UNIT IV

FILES, MODULES, PACKAGES

Files: text files, reading and writing files, format operator; modules, packages; Illustrative programs: word count, copy.

FILES:

- A file is a collection of data stored on a secondary storage device like hard disk.
- A file is used for real-life applications that involve large amounts of data.
- Each file is identified by a name and the type of the file. Eg: Sample.doc – here Sample is the file name and .doc specifies the file type is a document.
- Console oriented I/O operations pose two major problems:
    - It becomes cumbersome and time consuming to handle huge amount of data through terminals.
    - They are volatile, i.e: when I/O is done using terminal, the entire data is lost if the program is terminated or computer is turned off.
- Therefore, it becomes necessary to store data on a permanent storage (the disks) and read whenever necessary, without destroying the data.

FILE PATH:

- Files are stored on a storage medium like the hard disk in such a way that they can be easily retrieved as and when required.
- Every file is identified by its path that begins from the root node or the root folder.
- In Windows, the drives (Eg: C:\, D:\, E:\) are the root folders and the file path is also known as pathname.

- Knowing the file path is essential to access the files.

- There are two ways of specifying file path. They are:

    - **Absolute Path** – Contains the entire path information starting from the root directory to the exact location of the file. Eg: ***C:/JIT/Python/Examples/File.py***
    - **Relative path** – Starts with respect to the current working directory and therefore lacks the leading slashes. Eg: the same file specified above is given as ***Examples/File.py*** when the current working directory is Python.

TYPES OF FILES:

- Python supports two types of files as Text files and Binary files

- **Text File** - is a stream of characters that can be sequentially processed by a computer in forward direction. For this reason a text file is usually opened for only one kind of operation (reading, writing, or appending) at any given time. Because text files can process characters, they can only read or write data one character at a time. In Python, a text stream is treated as a special kind of file.

- Depending on the requirements of the operating system and on the operation that has to be performed (read/write operation) on the file, the newline characters may be converted to or from carriage-return/linefeed combinations. Besides this, other character conversions may also be done to satisfy the storage requirements of the operating system. However, these conversions occur transparently to process a text file. In a text file, each line contains zero or more characters and ends with one or more characters

- Another important thing is that when a text file is used, there are actually two representations of data- internal or external. For example, an integer value will be represented as a number that occupies 2 or 4 bytes of memory internally but externally the integer value will be represented as a string of characters representing its decimal or hexadecimal value.

FILES:TEXTFILE

- A textfile is a sequence of characters stored on a permanent medium like a hard drive, flash memory, or CD-ROM.
- A text file is a file containing characters, structured as individual lines of text. In addition to printable characters, text files also contain the nonprinting newline character, \n, to denote the end of each text line. the newline character causes the screen cursor to move to the beginning of the next screen line. Thus, text files can be directly viewed and created using a text editor.
- In contrast, binary files can contain various types of data, such as numerical values, and are therefore not structured as lines of text. Such files can only be read and written via a computer program.

Using Text Files

Fundamental operations of all types of files include opening a file, reading from a file, writing to a file, and closing a file. Next we discuss each of these operations when using text files in Python.

OPENING TEXT FILES

All files must first be opened before they can be read from or written to. In Python, when a file is (successfully) opened, a file object is created that provides methods for accessing the file.

All files must first be opened before they can be used. In Python, when a file is opened, a file object is created that provides methods for accessing the file.

OPENING FOR READING

The syntax to open a file object in Python is

file_object = *open("filename", "mode")* where *file_object* is the variable to add the file object.

To open a file for reading, the built-in open function is used as shown,

input_file=open('myfile.txt','r')

The modes are:

· ‘r’ – Read mode which is used when the file is only being read

· ‘w’ – Write mode which is used to edit and write new information to the file (any existing files with the same name will be erased when this mode is activated)

· ‘a’ – Appending mode, which is used to add new data to the end of the file; that is new information is automatically amended to the end

· ‘r+’ – Special read and write mode, which is used to handle both actions when working with a file

Mode in File

Module	Description
r	Read only
w	mode Write
a	only Appending
r+	only Read and write only

Differentiate write and append mode:

Write mode	Append mode
• It is used to write a string in a file • If file is not exist it creates a new file • If file is exit in the specified name, the existing content will overwrite in a file by the given string	• It is used to append (add) a string into a file • If file is not exist it creates a new file • It will add the string at the end of the old file

File Operation:

- **Open a file**
- **Reading a file**
- **Writing a file**
- **Closing a file**

1. **Open ( ) function:**

   · Pythons built in open function to get a file object.

   · The open function opens a file.

   · It returns a something called a file object.

   · File objects can turn methods and attributes that can be used to collect

   Syntax:

   file_object=open("file_name" , "mode")

1. **Example: (Observation and Record)**

   fp=open("a.txt","r")

   Create a text file

   fp=open ("text.txt","w")

2. **Read ( ) function**

   Read functions contains different methods

   · read() – return one big string

   · readline() – return one line at a time

   · readlines() – return a list of lines

   Syntax:

   file_name.read ()

   2. Example: (Observation and Record)

   fp=open("a.txt","w")

   print(fp.read())

   print(fp.read(6))

   print (fp.readline())

   print (fp.readline(3))

   print (fp.readlines())

a.txt

A file stores related data, information, settings or commands in secondary storage device like magnetic disk, magnetic tape, optical disk, flash memory.

Reading file using looping:

3. **Reading a line one by one in given file (Observation and Record)**

```
fp=open("a.txt","r")
for line in fp:
print(line)
```

4. **Write ( ) function**

This method is used to add information or content to existing file.

Syntax:

```
file_name.write( )
```

4. **Example: (Observation and Record)**

```
fp=open("a.txt","w")
fp.write("this file is a.txt")
fp.write("to add more lines")
fp.close()
```

Output: a.txt

A file stores related data, information, settings or commands in secondary storage device like magnetic disk, magnetic tape, optical disk, flash memory.
this file is a.txt to
add more lines

5. **Close ( ) function**

   It is used to close the file.

   Syntax:

   File name.close()

5. **Example: (Observation and Record)**

```
fp=open("a.txt","w")
fp.write("this file is a.txt")
fp.write("to add more lines")
fp.close()
```

6. **Splitting line in a text line: (Observation and Record)**

```
fp=open("a.txt","w")
for line in fp:
        words=line.split()
print(words)
```

7. **Write a program for one file content copy into anotherfile: (Observation and Record)**

```
source=open("a.txt","r")
destination=open("b.txt","w")
for line in source:
        destination.write(line)
source. close()
destination.close()
```

Output:

Input a.txt	Output b.txt
A file stores related data, information, settings or commands in secondary storage device like magnetic disk, magnetic tape, optical disk, flash memory	A file stores related data, information, settings or commands in secondary storage device like magnetic disk, magnetic tape, optical disk, flash memory

8. **Write a program to count number of lines, words and characters in a textfile: (Observation and Record)**

```
fp = open("a.txt","r")
line =0
word = 0
character = 0
for line in fp:
        words = line . split ( )
        line = line + 1
        word = word + len(words)
        character = character +len(line)
print("Number of line", line)
print("Number of words", word)
print("Number of character", character)
```

Output:

Number of line=5

Number of words=15

Number of character=47

FORMAT OPERATOR

The argument of write has to be a string, so if we want to put other values in a file, we have to convert them to strings. The easiest way to do that is with str:

```
>>> x = 52
>>> fout.write(str(x))
```

An alternative is to use the format operator, %. When applied to integers, % is the modulus operator. But when the first operand is a string, % is the format operator.

The first operand is the format string, which contains one or more format sequences, which specify how the second operand is formatted. The result is a string.

For example, the format sequence '%d' means that the second operand should be formatted as a decimal integer:

```
>>> camels = 42
>>> '%d' % camels
'42'
```

The result is the string '42', which is not to be confused with the integer value 42.

A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

```
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

If there is more than one format sequence in the string, the second argument has to be a tuple. Each format sequence is matched with an element of the tuple, in order.

The following example uses '%d' to format an integer, '%g' to format a floating-point number, and '%s' to format a string:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

The number of elements in the tuple has to match the number of format sequences in the string. Also, the types of the elements have to match the format sequences:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

COMMAND LINE ARGUMENTS

Command-line arguments in Python show up in sys.argv as a list of strings (so you'll need to import the sys module).

For example, if you want to print all passed command-line arguments:

```
import sys
print(sys.argv)  # Note the first argument is always the script filename.
```

sys.argv is a list in Python, which contains the command-line arguments passed to the script. With the len(sys.argv) function you can count the number of arguments. If you are gonna work with command line arguments, you probably want to use sys.argv.

To use sys.argv, you will first have to import the sys module.

Example

```
import sys
print 'This is the name of the script: ', sys.argv[0]
print 'Number of arguments: ', len(sys.argv)
print 'The arguments are: ' , str(sys.argv)
```

Output

```
This is the name of the script:  sysargv.py
Number of arguments in:  1
The arguments are:  ['sysargv.py']
```

ERRORS AND EXCEPTIONS, HANDLING EXCEPTIONS

Various error messages can occur when executing Python programs. Such errors are called exceptions. So far we have let Python handle these errors by reporting them on the screen. Exceptions can be "caught" and "handled" by a program, however, to either correct the error and continue execution, or terminate the program gracefully.

WHAT IS AN EXCEPTION?

An exception is a value (object) that is raised ("thrown") signaling that an unexpected, or "exceptional," situation has occurred. Python contains a predefined set of exceptions referred to as standard exceptions.

ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisionError	Raised when division or modulo by zero takes place for all numeric types.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
IndexError	Raised when an index is not found in a sequence.

IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
OSError	Raised for operating system-related errors.
SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
RuntimeError	Raised when a generated error does not fall into any category.

ERRORS,EXCEPTION HANDLING

Errors

· Error is a mistake in python also referred as bugs .they are almost always the fault of the programmer.

· The process of finding and eliminating errors is called debugging

Types of errors

o Syntax error or compile time error

o Run time error

o Logical error

Syntax errors

- Syntax errors are the errors which are displayed when the programmer do mistakes when writing a program, when a program has syntax errors it will not get executed
    - Leaving out a keyword
    - Leaving out a symbol, such as colon, comma, brackets
    - Misspelling a keyword
    - Incorrect indentation

Runtime errors

· If a program is syntactically correct-that is, free of syntax errors-it will be run by the python interpreter

· However, the program may exit unexpectedly during execution if it encounters a runtime error.

· When a program has runtime error it will get executed but it will not produce output

- Division by zero
- Performing an operation on incompatible types
- Using an identifier which has not been defined
- Trying to access a file which doesn‟t exit

Logical errors

· Logical errors are the most difficult to fix

· They occur when the program runs without crashing but produces incorrect result

- Using the wrong variable name
- Indenting a blocks to the wrong level
- Using integer division instead of floating point division
- Getting operator precedence wrong

Exception handling

Exceptions

· An exception is an error that happens during execution of a program. When that Error occurs

Errors in python

· IO Error-If the file cannot be opened.

· Import Error -If python cannot find the module

· Value Error -Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value

· Keyboard Interrupt -Raised when the user hits the interrupt

· EOF Error -Raised when one of the built-in functions (input() or raw_input()) hits an end-of-file condition (EOF) without reading any data

Exception Handling Mechanism

1. try –except
2. try –multiple except
3. try –except-else
4. raise exception
5. try –except-finally

1. **Try –Except Statements**

· The try and except statements are used to handle runtime errors

Syntax:

```
try :
    statements
```

except :

statements

The try statement works as follows:-

- First, the try clause (the statement(s) between the try and except keywords) is executed.
- If no exception occurs, the except clause is skipped and execution of the try statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.

9. **Example: (Observation and Record)**

```
X=int(input("Enter the value of X"))
Y=int(input("Enter the value of Y"))
try:
    result = X / ( X – Y )
    print("result=",result)
except ZeroDivisionError:
    print("Division by Zero")
```

Output:1	**Output : 2**
Enter the value of X = 10 Enter the value of Y = 5 Result = 2	Enter the value of X = 10 Enter the value of Y = 10 Division by Zero

2. **Try – Multiple except Statements**

o Exception type must be different for except statements

Syntax:

try:

statements

```
except errors1:
        statements
except errors2:
        statements
except errors3:
        statements
```

10. **Example (Observation and Record)**

```
X=int(input("Enter the value of X"))
Y=int(input("Enter the value of y"))
try:
        sum = X + Y
        divide = X / Y
        print (" Sum of %d and %d = %d", %(X,Y,sum))
        print (" Division of %d and %d = %d", %(X,Y,divide))
except NameError:
        print(" The input must be number")
except ZeroDivisionError:
        print("Division by Zero")
```

Output:1	Output 2:	Output 3:
Enter the value of X = 10 Enter the value of Y = 5 Sum of 10 and 5 = 15 Division of 10 and 5 = 2	Enter the value of X = 10 Enter the value of Y = 0 Sum of 10 and 0 = 10 Division by Zero	Enter the value of X = 10 Enter the value of Y = a The input must be number

3. **Try –Except-Else**

o The else part will be executed only if the try block does not raise the exception.

o Python will try to process all the statements inside try block. If value error occur, the flow of control will immediately pass to the except block and remaining statements in try block will be skipped.

Syntax:

```
try:
        statements
except:
        statements
else:
        statements
```

11. **Example (Observation and Record)**

```
X=int(input("Enter the value of X"))
Y=int(input("Enter the value of Y"))
try:
        result = X / ( X – Y )
exceptZeroDivisionError:
        print("Division by Zero")
else:
        print("result=".result)
```

Output:1	**Output : 2**
Enter the value of X = 10 Enter the value of Y = 5 Result = 2	Enter the value of X = 10 Enter the value of Y = 10 Division by Zero

4. **Raise statement**

· The raise statement allows the programmer to force a specified exception to occur.

Example:

>>> raise NameError('HiThere')

Output:

Traceback (most recent call last):

File “<stdin>”, line 1, in <module>

NameError: HiThere

- If you need to determine whether an exception was raised but don‟t intend to handle it, a simpler form of the raise statement allows you to re-raise the exception:

Example

try:

... raise NameError('HiThere')

... except NameError:

... print('An exception flew by!')

... raise

Output:

An exception flew by! Traceback

(most recent call last):

File “<stdin>”, line 2, in <module>

NameError: HiThere

5. **Try –Except-Finally**

   - A finally clause is always executed before leaving the try statement, whether an exception has occurred or not.
   - The finally clause is also executed “on the way out” when any other clause of the try statement is left via a break, continue or return statement.

     Syntax

```
try:
        statements
except:
        statements
finally:
        statements
```

12. **Example (Observation and Record)**

```
X=int(input("Enter the value of X"))
Y=int(input("Enter the value of Y"))
try:
        result = X / ( X – Y )
except Zero DivisionError:
        print("Division by Zero")
else:
        print("result=".result)
finally:
        print ("executing finally clause")
```

Output:1	**Output : 2**
Enter the value of X = 10 Enter the value of Y = 5 Result = 2 executing finally clause	Enter the value of X = 10 Enter the value of Y = 10 Division by Zero executing finally clause