SNS COLLEGE OF ENGINEERING

Kurumbapalayam(Po), Coimbatore – 641 107
Accredited by NAAC-UGC with ‘A’ Grade
Approved by AICTE, Recognized by UGC & Affiliated to Anna University, Chennai

Department of Artificial Intelligence and Data Science

Course Name – Computational Thinking and Python Programming

I Year / I Semester

Unit 5-Files

FILES

File is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk).
Since, random access memory (RAM) is volatile which loses its data when computer is turned off, we use files for future use of the data.
When we want to read from or write to a file we need to open it first. When we are done, it needs to be closed, so that resources that are tied with the file are freed. Hence, in Python, a file operation takes place in the following order.

1. Open a file
2. Read or write (perform operation)
3. Close the file

Opening a file

Python has a built-in function open() to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

```
>>> f = open("test.txt") # open file in current directory
>>> f = open("C:/Python33/README.txt") # specifying full path
```

We can specify the mode while opening a file. In mode, we specify whether we want to read 'r', write 'w' or append 'a' to the file.
We also specify if we want to open the file in text mode or binary mode.
The default is reading in text mode. In this mode, we get strings when reading from the file. On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like image or exe files.

Python File Modes

Mode : Description
'r' : Open a file for reading. (default)
'w' : Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
'x' : Open a file for exclusive creation. If the file already exists, the operation fails.
'a' : Open for appending at the end of the file without truncating it. Creates a new file if it does not exist.
't' : Open in text mode. (default)
'b' : Open in binary mode.
'+' : Open a file for updating (reading and w

f = open("test.txt") # equivalent to 'r' or 'rt'
f = open("test.txt",'w') # write in text mode
f = open("img.bmp",'r+b') # read and write in binary mode

Hence, when working with files in text mode, it is highly recommended to specify the encoding type.
f = open("test.txt",mode = 'r',encoding = 'utf-8')

Closing a File

When we are done with operations to the file, we need to properly close it.
Closing a file will free up the resources that were tied with the file and is done using the close() method.

Python has a garbage collector to clean up unreferenced objects but, we must not rely on it to close the file.

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

This method is not entirely safe. If an exception occurs when we are performing some operation with the file, the code exits without closing the file. A safer way is to use a try...finally block.

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

This way, we are guaranteed that the file is properly closed even if an exception is raised, causing program flow to stop.
The best way to do this is using the with statement. This ensures that the file is closed when the block inside with is exited.
We don't need to explicitly call the close() method. It is done internally.

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

Reading and writing

A text file is a sequence of characters stored on a permanent medium like a hard drive, flash memory, or CD-ROM.
To write a file, you have to open it with mode 'w' as a second parameter:

```
>>>	fout = open('output.txt', 'w')
>>>	print fout
<open file 'output.txt', mode 'w' at 0xb7eb2410>
```

If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful! If the file doesn’t exist, a new one is created.
The write method puts data into the file.

```
>>>	line1 = "This here's the wattle,\n"
>>>fout.write(line1)
```

Again, the file object keeps track of where it is, so if you call write again, it adds the new data to the end.

```
>>>	line2 = "the emblem of our land.\n"
>>>	fout.write(line2)
```

When you are done writing, you have to close the file.

```
>>> fout.close()
```

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 an integer (d stands for “decimal”):

```
>>> 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:

```
>>>	camels = 42
>>>	'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: illegal argument type for built-in operation
```

Filenames and paths

Files are organized into **directories** (also called "folders"). Every running program has a "current directory," which is the default directory for most operations. For example, when you open a file for reading, Python looks for it in the current directory.

The os module provides functions for working with files and directories ("os" stands for "operating system"). os.getcwd returns the name of the current directory:

```
>>>     import os
>>>     cwd = os.getcwd()
>>>     print cwd /home/dinsdale
```

cwd stands for "current working directory." The result in this example is /home/dinsdale, which is the home directory of a user named dinsdale.

A string like cwd that identifies a file is called a **path**. A **relative path** starts from the current directory; an **absolute path** starts from the topmost directory in the file system.

The paths we have seen so far are simple filenames, so they are relative to the current directory. To find the absolute path to a file, you can use os.path.abspath:

```
>>>     os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

os.path.exists checks whether a file or directory exists:

```
>>> os.path.exists('memo.txt')
True
```

If it exists, os.path.isdir checks whether it’s a directory:

```
>>>     os.path.isdir('memo.txt')
False
>>>     os.path.isdir('music')
True
```

Similarly, os.path.isfile checks whether it’s a file.
os.listdir returns a list of the files (and other directories) in the given directory:

```
>>>     os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

To demonstrate these functions, the following example “walks” through a directory, prints the names of all the files, and calls itself recursively on all the directories.

```
def walk(dirname):
for name in os.listdir(dirname):
path = os.path.join(dirname, name)
if os.path.isfile(path):
print path
else:
walk(path)
```

os.path.join takes a directory and a file name and joins them into a complete path.