Unit 5
Files

It is the last major built-in object type in Python.

Persistent - remain forever
Used when data need to be 'persistent'.

Two approaches to maintain data in programs:
1. Read and write in a text file
2. Store data as a database

The built-in 'open()' function creates a Python file object, which serves as a link to a file residing on your machine. After calling 'open()', strings of data can be transferred to and from the associated external file by calling the returned file object's methods.

Most of the file methods are concerned with performing 'input from' and 'output to' the external file associated with a file object, but other file methods allow us to seek to a new position in the file, flush output buffers and so on..

Common file operations:

(1) Opening files

Syntax:
fhandle = open('filename', 'mode')

meaning:
'filename' - may include a platform-specific and absolute or relative directory path prefix. Without a directory path, the file is assumed to exist in the current working directory

mode (first letter)-
'r' means read.
'w' means write. If the file doesn't exists, it is created. If the file exists, it is overwritten.
'x' means write, but only if the file does not exist.
'a' opens for appending text at the end

mode (second letter)-
't' or nothing means text.
'b' means binary.
'+' means read and write a text file. This can be used only with 'a' mode.. ('a+')

Note: After performing all the operations on file, it has to be closed using the close().

(2) Closing files

Syntax:
(a) fhandle.close()

meaning:
Manual close of file object

fhandle.close()

(b) fhandle.flush()

meaning:
Flush output buffer to disk without closing
[Useful when we perform both write and read operations on the same file and we need to read the contents written by the last write()]

(c) Close files automatically by using 'with' ...(context manager)

>>>with open(filename,mode) as fhandle:
. . . #perform file operations here
. . .
>>>
After the block of code is complete, the file 'filename' is closed automatically by the context manager.

Note: Use of 'with' clause (context manager) for file operations is the most preferred and safest manner of handling files.

(3) Writing to files

File to be written should be opened in 'w' or 'a' mode.

Syntax:
(a) fhandle.write(aString)

meaning:
Writes a string of characters (or bytes) into file

(b) fhandle.writelines(aList)

meaning:
Writes all line strings in a list into file

(c) Write a Binary file with write()

mode - 'wb' or 'ab'

Instead of strings, bytes are written and read from binary files.

eg:
Let's generate 256 'byte values' from 0 to 255:

```
>>>bdata = bytes(range(0,256))
>>>len(bdata)
256

>>>f1=open('bfile','wb')
>>>f1.write(bdata)
256
>>>f1.close()
```

(4) Reading from files

File to be written should be opened in 'r' or nothing.

Syntax:
(a) aString = fhandle.read()

meaning:
Reads an entire file into a single string

(b) aString = fhandle.read(N)

meaning:
Reads up to next N characters (or bytes) into a string

(c) aString = fhandle.readline()

meaning:
Reads next line (including '\n' newline) into a string

#same as using: for line in fhandle: ... that reads the next line in file to var 'line'

(d) aList = fhandle.readlines()

meaning:
Reads entire file into a list of line strings (with '\n')

(e) Read a Binary file with read()

mode - 'rb'

eg:
```
>>>f1=open('bfile','rb')
>>>bdata1=f1.read()
>>>len(bdata1)
256
>>>f1.close()
```

(5) File position

It's suitable while working with binary files.

Syntax:
(a) fhandle.seek(offset,origin)

meaning:
If origin is 0, (default), go offset bytes from the start
If origin is 1, go offset bytes from current position
If origin is 2, go offset bytes relative to the end

(b) fhandle.tell()

meaning:
Returns the current offset from the beginning of the file, in bytes.

(6) Iterations on file object - walking thru file contents..

(a) Using for loop..

Syntax:
for line in open('filename'):
    #use line

meaning:
File iterators reads line by line

(b) In a print statement

Syntax:
>>>print(open('filename').read())
#prints contents of 'filename' line by line iteratively

(c) Using __next__()

The __next__() in 3.x ('next' in 2.x - almost similar) returns the next line from a file each time it is called.

The only noticeable difference is that __next__ raises a built-in 'StopIteration' exception at end-of-file instead of returning an empty string

Syntax:

```
>>>f = open('filename')
>>>f.__next__()       # use f.next() or next(f) in 2.x
# reads and prints the first line in 'filename'
>>>f.__next__()
# reads and prints the next line in 'filename'

if end of file is reached..
>>>f.__next__()
TraceBack (most recent call last):
   File "<stdin>", line 1, in <module>
StopIteration
>>>
```

(c) Using 'iter()'

Syntax:

```
>>>f = open('filename')
>>>iter(f) is f
True
>>>my_iter = f.__iter__()
>>>my_iter.__next__()
# reads and prints the first line in 'filename'
>>>my_iter.__next__()
# reads and prints the next line in 'filename'
...
```

Some examples:

eg(1):

```
>>>myfile = open('myfile.txt','w')
>>>myfile.write('Hello text file\n')
16
>>>myfile.write('goodbye text file\n')
18
>>>myfile.close()
```

eg(2):
```
>>>myfile = open('myfile.txt')
>>>myfile.readline()
'Hello text file\n'
>>>myfile.readline()
'goodbye text file\n'
>>>myfile.readline()
''
```

eg(3): Read all at once!!
```
>>>open('myfile.txt').read()
'Hello text file\ngoodbye text file\n'
>>>
```

eg(4): User-friendly!!
```
>>>print(open('myfile.txt').read())
Hello text file
goodbye text file
>>>
```

eg(5): Using iterators # observe no 'reads' here!!
```
>>>for line in open('myfile.txt'):
. . .       print(line, end='')
. . .
Hello text file
goodbye text file
>>>
```

Storing Python Objects in Files: Conversions

Any Python object can be written into a text file on multiple lines. It must only be converted to strings before writing!

Note: write() does not do any automatic to-string formatting for us!!

eg:

```
>>>X, Y, Z = 43, 55, 12   # set of integer variable

>>>S = 'Hello'   # a string

>>>D = { 'a': 1, 'b': 2 }       # a dictionary

>>>L = [ 1, 2, 3 ]     # a list


>>>F =open('datafile.txt','wt')

>>>F.write(S + '\n')

>>>F.write('%s, %s, %s\n' %(X, Y, Z))

>>>F.write(str(L) + '\n' + str(D) + '\n')

>>>F.close()
>>>


>>>F = open('datafile.txt','rt')
# reads the contents..
>>>F.readline()
'Hello\n'
>>>F.readline()
'43, 55, 12\n'
>>>
```

Use 'eval' to convert from strings to objects:

eg:

```
>>>line = F.readline()
>>>print(line)
'43, 55, 12'
>>>eval(l)
( 43, 55, 12 )
>>>
```

Storing Native Python Objects: Pickle

Saving data structures to a file is called **'Serializing'**.

Using ‘eval’ to convert strings to objects is a powerful tool. But, Python provides the 'pickle' module to save and restore any object in a special binary format.

The 'pickle' module is an advanced tool that allows us to store almost any Python object in a file directly, without any string conversion!!

It's like a 'super-general' data formatting and parsing utility.

Two functionalities: Pickling and Unpickling

1. Pickling - Converting objects to 'strings of bytes'

(a) dumps()
Syntax:
import pickle
pkld = pickle.dumps(obj)
#returns the pickled representation of obj as a 'bytes' object

(b) dump()
Syntax:
import pickle
pickle.dump(obj, file)
write a pickled representation of obj to file object

2. Unpickling - Converting strings of bytes to objects

(a) loads()
Syntax:
import pickle
obj = pickle.loads(data)
read and return object from the pickle data

(b) load()

Syntax:
import pickle
obj = pickle.load(file)
#read and return an object from pickle data stored in a
file object

eg(1):
Using dump() and load() to/from a file object

```
>>>D = { 'a' : 1, 'b' : 2 }
>>>F = open('datafile.pkl', 'wb')  #do not forget to make it a binary file
>>>import pickle
>>>pickle.dump(D, F)
>>>F.close()
>>>
```

```
>>>F = open('datafile.pkl', 'rb') # read as a binary file
>>>E = pickle.load(F)
>>>E
{ 'a' : 1, 'b' : 2 }
```

eg(2):
Using dumps() and loads() without file object

```
>>>import pickle
>>>import datetime

>>>now1 = datetime.datetime.today()
>>>now1
datetime.datetime(2017, 10, 17, 9, 7, 8, 296378)

>>>pickled = pickle.dumps(now1)
>>>now2 = pickle.loads(pickled)
>>>now2
datetime.datetime(2017, 10, 17, 9, 7, 8, 296378)
>>>
```

Databases

'anydbm' - provides Generic access to DBM-style databases
In Python 2.x : 'anydbm'
In Python 3.x : 'dbm'

The module 'anydbm' provides an interface for creating and updating database files.

eg:
Let's create a database file.

Syntax:

```
>>>import dbm       #anydbm in Python 2.x
>>>db = dbm.open('database file','flag')
```

where,
'database file' is the name of the database file to be created or accessed

'flag' indicates the mode of operation..
- 'r' - default mode, for read-only access of an existing database
- 'w' - read-write access for an existing database
- 'c' - read-write access for a new or existing database
- 'n' - read-write access for a new database

Note: 'r' and 'w' fail if database doesn't exist
- 'c' creates it only if it doesn't exist
- 'n' always creates a new database

```
>>>import dbm
>>>db1 = dbm.open('newdb.db', 'c' )
```

'db1' is a database object that can be used like a dictionary!!

eg:
```
>>>db1['usn'] = 'University Number'
>>>db1['name'] = 'Student Name'
>>>db1['course'] = 'Course Name'
>>>db1.close()
```

Accessing the database.. is similar to accessing a dictionary

eg:

```
>>>db1=dbm.open('newdb.db')
>>>for key in db1:
. . .       print(key , db1[key])
. . .
#prints the database contents as key - value pairs
>>>db1.close()
```