UNIT V: FILES, MODULES, PACKAGES

Files and exception : text files, reading and writing files, format operator; command line arguments, errors and exceptions, handling exceptions, modules, packages

Illustrative programs:

* word count
* copy file

FILES INTRODUCTION

* A file is a collection of data that can be stored on a secondary storage device like hard disk.

* A file is said to be persistent (i.e) data stored in permanent storage.

* most of the programs are transient (i.e) they run for a short time and produce some o/p, but when they end, their data disappears.

* one of the simplest ways for programs to maintain their data is by reading & writing text files.

* A file is basically used because real life applications involve large amounts of data and in such situations the console oriented I/o operations pose a major problems.

1) Time consuming to handle huge amounts of data thro' terminals
2) when doing I/o using terminal, the entire data is lost when either the program is terminated or computer is turned off.

* Therefore it becomes necessary to store data on a permanent storage and read whenever necessary.

⇒ Types of files

* python supports two types of files
  - → Text files
  - → Binary files

⇒ Text Files

* A text file is a stream of characters that can be sequentially processed by a computer in forward direction.
* In a text file, each line of data ends with a newline character. Each file ends with a special character called the EOF (End-of-file) marker.
* The data stored in a text-file are in human-readable form. eg:) files with .txt / .py extn

⇒ Binary files

* Binary file may contain any type of data (images, exe, etc), encoded in binary form, for storage and processing.
* The contents of a binary file are not human readable.
* Binary files can be processed sequentially or randomly.

OPENING AND CLOSING FILES

* The file must be opened before reading from or writting to a file.
* The file is opened using the bulit-in open() function.
* syntax

```
fileobj = open ( filename , access_mode )
```

⇒ filename specifies the name of the file that needs to be accessed.

⇒ access_mode indicates the mode in which the file has to be opened (read, write, etc)

⇒ The open() function returns a file object, which is used to read, write or perform any other operation on the file.

⇒ access mode is optional., the default file access mode is 'r' (read)

eg) `file = open ( "a.txt", "r" )`

⇒ opens a file a.txt in read mode.

⇒ Access modes

mode	purpose
r	opens a file for reading only (default mode). File pointer is placed at the beginning of the file.
rb	opens a file for reading only in binary format
r+	read and write
rb+	read and write in binary format
w	opens the file for writing only
wb	write only in binary format
a	opens the file for appending
a+	opens the file for reading & appending
ab	appending in binary format
ab+	appending / reading in binary format

⇒ File object attributes

* when a file is opened file object is returned.
* using the file object attributes information regarding the file can be obtained.

Attribute	information obtained
file obj. closed	returns True if file closed
file obj. mode	mode in which file is opened
file obj name	name of the file

eg)
```
f=open("a.txt","r")
print(f.name)
print(f.mode)
print(f.closed)
```

o/p
```
a.txt
r
False
```

⇒ close() function.

* The close() method is used to close the file object.
* once the file object is closed, the file can't be accessed further.
* syntax

```
fileobj.close()
```

eg)
```
file = open("a.txt","r")
print(file.closed)
file.close()
print(file.closed)
```

o/p: False
True

READING AND WRITING FILES

* read() and write() are used to read data from a file and write data to files respectively.

⇒ read() and readline() method

* The read() method is used to read a string from an already opened file.

syntax.

```
fileobj.read(count)
```

⇒ count specifies the no. of. bytes to be read from the opened file. (optional)

⇒ read() method reads the entire contents of the file till the end.

eg) f.py

```
file = open("a.txt","r")
str = file.read()
print(str)
```

a.txt

```
rmk
rmd
rmkcet
```

o/p

```
rmk
rmd
rmkcet
```

* readline() method is used to read a single line from the file.

eg)
```
file = open("a.txt","r")
print(file.readline())
print(file.readline())
```

o/p
```
rmk
rmd
```

⇒ write() and writelines() method

* The write() method is used to write a string to an already opened file.

* syntax

```
fileobj.write(string)
```

eg)
```
file = open("a.txt","w")
file.write("Hello World!")
file = open("a.txt","r+")
print(file.read())
```

o/p
```
Hello World!
```

* writelines() method is used to write a list of strings

eg)
```
file = open("a.txt","r+")
l = ["Hello","world!"]
file.writelines(l)
print(file.read())
```

O/P
Helloworld!

⇒ Opening files using with keyword

* A file can be opened using the with keyword.
* Advantage
  * The file is property closed after it is used, even if it is not explicitly closed using the close() method.

syntax.
```
with open("filename","access mode") as fileobj:
```

eg)
```
with open("a.txt","r") as file:
    print(file.read())
print(file.closed)
```

I/P: a.txt
grapes apple

O/P: grapes apple
True

ILLUSTRATIVE PROGRAMS

⇒ WORD COUNT

* count the frequency of words in the given file.

program.

```
file = open("a.txt", "r")
a = []
l = file.readlines()
for x in l:
    a.extend(x.split(" "))
d = dict()
for x in a:
    s = x.strip()
    if s not in d:
        d[s] = 1
    else
        d[s] = d[s] + 1
for x in d:
    print(x, d[x])
file.close()
```

Input:

a.txt

banana grapes mango mango
grapes mango
banana
apple

output

banana 2
grapes 2
mango 3
apple 1

* readlines() function

⇒ It is used to read all the lines from the file and return as a list.

readlines() function

It is used to read all the lines from the file and return as a list

COPY FILE

* copy the contents of one file to another file.

program

```
fin = open ("a.txt", "r")
s = fin.read()
print ("contents of input file : \n", s)
fin.close()
Fout = open ("b.txt", "w")
Fout.write (s)
fout = open ("b.txt", "r")
st = Fout.read()
print ("contents of output file : \n", st)
Fout.close()
```

input.

a.txt

```
apple
orange
banana
```

output

```
contents of input file :
apple
orange
banana

contents of output file
apple
orange
banana
```

FORMAT OPERATOR

* % is the format operator.
* when % is applied to integers, it is the modulus operator but when the first operand is a string, % is the format operator.

syntax

"format string" % (tuple of expressions)

* The format string consist of the format specifiers or format sequence
  - %d - decimal integer
  - %f - floating point number
  - %s - string
* The corresponding value for the format sequence is specified as tuple variables to the right of % (format operator)

eg)
```
cars = 52
print ("In july we sold %d cars" % cars)
```

o/p

In july we sold 52 cars

eg). print ("In %d days we made %f million %s"
% (34, 6.1, "dollars")

o/p:

In 34 days we made 6.100000 million dollars

[* By default floating point format prints 6 decimal places]

* The no. of expressions in the tuple has to match the number of format sequences in the string.
* The types of the expressions have to match the format sequences.

eg) print("%d %d %d" % (1,2))

o/p

TypeError: not enough arguments for format string

eg) print ("%d" % 'dollars')

o/p

TypeError: illegal argument type for built-in operation.

* for more formatting,

→ number can be given along with format sequence

eg) >>> "%6d" % 89
' 89'

→ the no. specifies the leading white spaces.

→ If the no is negative, traiing white space

eg)

```
>>> "%-6d" % 89
'89    '
```

→ for floating point nos, the no. of digits after the decimal point can also be specified

eg)

```
>>> "%.2f" % 3.4
'3.40'
```

eg) program to print the contents of a dictionary as a formatted report.

```
d = {'mary': 6.23, 'joe': 5.45, 'josh': 4.25}
for x in d:
    print("%-20s %12.2f" % (x, d[x]))
```

o/p

mary	6.23
joe	5.45
josh	4.25

COMMAND LINE ARGUMENTS

* The sys module in python provides access to any command-line arguments thro' the sys.a

* sys.argv is the list of command-line arg.

* len() can be used to find the no. of. command line arguments.

* sys.argv[0] has the program name

eg) program to illustrate command line arg.

cmd.py

```
import sys
n=len(sys.argv)
s=0
print ("no.of.arguments :",n)
print ("name of the program :", sys.argv[0])
for i in range (1,n):
    s = s+int (sys.argv [i])
print ("sum =",s)
```

o/p.

```
$ python cmd.py 10 20 30
no.of.arguments : 4
name of the program : cmd.py
sum = 60
```

syntax for command line arguments

$ python filename.py arg1 arg2

* Arguments specified in the command prompt while running the program are called command line arguments.

ERRORS AND EXCEPTION

* programmers make mistake when they write programs which is said to be errors / bugs
* The process of tracking them down is called debugging.
* Errors can be
  1) Syntax errors
  2) semantic errors (logical)

1) Syntax errors

* Syntax errors occur when there is something wrong with the structure of the program.
* Syntax errors are discovered by the interpreter when it is translating the source code into byte code.
* It occurs due to poor understanding of language

eg) n=4
```
if(n%2==0)          # syntax error, colon omitted
    print("even")
else:
    print("odd")
```

o/p:

syntax Error : invalid syntax.

2) Semantic / Logical Error

* In semantic/logical error the program may not produce the expected output.
* Logic error occurs due to poor understanding of the problem and its solution.

eg)
```
a = 10
b = 5
print("sum=", a-b)
```

o/p:

sum = 5 # But the expected o/p is 15

⇒ Exceptions

* Exceptions are run-time anomalies or unusual conditions that a program may encounter during execution.

* exception can be categorised as synchronous and asynchronous exceptions.

* synchronous exceptions can be controlled by the program.

eg) divide by zero, array index out of bound, etc

* asynchronous exceptions on the other hand are caused by events that are beyond the control of the program

eg) interrupt from the keyboard, disk failure, etc

* When a program raises an exception, it must handle the exception or the program will be immediately terminated with an error message.

eg) >>> 5/0

ZeroDivisionError: integer division or modulo by zero.

>>> 'Roll' + 123

TypeError: can't concatenate 'str' and 'int' objects.

* Here ZeroDivisionError and TypeError are built-in exceptions.

HANDLING EXCEPTIONS

* Exceptions can be handled in the program using try and except block.

syntax.

```
try:
    statements of try block

except ExceptionName:
    statements of except block.
```

=> A critical operation which can raise exception is placed inside the try block and the code that handles exception is written in except block

* First, try block is executed, if an exception occurs then rest of the statements in the try block are skipped and the corresponding except block is executed

* If there is no exception raised in try block then the except block is skipped.

* If there is no corresponding except block for the raised exception then it is an unhandled exception and the program is terminated with an error message.

9) program to handle the divide by zero exception

```
num = int(input("Numerator:"))
denom = int(input("Denominator:"))
try:
    quo = num/denom
    print ("Quotient:", quo)
except ZeroDivisionError:
    print ("Denominator cannot be zero")
```

O/P (when there is exception)

```
Numerator: 10
Denominator: 0
Denominator cannot be zero
```

O/P (when there is no exception)

```
Numerator: 10
Denominator: 2
Quotient: 5.0
```

* when exception occurs control goes to except block.
* when no exception, except block is skipped.

⇒ multiple except block

* There can be multiple except block for a single try block.

syntax:

```
try:
    statements
except Exception1:
    statements
except Exception2:
    statements.
```

* The except block which matches with the generated exception will get executed.
* A single try statement can have multiple except statements to catch different types of exceptions.

eg) program to illustrate multiple except block.

```
num = input()
denom = input()
try:
    quo = int(num) / int(denom)
    print("quotient:", quo)
except (ZeroDivisionError):
    print("denominator can't be zero")
except (ValueError):
    print("Numerator/Denominator should be int")
```

a single

o/p (ZeroDivisionError)

10
0
denominator can't be zero

o/p (ValueError)

10
a
Numerator/Denominator should be int).

⇒ Multiple Exceptions in a single block

* A single except statement can have multiple exception name specified as a parethesized tuple.

syntax:

```
try:
    statements
except (Exception1, Exception2):
    statements
```

eg)

```
num = input()
denom = input()
try:
  q = int(num) / int(denom)
  print ("quotient:", q)
except( ZeroDivisionError, ValueError):
  print ("check the value!")
```

o/p ⇒ ZeroDivisionError
10
0
check the value

o/p ⇒ ValueError
10
a
check the value

except block without exception name

syntax

```
try:
    statements
except Exception1:
    statements
except :                    # no exception name
    statements.
```

* An except block can be specified without an exception name, but it should be specified as the last except statement after all other except blocks
* This except block can handle any type of exception.

eg)

```
n = input()
d = input()
try:
    q = int(n) / int(d)
    print("Quotient", q)
except ZeroDivisionError:
    print("denominator can't be zero")
except :
    print("check the value!")
```

o/p:

```
10
a
check the value!
```

⇒ The else statement

* The try... except block can optionally have an else statement.
* The else statement should follow all except blocks.
* The statements in the else block is executed only if the try block does not raise any exception.

eg:

```
n=10
d=2
try:
    q=n/d
    print("Quotient:",q)
except ZeroDivisionError:
    print("denominator can't be zero")
else:
    print("No exception raised")
```

o/p:

```
Quotient: 5.0
No exception raised.
```

⇒ The finally Block

* The try block has another optional block finally.
* The finally block are executed irrespective of whether an exception has occured or not
* They are used for clean-up actions.

syntax:

```
try:
    statements
except:
    statements
Finally:
    statements
```

eg)

```
try:
    n = 10
    d = 0
    q = n/d
    print ("Quotient:", q)
except:
    print ("denominator can't be zero")
Finally:
    print ("executes always")
```

o/p:

denominator can't be zero.

executes always

⇒ Handling exceptions in invoked functions

* try.. except block can be placed inside a function definition

eg)

```
def div(n, d):
    try:
        print (n/d)
    except:
        print ("denom is zero")
```

div(10,0)

o/p

10

denom is zero

eg) def div(n,d):
    print(n/d)

try:
    div(10,0)
except ZeroDivisionError:
    print("denominator can't be zero")

o/p
denominator can't be zero

⇒ RAISING EXCEPTIONS

* Exception can be raised deliberately using the raise keyword.

syntax.

raise Exception(args)

⇒ Exception is the name of exception to be raised eg) ValueError.

⇒ args is optional and specifies a value for the exception argument.

eg) try:
    num=10
    print(num)
    raise ValueError
except:
    print("Exception caused intentionally")

o/p
10
Exception caused intentionally

⇒ Exceptions with arguments (instantiating e

* can pass arguments to the exception that is raised deliberately

eg)
```
try:
    raise Exception("Hello","world")
except Exception as eobj:
    print(eobj)
    print(eobj.args)
    arg1,arg2 = eobj.args
    print(arg1,arg2)
    print(eobj.args[0])
```

o/p:

```
("Hello","world")
("Hello","world")
Hello world
Hello
```

eobj

* eobj is of type <class Exception>
* eobj.args is of type <class tuple>

MODULES : REFER UNIT. II for Built-in Modules

* USER DEFINED MODULES

* The user can create his own module.
* modules are pre-written pieces of code that are used to perform a specific task.
* It allows to reuse one or more functions which are not defined in that program.
* Every python program can be considered as a module (i.e) every file with .py extension.

⇒ creating user defined module

* write the piece of code that performs a specific task in a python file (.py extn), which is considered to be the module.
* In the program, the functions / variables from the module can be used after importing as

```
import modulename
        or
from modulename import functionname
```

eg) program to illustrate user defined module

⇒ creating the user defined module:

mymodule.py

```
def large(a,b):
    if(a>b):
        return a
    else
        return b
```

⇒ writing a program, using the module

```
import mymodule
a=int(input())
b=int(input())
max = mymodule.large(a,b)  #invokes the larg
print("maximum:",max)      #fn from user
                           #defined module
```

o/p

```
10
5
maximum:10
```

* The dir() function is used to list the identifiers defined in a module (i.e) functions / variables / classes.

```
>>> import mymodule
>>> dir(mymodule)
['__builtins__', '__cached__', '__doc__', '__file__',
 '__loader__', '__name__', '__package__', '__spec__',
 'large']
```

* Every module has a name. The name of the module can be found by using the __name__ attribute.

eg)
```
print("Hello")
print("Name of this module is :", __name__)
```

o/p

```
Hello
Name of this module is : __main__
```

PACKAGES

* A package is a hierarchical file directory structure that has modules and other packages within it.

* creating packages

  * Every package in python is created as a directory (folder)
  * It must have a special file called __init__.py (This file may not have even a single line of code)
  * This file is added to indicate that this is a python package.
  * Inside this package, modules and other packages can be placed.

* In programs, the packages has to be imported as

```
import packagename.modulename
```

(or)

```
from packagename import modulename
```