UCL

INST0004 Programming 2

Lecture 10: Files & Exceptions Handling - Revision

Module Leader:
Dr Daniel Onah

2023-24


UCL

UCL

Copyright Note

Important information to adhere to

Summary of Previous Lecture

Recap of previous lecture

Re: Cap☺

In the last lecture we looked at ...

- basic concepts of Sets and its constructs
- we discussed the concepts of Mutable and Immutable objects
- we discussed how to implement subsets of a Set
- we discussed the various Set methods and their implementations
- we performed a few implement of Set in Python
- we discussed about the concepts of sorting and searching in Python

Learning Outcomes

The learning outcomes for the lecture

The objective of this week's lecture is to introduce files and exception handling in Python programming. At the end of the lecture, you should be able to:

- understand **how to open and read a text file object**
- understand how to **write to a text file**
- understand more on the various **built-in methods** for file processing and handling
- understand how to **implement** reading and writing to a csv file
- understand how to **implement and handle exceptions** in Python programming
- we conduct a general revision of some of the essential concepts of object-oriented programming covered in the module

Introduction to File Handling

File handling mechanism in Python ...

In this section we will be covering various ways you could read and write to a file in Python. We would also be looking at how to encrypt data. We will also at the end look at how we could write our program to report/capture and recover from any exception error problems. For example, missing files or malformed content using exception handling mechanism in Python programming.

Opening a File

Opening a File in Python ...

Reading and writing to files is a common place in research application and programming. *Examples of text files does not only include simple text editor*, such as files written using Notepad **(text file (txt))**, but also includes **Python source code**, **comma-separated values (CSV)** and **HTML** files and so on. **Remember**, *when opening a file for reading, ensure that such file exist, otherwise you will run into an exception error*.

Opening a File

Opening a File in Python ...

Data processing from files is a very useful skill to have and in many disciplines because it is necessary that we would want to process large data sets, analysed or manipulate stored files. To access a file using **Python programming language**, **we must first *open*** the file. ***When we open a file, we present the name of the file, or if the file is stored in a different directory or folder, the file name should be preceded by the directory path***. **Note:** *that in order to process your file, you should firstly specify whether the file is to be opened for reading or writing.*

Now, suppose we want to read data from a file named ***input.txt***, located in the same directory as our program, we could do this by using the following function call to open the file:

```
dataFile = open("input.txt", "r") # read mode
```

Opening a File *"with"* statement

Opening a File in Python ...

Another way we can open and read a file is by using context manager approach. *This approach start the process of reading the file by using a special* ***with*** **statement** to open the file and **save** it with a file name. The Python *with* statement creates a **runtime context** that allows you to run a group of statements under the control of a ***context manager***. Let's look at an example to help us understand this approach properly.

Using Context Manager approach:

```
# opening the file in read mode using context manager approach.
with open("input.txt", "r") as dataFile:
```

This statement opens the file for reading in a read-mode indicated by the string argument **"r"** and returns a file object that is associated with the file named ***input.txt***.

Opening a File

Opening a File in Python ...

The file object returned by the open function must be saved in a variable (e.g. **dataFile**). All operations for accessing a file are made through the file object. To *open a file for writing*, you ***provide the name of the file as the first argument*** to the *open()* function and the string **"w"** *for writing to the file in a write-mode* as the second argument:

```
output = open("output.txt", "w") # writing to a file
```

Suppose we already have the output file or this already exists, then it will be emptied or content cleaned before the new data is written into it (using the write mode "w"). *However, if the file does not exist, an empty file is created and the content would be written to the file.*

Opening and Closing File

Opening and closing file in Python ...

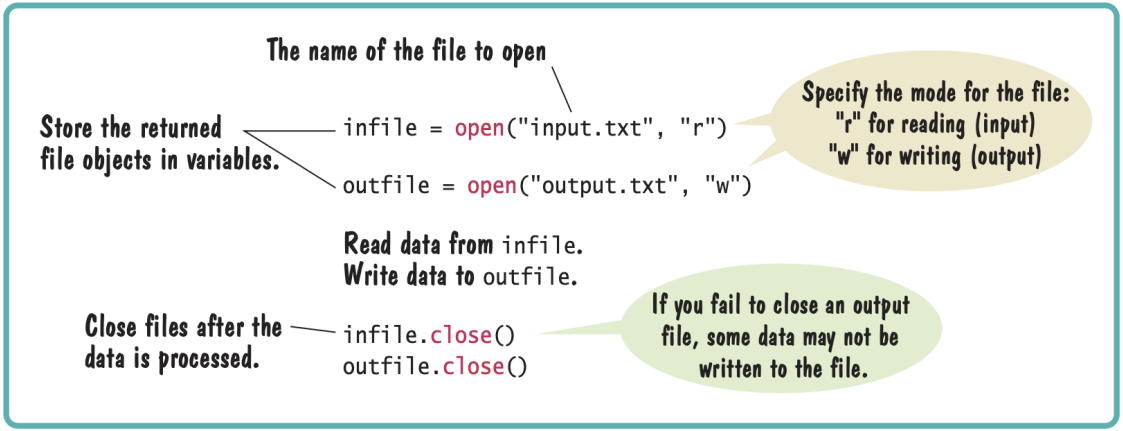
When you are done processing a file, be sure to ***close the file using the close method***.

```
output.close() # close file after process complete
```

After a file has been closed, it cannot be used again until it has been reopened. Attempting to do so will result in an exception.

Opening, Writing and Closing File illustration

Opening,writing and closing file in Python ...



Reading a File

Reading file in Python ...

Reading a file is a very interest process in data science and analysis. To read a ***line of text*** from a file, you have to call the *readline()* method on the file object that was *returned when you opened the file*. This simply means, **use the readline() method to obtain lines of text from a file**.

```
line = inFile.readline() # reading line of text in a file
```

When a file is opened, an input marker is positioned at the beginning of the file. The readline method reads the text, starting at the current position and continuing until the end of the line is encountered. The input marker is then moved to the next line. The readline method returns the text that it read, including the newline character that denotes the end of the line. For example, suppose **input.txt** contains lines of text as illustrated in the next section. *How do we read the lines of text?*

Reading a File

Reading file in Python ...

Reading strings from a text file using the readline() method is very interesting. The mechanism is that if for example we have the following days in a text file:

```
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```

On the first call to **readline()** method, this would return the string “Monday”. Note that the denote the new line character that indicates the end of the line read, which allow us to read the next string in the text file. When we get to the end of the file after the last string, calling the **readline()** method again will return an empty string (“ ”).

Reading Multiple Lines of Text

Reading file in Python ...

Reading multiple lines of text is similar to reading a sequence of values using the **input** function. One way is to continuously reading the line of text repeatedly until any stop or sentinel control value is reached. Let's see an example to help us understand how this is done.

```
#!/usr/bin/env python
"""
A program to demonstrate reading multiple lines of text
"""
line = dataFile.readline()
while line != "": #  empty string returned when end of file reached
    # process the line
    line = dataFile.readline()
```

Reading Multiple Lines of Text - Typecasting

Reading file in Python ...

Remember that the sentinel value is an empty string that is returned as the **readline()** method reaches the end of the file. Similar to the function for the **input** function, the readline() method can only return strings. If we have a data file with numeric data, then we will need to typecast or convert this to integer (int) or floating point (float) literals. For example:

```
1 #!/usr/bin/env python
2 """
3 A program to demonstrate reading typecsting text file data
4 """
5 line = dataFile.readline()
6 output = float(line) # casting the data to floating point literal
```

Note: *When we convert the data type (typecasting) from string to numeric data type, the newline character at the end of the line is ignored during the processing.*

Writing to a File

Writing to a file in Python ...

In order to write to a file in Python, the file must be open and ready to be written to. What this means is that you can only write to a file that has been opened for writing. To write to a file, we invoke the **write()** method to the file object. For example, we can write the string **"Welcome to Python Programming!"** to our output file using the statement construct below.

```
#!/usr/bin/env python
"""
A program to demonstrate writing a string to a file
"""
# writing a string to a file object "output"
output.write("Welcome to Python Programming")
```

Writing to a File

Writing to a file in Python ...

As you already know, the **print** function adds a newline character at the end of its output at the beginning of a new line or to start a new line. In writing to an output file, when starting a new line, we must explicitly write the newline character. The **write()** method takes a single string as an argument and writes the string immediately into the file object. The string is then appended to the end of the file, following any text or data previously written to the file. We can also write formatted strings to a file with the write method. Let's look at an example to help us understand this better.

```
1 #!/usr/bin/env python
2 """
3 A program to demonstrate writing formatted string to a file
4 """
5 output.write("Number of entries: %d\nTotal: %8.2f\n" % (count, total))
```

Processing File

File processing example ...

There are several file processing techniques that could be applied for processing data from any given file format (**.csv**, **.txt**, **.xlsx**, **.html** and so on). Let's look at a typical example of a file. Suppose you are given text file that contains a sequence of numeric values stored per line within the file. You are required to read (process) the file values and write them to a new output file, aligned in column as the appear in the original file. Finally, you are required to compute the total and average of the values.

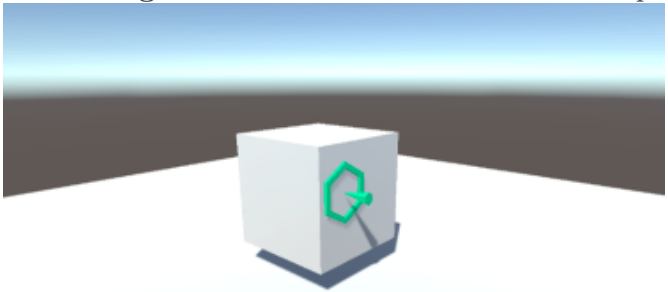
Dataset: Let's name the input file as **data.csv** with the following data.

```
78.3894
62.37
90.124
76.7624
89.906
42.878
```

Processing File

File processing example ...

Note that, whether we create another file called **output.csv** to store the output data or not. The program construct within the code listing will create this output file for us with our selected name in most cases. In this sample, we already created the output.csv file so we could utilise it to store the process file data. After reading the file, we will also want to compute the **total** and **average** of the numeric data stored in the input file.



Processing File

File processing example ...

```
#!/usr/bin/env python
"""
A program to demonstrate file processing
"""
class FileProcessing:
    global total, count
    def __init__(self, inputFile, outputFile):
        self.inputFile = inputFile
        self.outputFile = outputFile
    def openFile(self):
    # the *with open* construct is known as context manager approach
    # this automatically closes the file after ptocessing
        with open(self.inputFile, "r") as inputFile:
            with open(self.outputFile, "w") as outputFile:
                FileProcessing.readWrite(self, inputFile, outputFile)
                FileProcessing.displayOuput(self, inputFile, outputFile)
```

Processing File

File processing example ...

```
17 def readWrite(self, inputFile, outputFile):
18         #reading the file
19         data = inputFile.readline()
20         while data != "":
21             value = float(data)
22             outputFile.write("%0.2f\n"%value)
23             data = inputFile.readline()
```

Processing File

File processing example ...

```
24      def displayOuput(self, inputFile, outputFile):
25          total = 0.0
26          average = 0.0
27          count = 0
28          with open(self.inputFile, "r") as data: # reading the file
29              for row in data:
30                  row = float(row)
31                  total = total + row
32                  count = count + 1
33                  average = total/count
34              print("Total: {} ".format(round(total, 2)))
35              print("Average: {} ".format(round(average, 2)))
```

Processing File

File processing example ...

```
36 # prompt for input and ourput file
37 inputFile = input("Enter the name of the input file: ")
38 outputFile = input("Enter the name of the output file: ")
39 fileObject = FileProcessing(inputFile, outputFile)
40 fileObject.openFile()
```

We can execute the program above by doing the following:

```
Enter the name of the input file: data.csv
Enter the name of the output file: output.csv
```

Output

```
Total: 440.43
Average: 73.4
```

Iterating over File Data

Iterating over the rows of a file ...

As you have notice in the previous example, within the **displayOutput()** method, we use a for loop to iterate over the data file. In file processing, we can also iterate over a file object to read the lines or rows of text within the file. In this section we will demonstrate how to navigate complex data files and challenges.

You have seen how easy it is to read a file line by line. However, Python can also treat a single file as a container of strings for which each individual lines can be represented as a string. We can **read this line or row by using a for-loop construct to iterate over the file object**.

Iterating over File Data

Iterating over the rows of a file ...

Let's see an example to help us understand what loop does. You would notice that the loop below will read all lines or rows from a file and display the outcome.

```
for row in dataFile:
    print(row)
```

At the beginning of each iteration, the loop variable ***row*** is assigned a string that contains the next *row* of text in the file. Within the body of the loop, we simply process the line of text in each row. Within the loop, we display the *row* to the console or terminal/command prompt.

However, there is a difference between a storage file (or file) and a container. In the case of the file, once this is been read, you cannot iterate over the file again without first **closing** and **reopening** the file.

Iterating over File Data - Displaying Text

Iterating over the rows of a file ... displays blank lines

You would noticed that when the rows of the input data is read and displayed, they will have a blank line between each word as follows:

```
Danny

Elaine

Jane
```

How the print() function works here is that, *it prints the first argument then starts a new line by printing a newline character before beginning the process of reading and displaying the second argument iteratively, and so on*. And so, because *each line ends with a newline character, the second newline creates a blank line in the output*.

Removing Newline Character from File - *rstrip()*

Iterating over the rows of a file ... removing the newline characters

Ideally, the blank lines or newline character should be removed before the input string is process and used fully. This is what happen when we read the text file, when the first line of the text file is read, the string row contains the newline character as shown below:

```
|D|a|n|n|y|\n|
```

In order for us to remove the newline character ($\backslash n$) at the end of the text, we need to use a string built-in method called **rstrip**.

```
line = row.rstrip()
```

This would produce a new string as follows:

```
|D|a|n|n|y|
```

Removing Newline Character from File

Iterating over the rows of a file ... removing the newline characters

For example reading a text file (**data.txt**) as seen below would display output with blank spaces as illustrate in the previous section.

```
#!/usr/bin/env python
"""
A program to remove newline or blanks from text file
"""
data = open("data.txt", "r")
for row in data:
    print(row)
```

Removing Newline Character from File

Iterating over the rows of a file ... removing the newline characters

In order to remove the newline character or the blank space between each string, we add the **rstrip()** method to the variable of the for-loop construct.

```
#!/usr/bin/env python
"""
A program to remove newline or blanks from text file
Using the rstrip() method
"""
data = open("data.txt", "r")
for row in data:
    row = row.rstrip()# remove the blank spaces
    print(row)
```

Removing Newline Character from File

Iterating over the rows of a file ... removing the newline characters

By default, the **rstrip()** method creates a new string in which all white space (blanks, tabs, and newlines) at the end of the string has been removed. For example, if there are two blank spaces following the name Danny in the first line of the text file.

```
|D|a|n|n|y|  |  |\n|
```

The **rstrip()** method will remove not only the newline character but also the blank spaces. The outcome would be as follows:

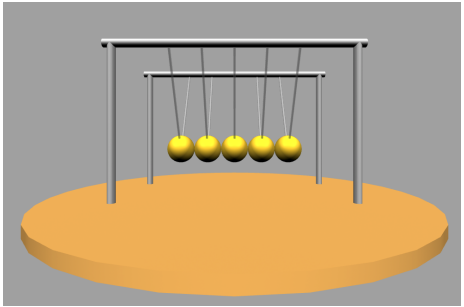
```
|D|a|n|n|y|
```

Removing Specific Character from File - *rstrip()*

Iterating over the rows of a file ... removing the newline characters

In order for us to remove specific characters such as: **#, ?, ., @$** and so on, from the end of string, we can pass the string argument containing those characters into the **strip()** method. Let's see an example, suppose we want to remove a ***hashtag*** or a ***question mark*** symbols or characters from the end of a string, we can use the following command:

```
row = row.rstrip("#?")# remove special characters
```



Removing Specific Character from File - *rstrip()*

Iterating over the rows of a file ... removing the newline characters

There are additional string methods that can be used to strip characters from a string. Let's take a look at some of these methods and how they are used for processing text files.

Methods for Stripping Characters	
Method	**Returns**
s.lstrip()	*This returns a new version of **s** in which white space (blanks, tabs, and newlines) are removed from the left (the front side) of s.*
s.lstrip(chars)	*If characters are provided within the string, they are removed instead of white space.*
s.rstrip() *s.rstrip(chars)*	*Same as **lstrip()** method, except that characters are removed from the right (towards the end) of **s**.*
s.strip() *s.strip(chars)*	*This behaves Similar to the **lstrip()** and **rstrip()** methods, except that the characters/spaces are removed from the front and end sides of **s**.*

Reading Words

Reading individual word in a text file ...

In file processing, sometimes we may want to read each individual words from a text file. Let's say for example, suppose we have a file (input file) which contains four lines of text as follows:

```
Old MacDonald had a farm
Ee i ee i o
And on his farm he had some cows
Ee i ee i oh
```

As there is no method for reading a word from a file, we first read a line and then split in into an individual words. We can use the **split()** method for this. The split method returns the list of *substrings* that *results from splitting the string at each blank space between the words*. Now, create a text file called **rhyme.txt** with the text above. To execute this program, you would need to type the name of the file as follows:

```
Enter the text file to read: rhyme.txt
```

Reading Words

Reading individual word in a text file ...

Remember that the blank spaces are not included as part of the substrings. the only act as the delimiters for where the string would be splitted. After splitting the string, we can now iterate over the list of *substrings* to print the individual words.

```
for aWord in words:
    print(aWord)
```

Also, just to mention that the **split()** method treats consecutive blank spaces as a single delimiter, so even if we have multiple spaces between the words (some or all of the words) such as:

```
|O|l|d| | | |M|a|c|D|o|n|a|l|d| | |h|a|d| |a| |f|a|r|m| |
```

The outcome would still result in the same five substrings as follows:

```
"Old" "MacDonald" "had" "a" "farm"
```

Reading Words

Reading individual word in a text file ...

By default, the **split()** method uses the white space characters as the delimiter. However, we can still use a different character delimiter of our choice. For example, suppose we have words separated using colon (**:**), instead of blank spaces as follows:

```
|O|l|d|:|M|a|c|D|o|n|a|l|d|:|h|a|d|:|a|:|f|a|r|m|
```

In this case, we can confidently specify the colon (:) as the delimiter to be used within the **split()** method.

```
for word in readWords:
    words = word.split(":")
```

The outcome would still result in the same five substrings as follows:

```
"Old" "MacDonald" "had" "a" "farm"
```

Reading Words

Reading individual word in a text file ...

However, note that when a delimiter is passed as an argument with a **split()** method, any other consecutive delimiters will not be treated or considered as been treated within the program. Therefore, consecutive delimiter cannot be treated as a single one, as same with the case where no argument is supplied. Thus, if we try to split the string construct below the outcome would be different.

```
|O|l|d|:|M|a|c|D|o|n|a|l|d|:|:|h|a|d|:|a|:|f|a|r|m|
```

This would result to the following outcome with six strings displayed, with an empty string corresponding to the **"word"** between the two consecutive colons:

```
"Old" "MacDonald" "" "had" "a" "farm"
```

Reading Words - *split()*

Reading individual word in a text file ...

Example of additional string methods that can be used to split words from a string.

Methods for Splitting Words	
Method	**Returns**
s.split()	*This returns a list of words from string* ***s****.*
s.split(sep)	*If the string* ***sep*** *is passed-in, it is used as the delimiter, otherwise, any white space character would be used*
s.split(sep, maxsplit)	If *maxsplit* is passed as one of the arguments, then only that number of splits will be made, resulting in at most *maxsplit + 1* words.
s.rsplit(sep, maxsplit)	*This behaves Similar to the* ***split()*** *method, except that the splits are made starting from the end of the string instead of from the beginning or front of the string.*
s.splitlines()	*This returns a list containing the individual lines of a string split using the newline character \n as the delimiter.*

Reading Words

Reading individual word in a text file ...

```
#!/usr/bin/env python
"""
A program to demonstrate reading words from a file
"""
def main():
    text = input("Enter the text file to read: ")
    readWords = open(text, "r")
    for word in readWords:
        word = word.rstrip() # remove empty space
        words = word.split() # split each words in the text
        for aWord in words:
            aWord = aWord.rstrip(".,?!") # remove characters
            print(aWord)
    readWords.close()
# start program
main()
```

Reading Characters

Reading individual character in a text file ...

Instead of reading the entire row of text, we can read individual characters as well with the **read()** method. The read() method takes a single argument that specifies the number of characters to read. The method returns a string containing the characters. When supplied with an argument of **1**.

```
char = textFile.read(1)
```

In this case, the read method returns a string consisting of the next character in the file. If the end of the file is reached, it returns an empty string (“”).

Reading Characters

Reading individual character in a text file ...

Let's look at an example to illustrate reading individual character from a file. First, create a file **character.txt** with some text. Ensure the file is in the same directory or folder as your Python source code script.

```
1  #!/usr/bin/env python
2  """
3  A program to demonstrate reading a single character from file
4  """
5  inputFile = open("character.txt", "r")
6  char = inputFile.read(1)
7  while char != "":
8      # processing character
9      char = inputFile.read(1)
10     print(char)
```

Note that the **read()** method read and return the newline characters that terminate the individual lines as they are encountered.

Character Counting - Example

Counting individual character in a text file ...

Suppose we are planning to count the occurrence of the characters in a text file. In order to do this, we should use a list of **26** counters representing the letters of the alphabet. Let's look at an example of how we could count the characters. One way is to use a list of 26 counters represented by integer values.

```
# this creates a list with 26 elements intialised to 0.
countChar = [0] * 26
```

In this case, the number of occurrence of the alphabet **"A"** or **"a"** would be represented in **counts[0]**, the count for alphabet **"B"** or **"b"** will be **counts[1]** and so on to **counts[25]** which would represent alphabet **"Z"** or **"z"**.

Character Counting - Example

Counting individual character in a text file ...

Let's use the **ord** function to return the Unicode value for each of the alphabet instead of using conditional statements such as *if*. Note that the Unicode value for alphabets in sequential order are represented using the following values, for example for alphabet **"A"**, the code is 65, for **"Z"** the code is 90. Now, if we subtract the code for alphabet **"A"** for example, we can obtain a value between 0 and 25 that we could used as the index for the character count (**countChar**) list.

```
code = ord(char) - ord("A")
countChar[code] = countChar[code] + 1
```

Note that all lowercase alphabets must be converted to uppercase before they are counted.

Character Counting - Example

Counting individual character in a text file ...

Let's look at an example to help us understand how the character counting program can be constructed.

```
#!/usr/bin/env python
"""
A program to demonstrate counting characters in a text file
"""
countChar = [0] * 26
textFile = open("character.txt", "r")
char = textFile.read(1)
while char != "":
    char = char.upper() # convert character to uppercase
    # ensure the character is an alphabet
    if char >= "A" and char <= "Z":
        code = ord(char) - ord("A")
        countChar[code] = countChar[code] + 1
textFile.close()
```

Reading File Records

Reading collection of records from file ...

In file processing, sometimes, we can be presented with a file that contains a collection of data records. Here each record might consists of multiple fields. For example, a student data file might consist of records with **student identification number**, **first name**, **last name**, **address**, **course**, **entry year** and so on. Also, a file containing bank account transaction may contain records such as **transaction date**, **deposit**, **description**, and **amount**. Normally, when working with files that contain data records, ***you generally have to first read the entire record before you can process the file***.

Pseudocode:

```
For each record in the file
    Read the entire record.
    Process the record
```

Reading File Records - *Example*

Reading collection of records from file ...

Let's look at an example on how we could read the data of a record. In this task, we are required to use a **readline()** method and a *while-loop* construct to check for the end of the file using a sentinel control value.

Let's create an **EmployeeRecord.txt** with the following record:

```
Danny Onah
190293001
Elaine Pang
201923203
Jane Sinclair
392019456
```

Reading File Records - *Example*

Reading collection of records from file ...

```
#!/usr/bin/env python
"""
A program to demonstrate reading data from a record
"""
dataFile = open("EmployeeRecord.txt", "r")
# Read the first field of the first record (name)
data = dataFile.readline()
while data != "": # Check for end of file
    employeeName = data.rstrip() # Remove the \n character
    data = dataFile.readline() # Read the second field
    # Convert to an integer. The \n character is ignored
    employeeNumber = int(data)
    # continue processing the data record
    data = dataFile.readline() # Read the first field of the next record
    print("Employee Name: ", employeeName, ",",  "Employee ID: ",
    employeeNumber)
```

Reading File Records - *Example*

Reading collection of records from file ...

Note that the first field of the record has to be obtained as the **priming read** in preparation in case the file contains no records. *Once, we are inside the loop construct, the remaining fields of the record are then read from the file.* Then the *newline character is stripped from the end of the string fields, and strings containing numerical fields are converted to their appropriate data type (e.g., **int**).* At the end of the loop body, the first field of the next record is obtained as the **modification read**.

There are two methods for reading an entire file. The call **read()** method returns a string with all characters in the file. The **readlines()** method reads the entire contents of a text file into a list.

Reading File Records - *Example Delimited*

Reading collection of records from file ...

Suppose the same file is created using a delimiter (:), how would we approach this. Let's say, we have the text file in this format below:

```
Danny Onah: 190293001
Elaine Pang: 201923203
Jane Sinclair: 392019456
```

We can extract the record with the help of a split() method.

Reading File Records - *Example Delimited*

Reading collection of records from file ...

```
#!/usr/bin/env python
"""
A program to demomnstrate reading a record with delimited contents
"""
fileData = open("EmployeeRecord2.txt", "r")
dataFile = fileData.readlines()
for row in dataFile:
    data = row.split(":")
    employeeName = data[0]
    employeeNumber = data[1]
    print("Employee Name:", employeeName, "," , "Employee Number: ",
    employeeNumber)
fileData.close()
```

Working with CSV files - *Example Delimited*

Reading CSV file ...

You have seen how to read and write text files and to process data stored in various formats, but what if you *need to process data stored in a spreadsheet or csv format?* How would you read and process such file formats. For example, *suppose you need to display a list of movies released in the **1990s** from a spreadsheet or csv file with movie contents as follows*:

Movie	Year	Director
American Beauty	1999	Sam Mendes
Airport 1975	1974	Jack Smight
Bitter Moon	1992	Roman Polanski
The Godfather	1972	Francis Ford Coppola
Fallen Angels	1995	Wong Kar-wai
Awakenings	1990	Penny Marshall
Traffic	2000	Steven Soderbergh
Hamlet	1996	Kenneth Branagh

Working with CSV files - *Example Delimited*

Reading CSV file ...

Most spreadsheet applications store their data in proprietary file formats that cannot be accessed directly by other programs. Fortunately, most can save a copy of the data in a portable format known as CSV (**Comma-Separated Values**). A CSV file is simply a text file in which each row of the spreadsheet is stored as a line of text. The data values in each row are separated or delimited by commas (**,**). For example, the CSV file created from the spreadsheet shown above would contain data represented in the following format:

```
"American Beauty","1999","Sam Mendes"
"Airport 1975","1974","Jack Smight"
"Bitter Moon","1992","Roman Polanski"
"The Godfather","1972","Francis Ford Coppola"
"Fallen Angels","1995","Wong Kar-wai"
"Awakenings","1990","Penny Marshall"
"Traffic","2000","Steven Soderbergh"
"Hamlet","1996","Kenneth Branagh"
```

Working with CSV files - *Example Delimited*

Reading CSV file ...

Comma-Separated Values (CSV) files are so common used for file processing tasks that the Python standard library provides tools for working with them. Let's explore the **csv** module and how to work with *CSV* files in Python. Viewing the above movies data in a csv format would look as follows:

```
American Beauty,1999,Sam Mendes
Airport 1975,1974,Jack Smight
Bitter Moon,1992,Roman Polanski
The Godfather,1972,Francis Ford Coppola
Fallen Angels,1995,Wong Kar - wai
Awakenings ,1990,Penny Marshall
Traffic,2000,Steven Soderbergh
Hamlet,1996,Kenneth Branagh
```

Reading CSV files

Reading CSV file ...

In order for us to read a CSV file, we will need to first open the file the same way we open a text (**.txt**) file.

```
dataFile = open("movies.csv")
```

Now, the next step is to import a CSV module to create a CSV reader using the reader() function.

```
from csv import reader
csvDataReader = reader(dataFile)
```

We can use a *for-loop* to iterate through the data in the CSV reader object. During each iteration of the **loop**, the data for one complete **row** is read from the file and stored in the loop variable row as a list of strings.

```
for row in csvDataReader:
    print(row)
```

Working with CSV files - *Example Delimited*

Reading CSV file ...

Let's look at how we could complete the original task of displaying the **titles** of all *movies* that were released in the ***1990s***. Create a file called ***movies.csv*** with the *movie* contents.

```
#!/usr/bin/env python
"""
A program to demonstrate reading csv file
"""
# import the csv module
from csv import reader
# open the csv file and create a CSV reader
dataFile = open("movies.csv")
csvDataReader = reader(dataFile)
# Read the rows of data
for row in csvDataReader:
    year = int(row[1])
    if year >= 1990 and year <= 1999:
        print(row[0])
```

Working with CSV files - *Example Delimited*

Reading CSV file ...

We can create and write to a CSV file just as we did with a text (.txt) file. Here, we first create a new csv file using the open function as follows:

```
dataFile = open("newMovies.csv", "w")
```

Then create a CSV writer using the writer() function from the csv module.

```
from csv import writer
csvWriter = write(dataFile)
```

To add a row of data to the CSV file, use the writerow() method. You can pass a list of the row's data to this method. For example, to **add a row of column headers**, you could pass a *list of strings or mixture of numbers*, one for each column in the file.

```
csvWriter.writerow(["Movies", "Year", "Director"]
```

Working with CSV files - *Example Delimited*

Reading CSV file ...

You can add numbers or a mixture of text and numbers to the csv file.

```
csvWriter.writerow(["The Godfater", 1972, "Francis Ford Coppola"]
```

To skip a row in the CSV file, you can pass an empty list to the **writerow()** method.

```
csvWriter.writerow([])
```

Remember to close the file after writing to it.

```
dataFile.close()
```

Working with CSV files

Reading CSV file ...

Let's look at an example to help us illustrates how we could read and write to a **csv** file. Suppose you are given the task of creating a new CSV file that contains a limited amount of information from a much larger collection of movie data. Specifically, the new file (**filteredMovies.csv**) should contain only the movie title, year of release, and list of the directors for those movies released in the 1990s. If the input CSV file contains the 3 columns then the new file should contain only the first and the last column of data and only those rows that contain movies from the 1990s, with the appropriate column headings.

Working with CSV files - *Example Delimited*

Reading CSV file ...

Let's look at an example to help us process the described case study in the previous section.

```
#!/usr/bin/env python
"""
A program to process a csv file and filter information needed
"""
from csv import reader, writer
# open the two csv files
dataFile = open("movies.csv")
csvDataReader = reader(dataFile)

outputDataFile = open("filteredMovies.csv", "w")
csvWriter = writer(outputDataFile)

# add the list of column headers to the csv file
fileHeaders = ["Movies", "Year", "Directors"]
csvWriter.writerow(fileHeaders)
```

Working with CSV files - *Example Delimited*

Reading CSV file ...

```
16 # skip the row of column headers in the reader
17 next(csvDataReader)
18
19 # filter the rows of data
20 for row in csvDataReader:
21     year = int(row[1])
22     if year >= 1990 and year <= 1999:
23         newDataRow = [row[0], row[2]]
24         csvWriter.writerow(newDataRow)
25 # close file
26 dataFile.close()
27 outputDataFile.close()
```

Exception Handling

Exception handling in Python ...

When you write your program you should anticipate errors. Your role as a programmer is to think of ways you would handle the errors or exceptions. There are two main aspects of dealing with program errors:

- ***detection***
- ***handling***

One example of error handling is when we try to use the **open** function to detect an attempt to read a file that does not exist. In this case, the function would not be able to handle the error as it is. One approach for us to handle this error, is to terminate the program or to request for another file. The **open** function would not be able to decide between those two alternatives. This would need to report the error to another program.

Raising or throwing Exceptions

Raising exceptions...

In Python programming, we use exception handling mechanism to handle. *Exception handling provides a flexible mechanism for passing control from the point of program error detection to handle that can deal with the error.*

In Python programming similar to other programming languages, when error is detected it is very easy to handle and control the errors. Here you would just raise an appropriate exception to handle this. For example, for a bank account program when a customer try to withdraw an amount more than the account balance.

```
if amount > balance:
    # what next?
```

In this case, we first look for a more suitable exception to handle the error problem. Python has a number of standard libraries that could be use to handle all sorts of exception conditions. In order to signal an exceptional condition, you have to use the raise statement to **raise an exception object**.

Raising or throwing Exceptions

Raising exceptions...

In the case of the bank program, amount withdraw more than balance, you could use ArithmeticError exception. Now the big question is that, "Is it proper for the arithmetic error to have a negative balance after the transaction?". It is not really the case, as you know Python can deal with negative numbers. **Is the amount to withdraw an illegal value?** Indeed it is. The program would consider this to be too large. Therefore, the standard python library most appropriate or suitable for this type of error is a **ValueError** exception.

```
if amount > balance:
    raise ValueError("Amount exceed account balance")
```

Raising an Exception

Raising exceptions...

When we raise an exception, execution does not continue or proceed with the next statement but with an **exception handler**.

Syntax `raise` *exceptionObject*

```
if amount > balance :
    raise ValueError("Amount exceeds balance")

balance = balance - amount
```

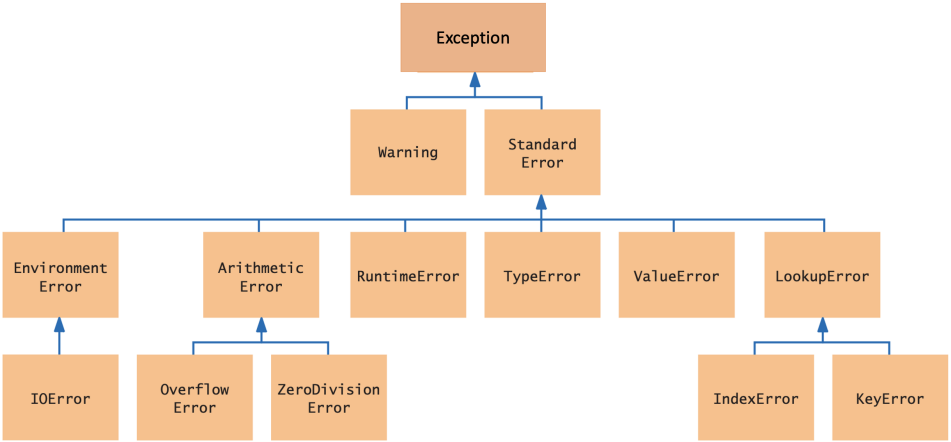
A new exception object is constructed, then raised.

This message provides detailed information about the exception.

This line is not executed when the exception is raised.

Hierarchy of Exception Classes

Raising exceptions...



Handling Exceptions

Exception handling in Python...

Every exception encountered in your program should be handled somewhere within your program constructs. *If an exception has no handler, an error message is displayed, and your program terminates*. Of course, such an **unhandled** exception would be confusing to our program users. In Python programming, we can handle exceptions with the ***try/except*** statement. *In order to do this, you would need to place the statement into a location of your program that knows how to handle a particular exception.* The **try block** contains one or more statements that may cause an exception of the kind that you are willing to handle. Each **except clause** contains the handler for an exception type.

Handling Exceptions

Exception handling in Python...

Let's look at an example to help us understand how this could be done.

```
#!/usr/bin/env python
"""
A program to demonstrate exception handling
"""
try :
    textFile = input("Enter filename: ")
    dataFile = open(textFile, "r")
    row = dataFile.readline()
    value = int(row)

except IOError :
    print("Error: file not found.")
except ValueError as exception :
    print("Error:", str(exception))
```

Handling Exceptions

Exception handling in Python...

Let's look at the description for the previous exception handling example. Two exceptions may be raised in the try block as follows:

- The open function can raise an **IOError** exception if the file with the given name cannot be opened.
- The int function can raise a **ValueError exception** if the string contains any **characters that cannot be part of an integer literal**.

If either of these exceptions is actually raised, then the rest of the instructions in the **try block** are skipped. Here is what happens for the various exception types:

- If an IOError exception is raised, t*hen the except clause for the IOError exception is executed.*
- If a ValueError exception occurs, then the second except clause is executed.

Handling Exceptions

Exception handling in Python...

Each **except clause** contains a handler. When the body of the except **IOError clause** is executed, then some function in the try block failed with an **IOError** exception. In this handler, we simply display an error message indicating that the **file cannot be found**. For the **ValueError** exception handler, when the body of this handler is executed, it displays the message included with the exception. The ***int*** function raises a ValueError **exception when it cannot convert a string to an integer value**. *The function includes a message as part of the exception, which contains the string that it was unable to convert.* For example, if the string passed to the *int* function was ”49x7”, then the message included with the exception will be:

```
invalid literal for int() with base 10: '49x7'
```

- If any other exception is raised, **it will not be handled by any of the except clauses of the try block**. It remains raised until it is handled by another try block.

Handling Exceptions

Exception handling in Python...

Syntax

```
try :
   statement
   statement
   . . .
except ExceptionType :
   statement
   statement
   . . .
except ExceptionType as varName :
   statement
   statement
   . . .
```

This function can raise an IOError exception.

```
try :
   infile = open("input.txt", "r")

   line = inFile.readline()
   process(line)

except IOError :
   print("Could not open input file.")

except Exception as exceptObj :
   print("Error:", str(exceptObj))
```

When an IOError is raised, execution resumes here.

Additional except clauses can appear here. Place more specific exceptions before more general ones.

This is the exception object that was raised.

Handling Exceptions - *Message*

Exception handling in Python...

When you raise an exception, you can provide your own message string. For example, when we called ...

```
raise ValueError("Amount exceeds balance")
```

The message of the exception is the string provided as the argument to the constructor by the programmer or the program developer. In these sample except clauses, we merely inform the user of the source of the problem. Often, it is better to give the user another chance to provide a correct input.

Handling Exceptions - *Finally Clause*

Exception handling in Python...

Occasionally, sometimes we may need to take some action whether or not an exception is raised within our program. The finally construct is used mostly to handle this situation. It is important to **close an output file** to ensure that all output is written to the file. In the following code segment, we open an output file, call one or more functions, and then close the file:

```
outfile = open(filename, "w")
writeData(outfile)
outfile.close() # the program execution may never get here.
```

Now suppose that one of the **methods** or **functions** before the last line raises an exception. Then the call to **close is never executed!** You solve this problem by placing the call to close the file inside a finally clause.

Handling Exceptions - *Finally Clause*

Exception handling in Python...

Let's see an example to help us understand how to use the finally clause in Python programming.

```
1 outfile = open(filename, "w")
2 try :
3     writeData(outfile)
4 finally :
5     outfile.close()
```

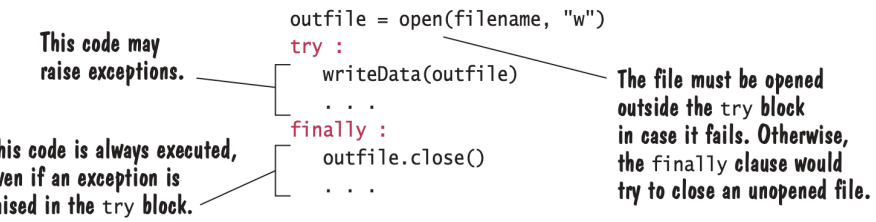
Ideally, in a normal case, there will be no problem. When the **try block is completed**, the finally clause is executed, and the file is **closed**. However, *if an exception occurs, the finally clause is also executed before the exception is passed to its handler*.

Handling Exceptions - *Finally Clause*

Exception handling in Python...

Syntax

```
try :
    statement
    statement
    . . .
finally :
    statement
    statement
    . . .
```




Summary

Summary

Let's revise the concepts of today's lecture

In this lecture we discuss the following ...

- We looked at the various approaches to file handling
- *we discussed the process of opening, reading and writing to a file object*.
- ***we looked at various built-in methods used for file processing***.
- we looked at the approaches that could be used for, reading and writing to a csv file objects
- we discussed about error and exception handling in Python.

Summary

Let's revise the concepts of today's lecture

In this lecture we discuss the following ...

- Once a try block is entered, the statements in a finally clause are guaranteed to be executed, whether or not an exception is raised.
- *Place the statements that can cause an exception inside a try block, and the handler inside an except clause.*
- ***When you raise an exception, processing continues in an exception handler.***
- To signal an exceptional condition, use the raise statement to raise an exception object.

Key Points

Key points to remember about file processing...

Processing Files

```
infile = open("input.txt", "r")

for line in infile :
   line = line.rstrip()
   words = line.split()
   fields = line.split(":")

infile.close()

outfile = open("output.txt", "w")
outfile.write("Hello\n")
outfile.write("%10.2f\n" % value)
outfile.close()
```

Remember to close files

Object Oriented Paradigm

Let's revise the a few concepts

We looked at the following: Object Oriented programming is all about modularity ...

- Data + Methods = Object
- Classes provide a specification of objects
- Instances are real concrete realization of classes
- Use the keyword **class** to define classes
- Use **.** with **class name to access class functions and instance variables** in an object instance
- Python uses the special name **__init__** for the constructor because its purpose is to **initialize an instance of the class**

Information Hiding & Encapsulation

Key points to remember about information hiding and encapsulation...

Information hiding: This is when you separate the description of how to use a class from the implementation details such as how the class method is defined. This is done so that programmers who use the class does not need to know the implementation details of the class definition. Information hiding is another way of avoiding information overloading.

Abstraction: Abstraction is another term for information hiding. When you abstract your class you are hiding and discarding some of the class details.

Information Hiding & Encapsulation

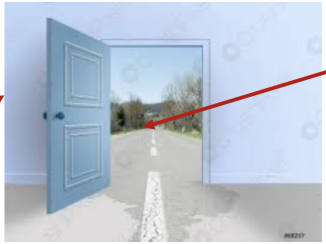
Key points to remember about information hiding and encapsulation...

Encapsulation: Encapsulation simply means grouping program application into unit in such a way that it is easy to use because there is a well-defined simple interface. Encapsulation and information hiding are two sides of the same coin. In Python class definition details are hidden with the use of a special private modifier (**data abstraction** - double or single underscore) before the fields and methods. *Encapsulation in a simple term means that the data and actions are combined in single item (as a class object) and that the details of the implementation are hidden*. The terms information hiding and encapsulation deal with the same general principle.

Illustrating Information Hiding & Encapsulation

Key points to remember about information hiding...

Ensure visibility: Is this a good practice?
NO



When your fields and methods are public, you are not enforcing Encapsulation, therefore, your program might be vulnerable to out accessibility.

Open door house

Illustrating Information Hiding & Encapsulation

Key points to remember about information hiding...

Enforce encapsulation:
Is this a good practice?
YES

Need a key to open door

Idea of encapsulation requires specific accessor (getter) and mutator (setter) methods to use to access the private fields within our program

Super() Constructor of superclass

Example of super() constructor & override method...

```
#!/usr/bin/env python
"""
A program to demonstrate the superclass constructor & override
"""
class BankAccount:
    def __init__(self, initialBalance):
        self.initialBalance = initialBalance

    def deposit(self, amount):
        self.amount = amount
        #return print("Initial balance: ", amount) # uncomment to see
    the output
```

Super() Constructor of superclass

Example of super() constructor & override method...

```
#!/usr/bin/env python
"""
A program to demonstrate the superclass constructor & override
"""
from BankAccount import BankAccount
class CheckingAccount(BankAccount):
    def __init__(self, initialBalance):
    # call the superclass constructor using the super() keyword
        super().__init__(initialBalance)
        self._transactions = 0 # instance variable added in subclass

    def deposit(self, amount): # method overrides superclass method
        super().deposit(amount) # calls superclass method
        transRewards = amount + self._transactions
        print("Current balance plus transaction rewards is:   {}".format
    (round(transRewards, 2)))
```

Super() Constructor of superclass

Example of super() constructor & override method...

```
16 amount = int(input("Enter amount: "))
17 accountObj = CheckingAccount(amount)
18 accountObj.deposit(amount)
```

Note that the **super()** function is used to provide access to methods and properties or attributes of a parent (superclass) or child (subclass) class. *The **super()** function returns an object that represents the **parent** or **base** class.*

Mutator and accessor

Let's revise the concepts of setter and getter methods...

- Now we know that setter method has a true goal instead of just assigning a new value to the target attributes. It has the goal of adding **extra behavior** to the attributes by enforcing encapsulation using the ***underscore symbol*** before the class instance variable or attributes.
- The addition of this extra behaviour makes the **attribute private** so they could not be modified from outside the class and these can only be access through the **getter** and **setter** methods.
- We want our programs to observe object-oriented programming paradigm by enforcing encapsulation and using the **setter** and **getter** methods as the only means of modifying and retrieving information from the attributes.

Points to Note

Important points to remember in Python programming

Know your Python programming environment:

- Set aside some time to become familiar with the programming environment that you will continue to use for your programming practice.
- A text editor is a program for entering and modifying text, such as a Python program.
- Python is case sensitive. You must be careful about distinguishing between upper- and lowercase letters.
- The Python interpreter reads Python programs and executes the program instructions.
- Develop a strategy for keeping backup copies of your work before disaster strikes.

Points to Note

Important points to remember in Python programming

Building blocks of a simple program:

- A comment provides information to the programmer.
- A function is a collection of instructions that perform a particular task.
- A function is called by specifying the function name and its arguments.
- A method can be called with the **self keyword** or by using the **class name**
- A string is a sequence of characters enclosed in a pair of single or double quotation marks.

Points to Note

Important points to remember in Python programming

Program errors - compile-time and run-time errors:

- A compile-time error is a violation of the programming language rules that is detected when the code is translated into executable form.
- An exception occurs when an instruction is syntactically correct, but impossible to perform.
- A run-time error is any error that occurs when the program compiles and runs, but produces unexpected results.

Further Reading

chapters to find further reading on the concepts

You can read further this week's lecture from the following textbook chapters:

- Python for Everyone (3/e) : **By Cay Horstmann & Rance Necaise** - *Chapter 7 Files and Exceptions*
- Learning Python (5^{th} Edition): **By Mark Lutz** - *Chapter 9 Tuples, Files, and Everything Else, Chapter 24 Exception Coding Details*
- Python Programming for absolute beginner (3rd Edition): **By Michael Dawson** - *Chapter 7 Files and Exceptions: The Trivia Challenge Game*
- Python Crash Course - A hands-on, project-based introduction to programming (2nd Edition): **By Eric Matthes** - *Chapter 10 Files and Exceptions*

Gift of the World!!!

Happy Holidays Everyone!

Best Wishes!!!

Wish you all the very best of luck ...