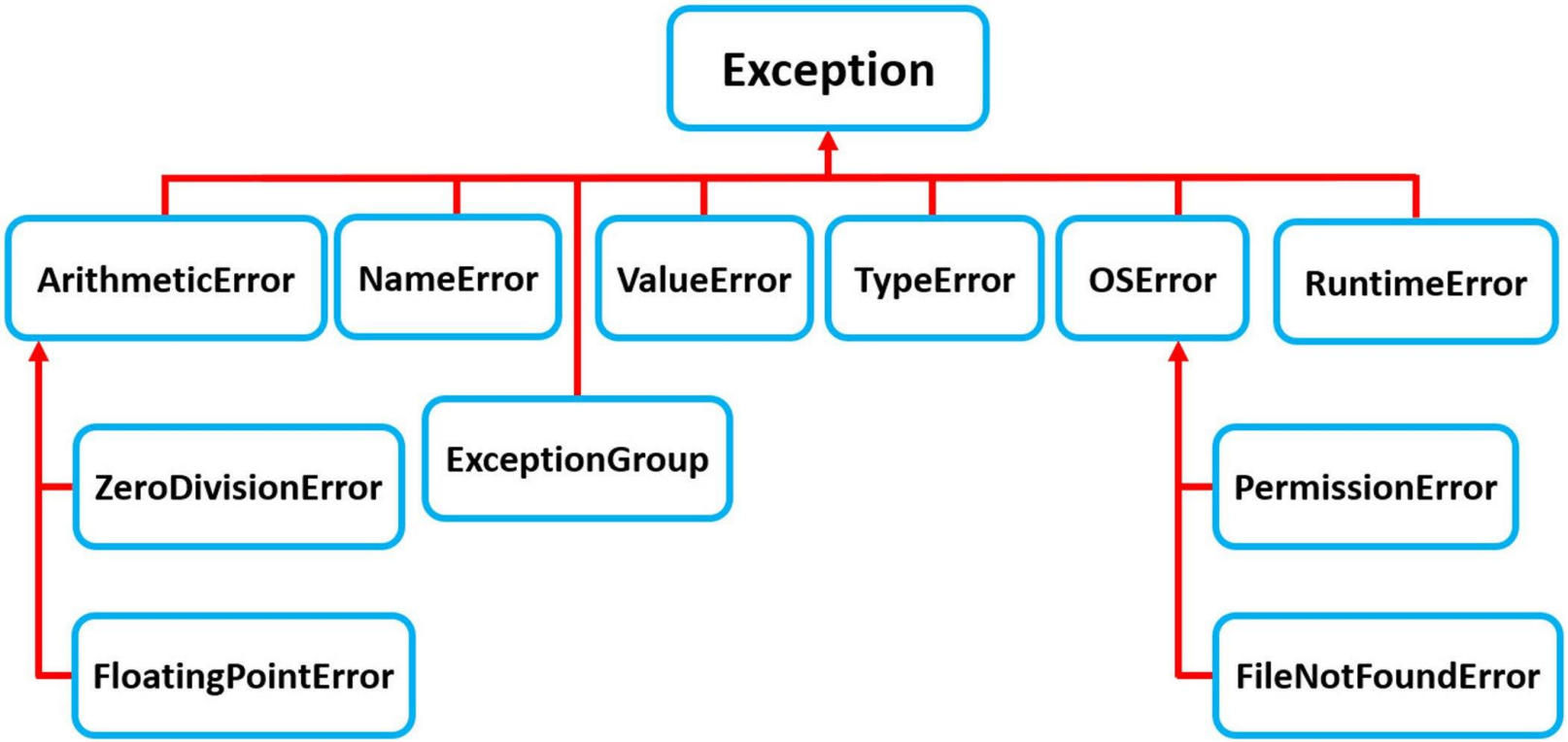
- Python provides two very important features to handle any unexpected error in your Python programs and to add debugging capabilities in them:
  - Exception Handling
  - Assertions

What is Exception?

- An exception is a condition that arises during the execution of a program.
- It is a signal that something unexpected happened.
- In Python, all built-in, non-system-exiting exceptions are derived from the Exception class. Exceptions have their own, descriptive names.

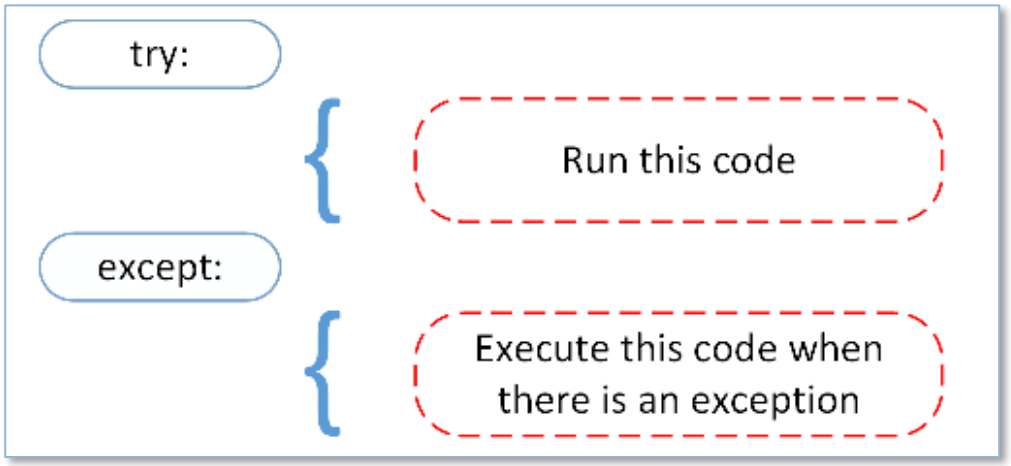
Common Exception

- Python contains over sixty different exceptions. The diagram below illustrates some of the common exceptions that you're likely to encounter:



Try and Except Block

- The try block is used to check some code for errors
  - i.e. the code inside the try block will execute when there is no error in the program.
- Whereas the code inside the except block will execute whenever the program encounters some error in the preceding try block



Catching Base Exceptions

- Using the base Exception class is another approach for catching multiple exceptions in Python.
- The Exception class is the base class for all built-in exceptions in Python, so by catching Exception, you can handle any type of exception that might occur in your code.

```
try:
    print(1/0)
except Exception as e:
    print(e)
```

```
division by zero
```

Catching Multiple Exceptions with Tuple

- Using a tuple of exception types is a simple and concise way to catch multiple exceptions in a single except block in Python.
- When an exception occurs in the try block, Python checks if the exception is an instance of any of the exception types listed in the tuple. If so, the corresponding except block is executed.

```
try:
    var = 'Hello'
    var += 5
except (TypeError, ValueError) as e:
    print(e)
```

```
can only concatenate str (not "int") to str
```

Using Multiple Except Blocks

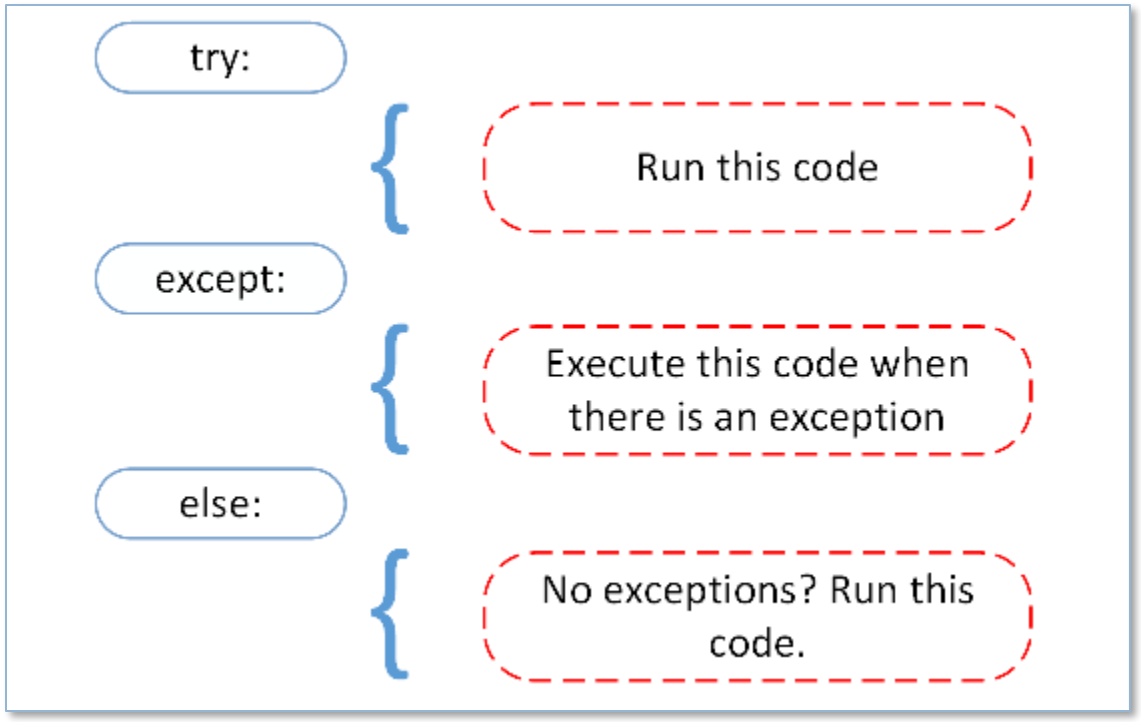
- Using separate except blocks is another way to catch multiple exceptions in Python. This method allows you to handle different exception types in different ways, making it ideal when you need to perform specific actions for each type of exception.

```
try:
    var = 'Hello'
    var += 5
except TypeError as e:
    print("TypeError occurred:", e)
except ValueError as e:
    print("ValueError occurred:", e)
```

```
TypeError occurred: can only concatenate str (not "int") to str
```

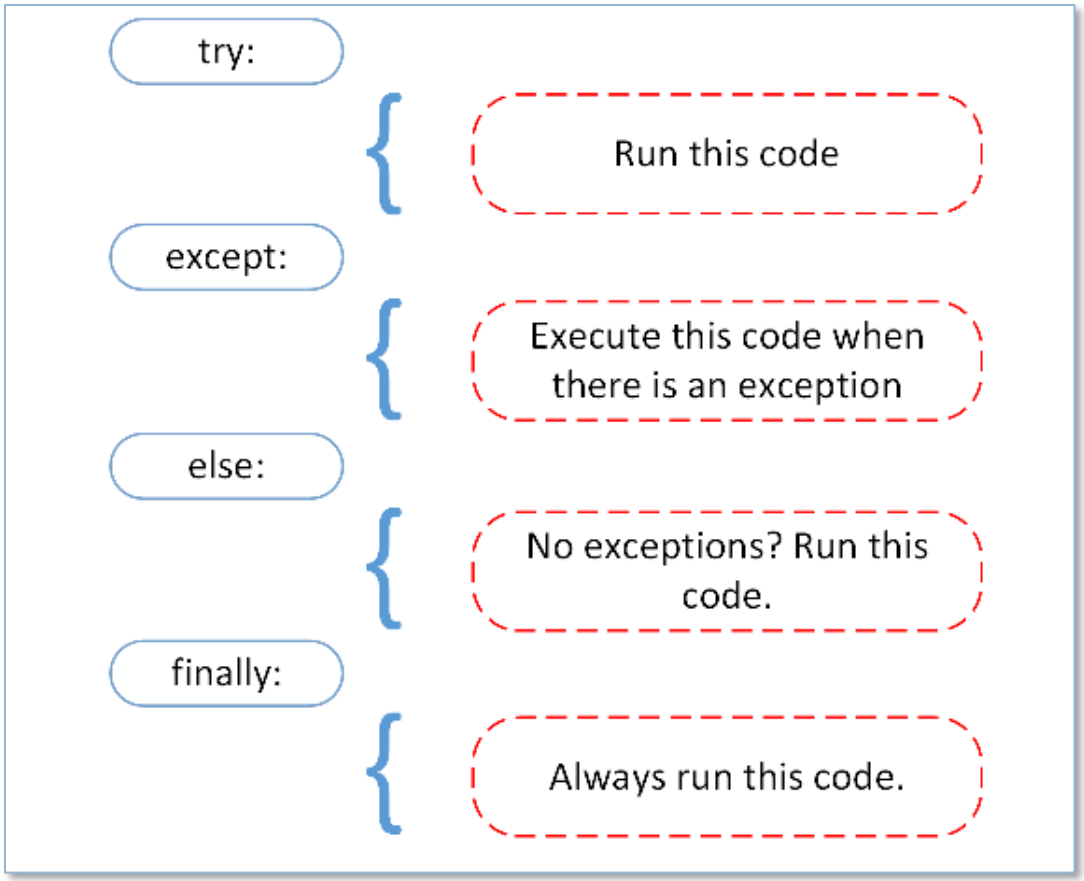
Else Block

- You can use the else keyword to define a block of code to be executed if no errors were raised:

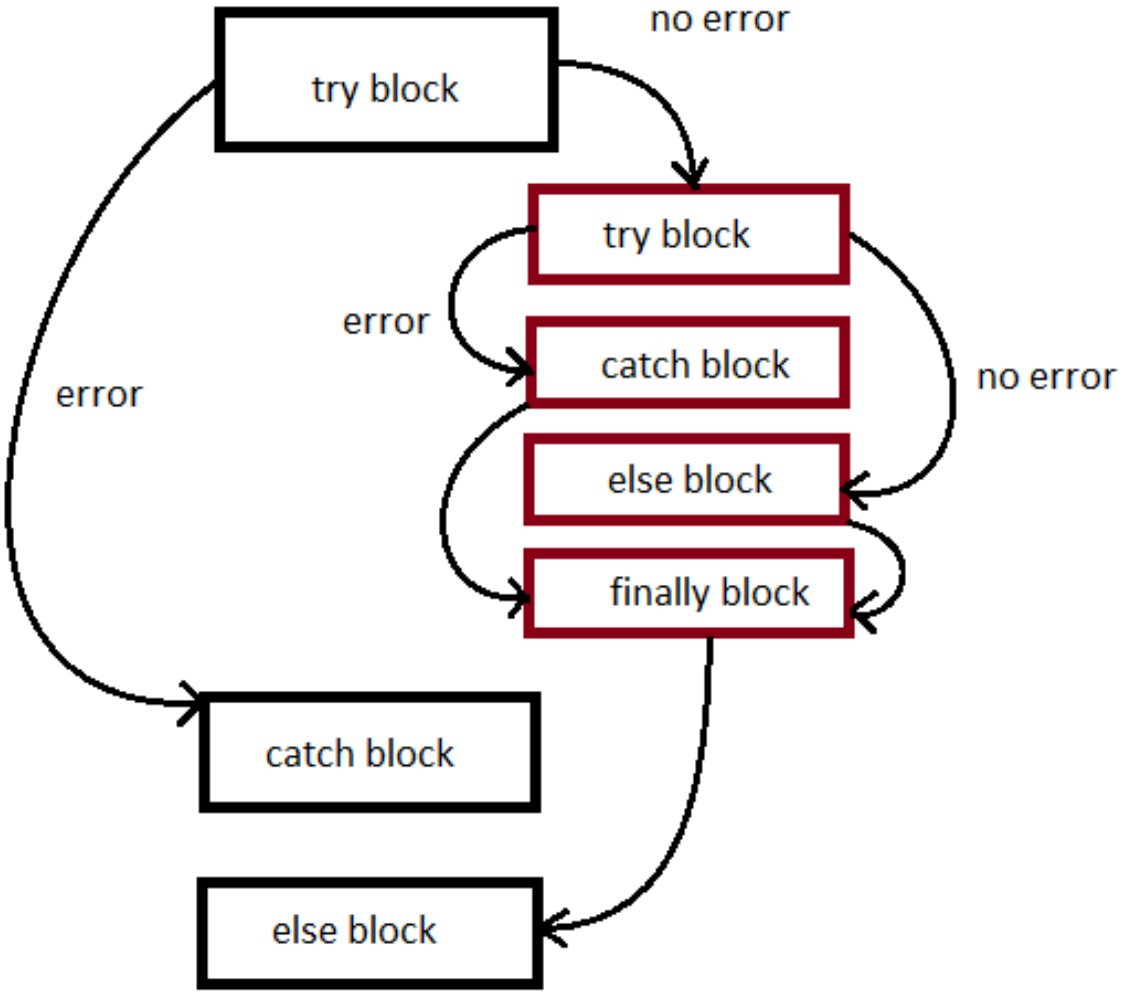


Finally Block

- The finally block, if specified, will be executed regardless if the try block raises an error or not.



Python Nested Try-Catch

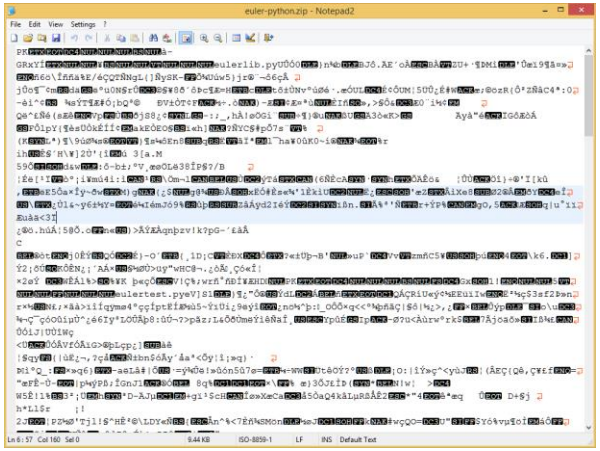


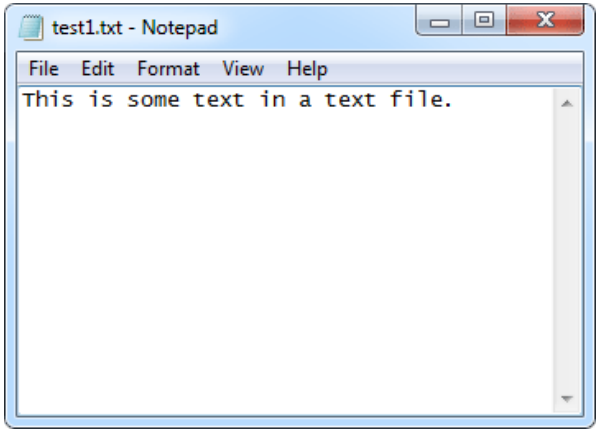
Common Python Exceptions

Exception name	When you'll encounter it	Example situation that raises the exception
SyntaxError	Raised when there is an error in Python syntax. If not caught, this exception will cause the Python interpreter to exit.	`pritn('test')`
KeyError	Raised when a key is not found in a dictionary.	`d = { 'a': 1}` `d['b']`
IndexError	Raised when an index is out of range.	`lst = [1, 2, 3]` `lst[10]`
KeyboardInterrupt	Raised when the user hits the interrupt key (Ctrl+C)	Pressing control+c

File

- There are two separate types of files that Python handles:
  - Binary file
  - Text files.
- Knowing the difference between the two is important because of how they are handled.





Binary File

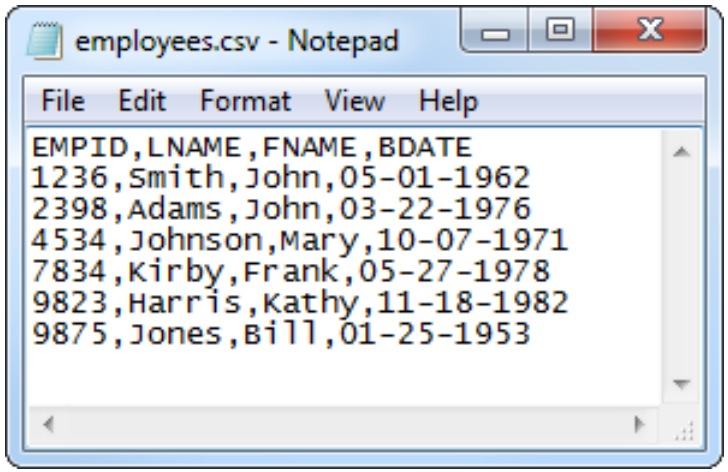
- Most files that you use in your computer are binary files. For example, Microsoft Word .doc file is a binary file, even if it has text in it.
- Other examples of binary files include:
  - Image files including .jpg, .png, .bmp, .gif, etc.
  - Database files including .mdb, .frm, and .sqlite
  - Documents including .doc, .xls, .pdf, and others.
- That's because these files all have requirements for special handling and require a specific type of software to open it. For example, you need Excel to open an .xls file

Text File

- A text file has no specific encoding and can be opened by a standard text editor without any special handling.
- Every text file must adhere to a set of rules:
  - Text files have to be readable as is. They can contain a lot of special encoding, especially in HTML or other markup languages, but you'll still be able to tell what it says
  - Data in a text file is organized by lines. In most cases, each line is a distinct element, whether it's a line of instruction or a command.
- Text files have some unseen character at the end of each line which lets the text editor know that there should be a new line. In Python, it is denoted by the "\n".

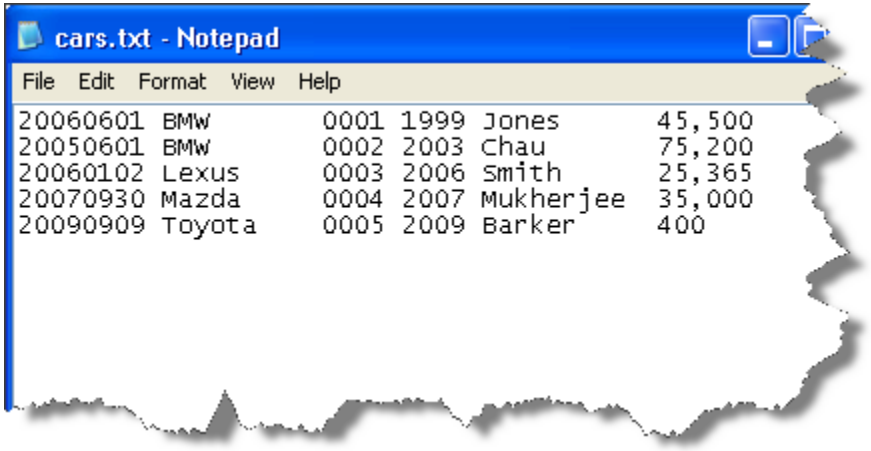
Comma Separated Value (CSV) Text File

- A CSV file is a comma separated values file commonly used by spreadsheet programs such as Microsoft Excel.
- It contains plain text data sets separated by commas with each new line in the CSV file representing a new database row and each database row consisting of one or more fields separated by a comma.



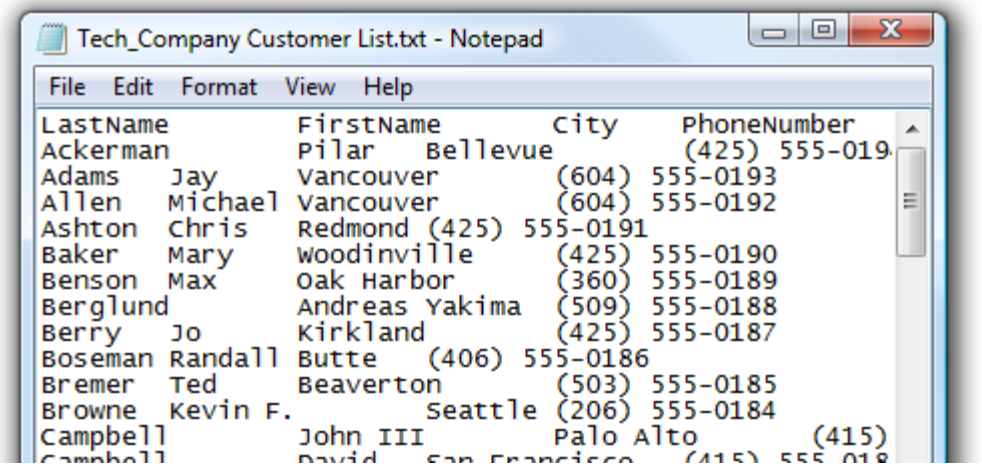
Fixed-width Text File

- Data in a fixed-width text file is arranged in rows and columns, with one entry per row.
- Each column has a fixed width, specified in characters, which determines the maximum amount of data it can contain.
- No delimiters are used to separate the fields in the file.



Tab-Separated Values (TSV) Text File

- A Tab-Separated Values (TSV) file (also called tab-delimited file) is a simple text format for storing data in a tabular structure, e.g., database table or spreadsheet data, and a way of exchanging information between databases.
- Each record in the table is one line of the text file



Opening a File in Python

- In order to open a file for writing or use in Python, you must rely on the built-in **open()** function.
- The **open()** function takes two parameters; *filename*, and *access_mode*.

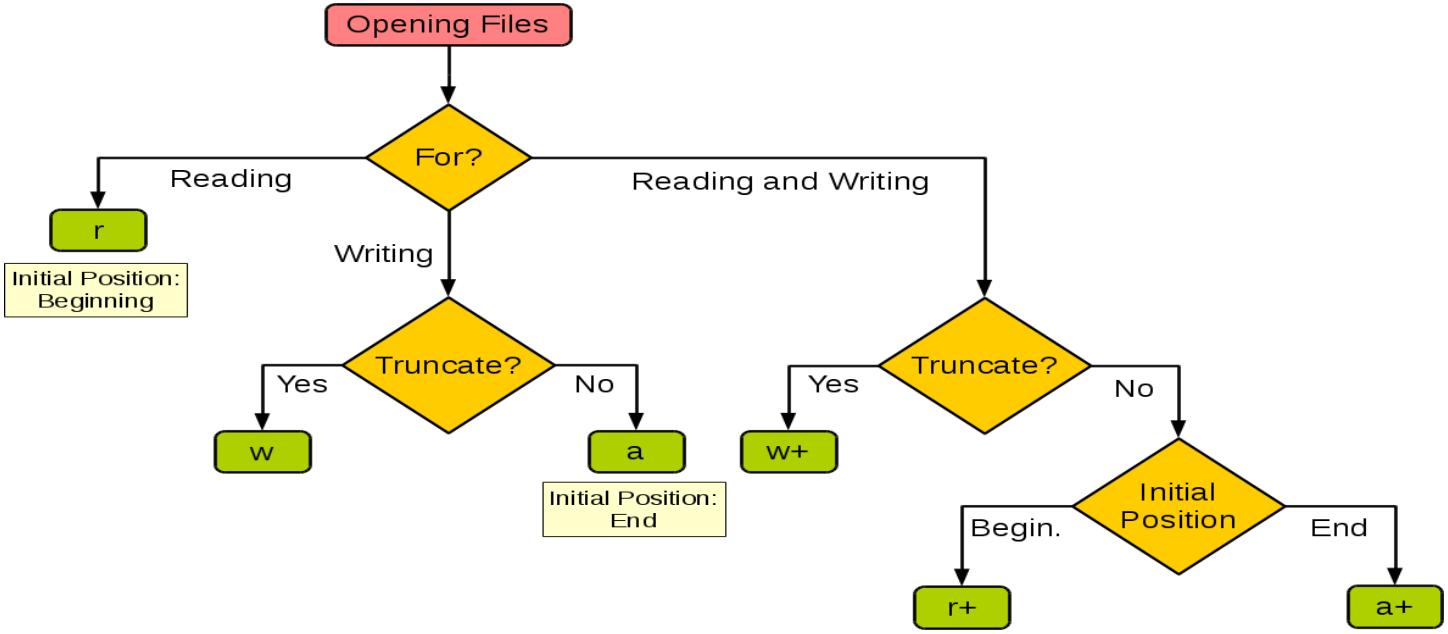
```
open(filename, access_mode)
```

- The access_mode attribute of a file object tells you which mode a file was opened in. And the filename attribute tells you the name of the file that the file object has opened.

File Access Modes

Mode	Function
r	Open file for reading only. Starts reading from beginning of file. This default mode.
rb	Open a file for reading only in binary format. Starts reading from beginning of file.
r+	Open file for reading and writing. File pointer placed at beginning of the file
rb+	Open a file for reading and writing in binary format. File pointer placed at beginning of the file
w	Open file for writing only. File pointer placed at beginning of the file. Overwrites existing file and creates a new one if it does not exists.
wb	Same as **w** but opens in binary mode.
w+	Same as **w** but also allows to read from file.
wb+	Same as **wb** but also allows to read from file.
a	Open a file for appending. Starts writing at the end of file. Creates a new file if file does not exist.
ab	Same as **a** but in binary format. Creates a new file if file does not exist.
a+	Same as **a** but also open for reading.
ab+	Same as **ab** but also open for reading.

File Access Modes Summary



Function	r	r+	w	w+	a	a+
Read	✓	✓		✓		✓
Write		✓	✓	✓	✓	✓
Create new file if not exist			✓	✓	✓	✓
Overwrite existing file			✓	✓		
Pointer place at beginning of file	✓	✓	✓	✓		
Point place at end of file					✓	✓

The File Object Attributes

- Once a file is opened and you have one file object, you can get various information related to that file.
- Here is a list of all attributes related to file object:

Attribute	Description
file.closed	Returns true if file is closed, false otherwise.
file.mode	Returns access mode with which file was opened
file.name	Returns name of the file.

Closing a File in Python

- When we are done with operations to the file, we need to properly close the file.
- Closing a file will free up the resources that were tied with the file and is done using Python **close()** method.
- Python has a garbage collector to clean up unreferenced objects but, we must not rely on it to close the file.

Example

```
f = open("0700.HK.csv", "r")    # Open text file for reading

print ("Name of the file: ", f.name)
print ("Closed or not : ", f.closed)
print ("Opening mode : ", f.mode)

f.close()                       # Close File
print ("File Closed")
print ("Closed or not : ", f.closed)
```

```
Name of the file:  0700.HK.csv
Closed or not :  False
Opening mode :  r
File Closed
Closed or not :  True
```

Exception Handling

- If an exception occurs when we are performing some operation with the file, the code exits without closing the file.

```
f = open("error.txt", "r")    # Open text file for reading

print ("Execute Close statement")
f.close()                          # Close File
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-8-b5c83f3b01c0> in <module>()
----> 1 f = open("error.txt", "r")    # Open text file for reading
      2
      3 print ("Execute Close statement")
      4 f.close()                          # Close File

FileNotFoundError: [Errno 2] No such file or directory: 'error.txt'
```

The try … except Block

- Error handling in Python is done through the use of exceptions that are caught in try blocks and handled in except blocks.
- If an error is encountered, a **try** block code execution is stopped and transferred down to the **except** block.
- In addition to using an except block after the try block, you can also use the **finally** block.
- The code in the finally block will be executed regardless of whether an exception occurs.

The try … except Block (cont.)

- By using **try** ... **finally** block, we are guaranteed that the file is properly closed even if an exception is raised, causing program flow to stop.

```
try:
    f = open("error.txt", "r")
    print ("File open successfully")
except:
    print ("Fail to open file!")
finally:
    print ("Execute finally statement")
```

```
Fail to open file!
Execute finally statement
```

Writing to File

- The **write()** method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.
- The write() method does not add a newline character ('\n') to the end of the string.
- Combine with file access mode, the target file can be overwrite or append

Example: Insert Data to a New File

- By using “w” in access file, the data will be written to a new file (or overwritten)

```
f = open("Testing.txt", "w")

for i in range(5):
    f.write("This is new line %d\n" % (i+1))
```

- Content in file “Texting.txt”

```
This is new line 1
This is new line 2
This is new line 3
This is new line 4
This is new line 5
```

Example: Append Data to Existing File

- By using “a” in access file, the data will be appended to existing file

```
f = open("Testing.txt", "a")

for i in range(3):
    f.write("This is append %d\n" % (i+1))
```

- Content in file “Texting.txt”

```
This is new line 1
This is new line 2
This is new line 3
This is new line 4
This is new line 5
This is append 1
This is append 2
This is append 3
```

Reading from a File

- To read a file in Python, we must open the file in reading mode. There are several ways to read data from a file.

Method	Description
read( )	Return specified number of characters from the file. If omitted it will read the entire contents of the file.
readline( )	Return the next line of the file.
readlines( )	Read all the lines as a list of strings in the file

Read File using readline() method

- If you want to read a file line by line, as opposed to pulling the content of the entire file at once, then you use the **readline()** function.

```
f = open("Testing.txt", "r")
f.readline()              # Read the whole line
```

```
'This is new line 1\n'
```

```
f.readline(8)          # Read the first 8 character of the line
```

```
'This is '
```

Read File using readlines() method

- When you use **readlines()** for reading the file or document line by line, it will separate each line and present the file in a readable format.

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

```
['This is new line 1\n',
 'This is new line 2\n',
 'This is new line 3\n',
 'This is new line 4\n',
 'This is new line 5\n']
```

Read File using read() method

- If you need to extract a string that contains all characters in the file, you can use the **read()** method:

```
f = open("Testing.txt", "r")
f.read(18)   # Read the first 18 characters
```

```
'This is new line 1'
```

```
f = open("Testing.txt", "r")
f.read()    # Read the data to End of File
```

```
'This is new line 1\nThis is new line 2\nThis is new line 3\nThis is new line 4
\nThis is new line 5\n'
```

File Pointer Location

- 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 = open("Testing.txt", "r")
f.read(18)   # Read the first 18 characters
```

```
'This is new line 1'
```

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

```
18
```

```
f.seek(57)   # bring file cursor to initial position
```

```
57
```

```
f.read()    # Read the data to End of File
```

```
'This is new line 4\nThis is new line 5\n'
```

Parsing CSV Files

- The csv library provides functionality to both read from and write to CSV files.
- Designed to work out of the box with Excel-generated CSV files, it is easily adapted to work with a variety of CSV formats.
- The csv library contains objects and other code to read, write, and process data from and to CSV files.

Reading CSV Files with csv

```
import csv

# Open and Read the file by CSV Reader
with open("csv_demo_file.txt") as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    line_count = 0

    for row in csv_reader:
        if line_count == 0:              # Process heading
            print("\t".join(row))
        else:                            # Process record
            print(row[0], "\t", row[1], "\t", row[2], "\t", row[3])

        # Increase the line counter
        line_count += 1

    # Print the number of records
    print("Processed", line_count, "lines.")
```

Reading from a CSV file is done using the reader object. The CSV file is opened as a text file with Python's built-in open() function, which returns a file object. This is then passed to the reader, which does the heavy lifting.

```
Name    Employee ID     Dept    Birthday
John     012345          HR      11/18/1997
Erica    001424          IT      10/24/1991
Processed 3 lines.
```

Reading CSV Files into Dictionary with csv

```
import csv

# Open and Read the file by Dictionary Reader
with open("csv_demo_file.txt", mode="r") as csv_file:
    csv_reader = csv.DictReader(csv_file)
    line_count = 0

    for row in csv_reader:
        if line_count == 0:                # Process heading
            print("\t".join(row))
            line_count += 1

        # Process record
        print(row["Name"], "\t", row["Employee ID"], "\t",
              row["Dept"], "\t", row["Birthday"])
        line_count += 1

    # Print the number of records
    print("Processed", line_count, "lines.")
```

Rather than deal with a list of individual String elements, you can read CSV data directly into a dictionary as well.

```
Name    Employee ID     Dept    Birthday
John     012345          HR      11/18/1997
Erica    001424          IT      10/24/1991
Processed 3 lines.
```

Writing CSV Files with csv

- You can also write to a CSV file using a writer object and the **write_row()** method:

```
import csv

with open("output_file.csv", mode="w") as csv_file:
    writer = csv.writer(csv_file)

    writer.writerow([123, "ABC", "Hello,Word"])
```

```
123,ABC,"Hello,Word"
```

Quoting Optional Parameter

- The *quotechar* optional parameter tells the writer which character to use to quote fields when writing.
- Whether *quoting* is used or not, however, is determined by the quoting optional parameter.

Parameter	Meaning
csv.QUOTE_MINIMAL	.writerow() will quote fields only if they contain the delimiter or the quotechar. This is the default case.
csv.QUOTE_ALL	.writerow() will quote all fields.
csv.QUOTE_NONNUMERIC	.writerow() will quote all fields containing text data and convert all numeric fields to the float data type.
csv.QUOTE_NONE	.writerow() will escape delimiters instead of quoting them. In this case, you also must provide a value for the escapechar optional parameter.

csv.QUOTE_MINIMAL

- The function ***writerow( )*** will quote fields only if they contain the delimiter or the quotechar. This is the default case.

```
import csv

# Define FileName
FileName = CoLabPath + "csv_output.csv"

# open and Write the File
with open(FileName, "w") as csv_file:
    writer = csv.writer(csv_file, delimiter=",",
                        quotechar="'",
                        quoting=csv.QUOTE_MINIMAL)

    # Quote fields only if they contain the delimiter or the quotechar (Default)
    writer.writerow([123, 456.78, "ABC", "Hello,Word"])
```

```
!cat "gdrive/My Drive/Colab Notebooks/csv_output.csv"
```

```
123,456.78,ABC,'Hello,Word'
```

csv.QUOTE_ALL

- The function ***writerow( )*** will quote all fields.

```
import csv

# Define FileName
FileName = CoLabPath + "csv_output.csv"

# open and Write the File
with open(FileName, "w") as csv_file:
    writer = csv.writer(csv_file, delimiter=",",
                        quotechar="'",
                        quoting=csv.QUOTE_ALL)

    # Quote all fields
    writer.writerow([123, 456.78, "ABC", "Hello,Word"])
```

```
!cat "gdrive/My Drive/Colab Notebooks/csv_output.csv"
```

```
'123','456.78','ABC','Hello,Word'
```

csv.QUOTE_NONNUMERIC

- The function ***writerow( )*** will quote all fields containing text data and convert all numeric fields to the float data type.

```
import csv

# Define FileName
FileName = CoLabPath + "csv_output.csv"

# open and Write the File
with open(FileName, "w") as csv_file:
    writer = csv.writer(csv_file, delimiter=",",
                        quotechar="'",
                        quoting=csv.QUOTE_NONNUMERIC)

    # Quote all fields containing text data and
    # convert all numeric fields to the float data type
    writer.writerow([123, 456.78, "ABC", "Hello,Word"])
```

```
!cat "gdrive/My Drive/Colab Notebooks/csv_output.csv"
```

```
123,456.78,'ABC','Hello,Word'
```

csv.QUOTE_NONE

- The function ***writerow( )*** will escape delimiters instead of quoting them. You also must provide a value for the escapechar optional parameter.

```
import csv

# Define FileName
FileName = CoLabPath + "csv_output.csv"

# open and Write the File
with open(FileName, "w") as csv_file:
    writer = csv.writer(csv_file, delimiter=",",
                        quotechar="'",
                        escapechar='\\',
                        quoting=csv.QUOTE_NONE)

    # Escape delimiters instead of quoting them.
    # You must provide a value for the escapechar optional parameter.
    writer.writerow([123, 456.78, "ABC", "Hello,Word"])
```

```
!cat "gdrive/My Drive/Colab Notebooks/csv_output.csv"
```

```
123,456.78,ABC,Hello\,Word
```

Writing CSV File from Dictionary with csv

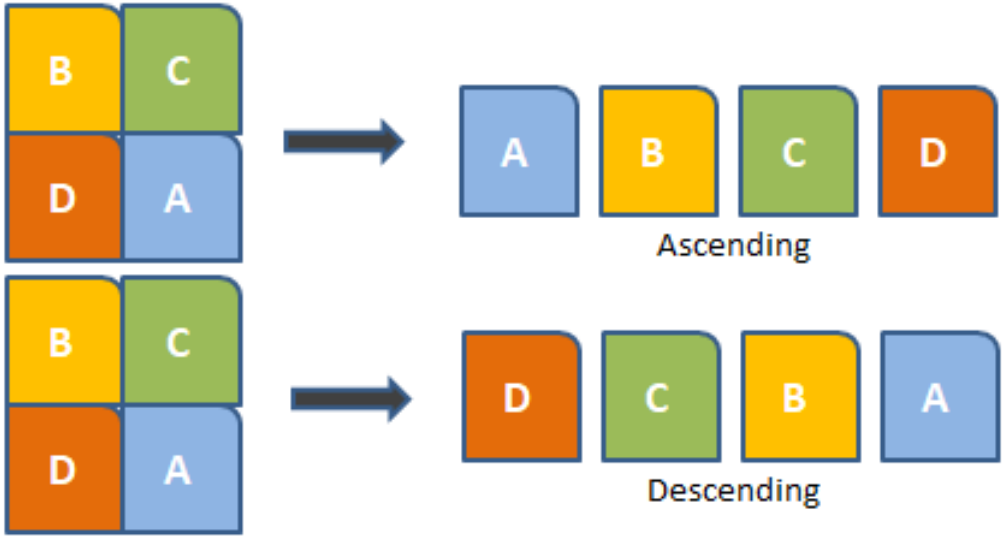
- Unlike DictReader, the fieldnames parameter is required when writing a dictionary. It also uses the keys in fieldnames to write out the first row as column names.

```python
import csv

with open("output_file.csv", mode="w") as csv_file:
    fieldnames = ["Line", "Record", "Data"]
    writer = csv.DictWriter(csv_file, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({"Line": "Line 1", "Record": "Record 1", "Data": "Data 1"})
    writer.writerow({"Line": "Line 2", "Record": "Record 2", "Data": "Data 2A, 2B"})
```

```
Line,Record,Data
Line 1,Record 1,Data 1
Line 2,Record 2,"Data 2A, 2B"
```

Sorting

- Python lists have a built-in ***sort( )*** function that modifies the list in-place.
- There is also a ***sorted( )*** function that builds a new sorted list from an iterable.



Function sort

- The ***sort( )*** function sorts the elements of a given list in a specific ascending or descending order.
- The syntax is:

```
list.sort(key=..., reverse=...)
```

```
# Define the list
MyList = [3, 5, 1, 2, 4]

# Sort the list
MyList.sort()

# Display the list
MyList
```

```
[1, 2, 3, 4, 5]
```

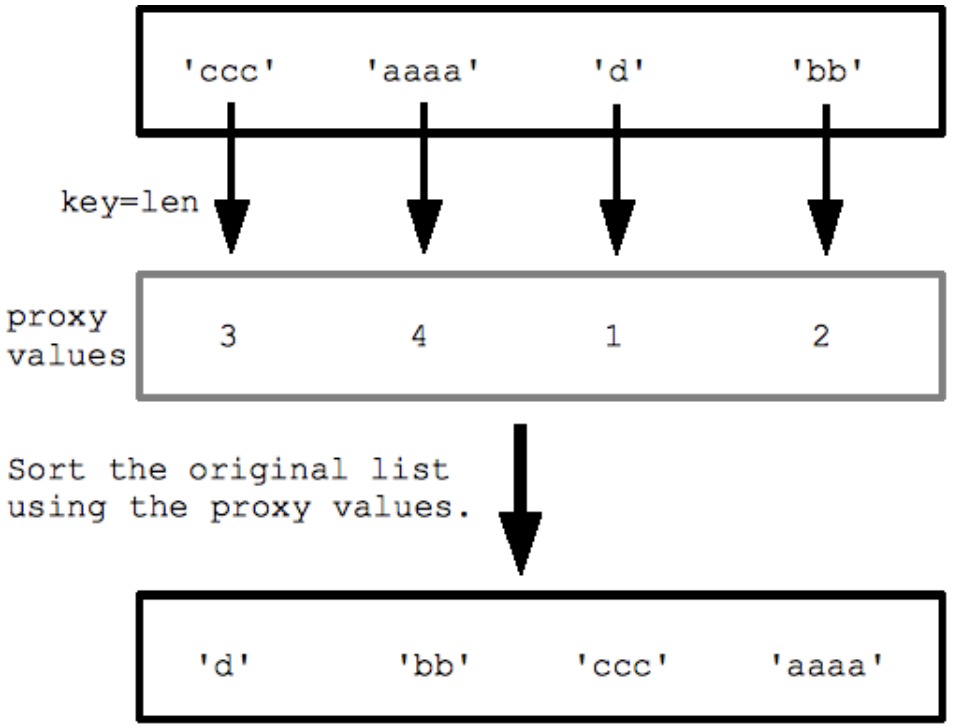
```
# Define the list
MyList = [3, 5, 1, 2, 4]

# Sort the list
MyList.sort(reverse=True)

# Display the list
MyList
```

```
[5, 4, 3, 2, 1]
```

Sort by Key

- The ***sort( )*** method also accepts a key function as an optional parameter. Based on the results of the key function, you can sort the given list.

```
list.sort(key=len)
```



Sort the List by Length of Item

- In order sort the list by the length of each element, you can specify **key=len** during sorting.

```
# Define the list
MyList = ["Apple", "aeroplane", "appreciate", "cat", "Color"]

# Sort the list by string length
MyList.sort(key=len)

# Display the List
MyList
```

```
['cat', 'Apple', 'Color', 'aeroplane', 'appreciate']
```

Sort the list with Case-insensitive

- The sort function is case-sensitive. If you want to sort the data without consider the case, you need to convert the data to upper or lower for sorting.

```
# Define the list
MyList = ["Apple", "aeroplane", "appreciate", "cat", "Color"]

# Sort the list by string length
MyList.sort()

MyList
```

```
['Apple', 'Color', 'aeroplane', 'appreciate', 'cat']
```

```
# Define the list
MyList = ["Apple", "aeroplane", "appreciate", "cat", "Color"]

# Sort the list after convert to upper
MyList.sort(key=str.upper)

MyList
```

```
['aeroplane', 'Apple', 'appreciate', 'cat', 'Color']
```

Function sorted

- The ***sorted( )*** function sorts the elements of a given iterable in a specific order (either ascending or descending) and returns the sorted iterable as a list. The syntax is:

```
sorted(iterable, key=None, reverse=False)
```

- The ***sorted( )*** function can take a maximum of three parameters:
  - **iterable** – A sequence (string, tuple, list) or collection (set, dictionary, frozen set) or any other iterator.
  - **reverse** (Optional) – If True, the sorted list is reversed (or sorted in descending order). Defaults to False if not provided.
  - **key** (Optional) – A function that serves as a key for the sort comparison. Defaults to None.:

Sorting List

- A list also has the ***sort( )*** method which performs the same way as ***sorted( )***.
- The only difference is that the ***sort( )*** method doesn't return any value and changes the original list.

```
Input_List = ['3', '5', '1', '2', '4']

# Sort the list in ascending order
ResultList = sorted(Input_List)

print(ResultList)
```

```
['1', '2', '3', '4', '5']
```

```
Input_List = ['3', '5', '1', '2', '4']

# Sort the list in descending order
ResultList = sorted(Input_List, reverse=True)

print(ResultList)
```

```
['5', '4', '3', '2', '1']
```

Custom Sorting using the key parameter

- The ***sorted( )*** function has an optional parameter called ‘key’ which takes a function as its value.
- This key function transforms each element before sorting, it takes the value and returns 1 value which is then used within sort instead of the original value.

Example: Sort by String Length

- For example, if we pass a list of strings in sorted( ), it gets sorted alphabetically. But if we specify key = len, i.e. give ***len*** function as key, then the strings would be passed to ***len***, and the value it returns, i.e. the length of strings will be sorted. Which means that the strings would be sorted based on their lengths instead

```
# Define the list
MyList = ["Apple", "aeroplane", "appreciate", "cat", "Color"]

# Sort the list by string length
ResultList = sorted(MyList, key=len)

# Display the List
ResultList
```

```
['cat', 'Apple', 'Color', 'aeroplane', 'appreciate']
```

Sorting using Custom Function

- You can define a function as the sorting key, below is an example that sorting for case-insensitive

```
# Define the function
def MyFunction(x):
    NewValue = x.upper()
    return NewValue

# Define the list
MyList = ["Apple", "aeroplane", "appreciate", "cat", "Color"]

# Sort the original list and store the result to new list
NewList = sorted(MyList, key=MyFunction)

# Print out the result
NewList
```

```
['aeroplane', 'Apple', 'appreciate', 'cat', 'Color']
```

Module operator

- Python provides convenience functions to make accessor functions easier and faster.
- The operator module has ***itemgetter( )***, ***attrgetter( )***, and a ***methodcaller( )*** function.

Example

```
import csv
import operator

# Define FileName
FileName = CoLabPath + "sorting_demo.csv"

# Open the File for Read
f = open(FileName, "r")

# Define the delimiter
csv_reader = csv.reader(f)

# Skip the Header
next(f)

# Sort the list by Mark
sortedlist = sorted(csv_reader, key=operator.itemgetter(1))

# Print the row
for row in sortedlist :
    print(row)
```

```
['Terence', ' 75', ' 12']
['John', ' 45', ' 12']
['Alison', ' 50', ' 18']
['David', ' 75', ' 20']
['Jimmy', ' 90', ' 22']
```