LEARNING TO CODE

python™

Coordinator: Konstantinos Emmanouilidis

IEEE SB OF THRACE

2017-2018

1. Files
2. OS Methods
3. Directories
4. Modules & Packages

FILE & DIRECTORY METHODS

- File Object Methods: Provide functions to manipulate files.
- OS Object Methods: Provide methods to process files as well as directories.

OPENING A FILE…

- **open()**: Creates a file object, which calls other support methods associated with it.
  - Syntax: file_object = open(file_name [,access_mode][,buffering])
  - **file_name**: A string that contains the name of the file that you want to access.
  - **access_mode**: Determines the mode in which the file has to be opened, i.e., read, write, append, etc. This is optional parameter and the default file access mode is read (r).
  - **buffering**: If it is set to 0, no buffering takes place. If the value is 1, line buffering is performed while accessing a file. If the value is greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default(default behavior).

Modes & Description
r Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
rb Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
r+ Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
rb+ Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
w Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
wb Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+ Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
wb+ Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.

a

Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.

ab

Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.

a+

Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

ab+

Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

CLOSING A FILE….

- close() method: flushes any unwritten information and closes the file object
- Python automatically closes a file when the reference object of a file is reassigned to another file.
- It is a good practice to use the close() method to close a file.
- **Syntax**: file_object.close()

EXAMPLE

```
# Open a file
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name

# Close opend file
fo.close()
```

```
Name of the file:  foo.txt
```

READING AND WRITING FILES

- ***write()*** method: writes any string to an open file.
  - Python strings can have binary data and not just text.
  - Does not add a newline character ('\n') to the end of the string
  - Syntax**:** file_object.write(string)

- ***read()*** method: reads a string from an open file.
  - Syntax**:** file_object.read(*count*)
  - ***count:*** is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if *count* is missing, then it tries to read as much as possible, maybe until the end of file.

EXAMPLES

```
# Open a file
fo = open("foo.txt", "wb")
fo.write( "Python is a great language.\nYeah its great!!\n")

# Close opend file
fo.close()
```

```
Python is a great language.
Yeah its great!!
```

```
# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str
# Close opend file
fo.close()
```

```
Read String is :  Python is
```

Attribute & Description
file.closed Returns true if file is closed, false otherwise.
file.mode Returns access mode with which file was opened.
file.name Returns name of the file.

FILE POSITIONS

- ***tell()*** method: tells you the current position within the file.In other words, the next read or write will occur at that many bytes from the beginning of the file.
- ***seek(offset[, from])*** method: changes the current file position.
- *offset :* indicates the number of bytes to be moved.
- *from :* specifies the reference position from where the bytes are to be moved.
- *from* is set to 0 for beginning,to 1 for current position,to 2 for the end of the file.

```
# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str

# Check current position
position = fo.tell();
print "Current file position : ", position

# Reposition pointer at the beginning once again
position = fo.seek(0, 0);
str = fo.read(10);
print "Again read String is : ", str
# Close opend file
fo.close()
```

```
Read String is :  Python is
Current file position :  10
Again read String is :  Python is
```

RENAMING & DELETING FILES

- import **os** module: provides methods that help you perform file-processing operations
- rename() Syntax: os.rename(current_file_name,new_file_name)
- remove() Syntax: os.remove(file_name)

EXAMPLES

```
import os

# Rename a file from test1.txt to test2.txt
os.rename( "test1.txt", "test2.txt" )
```

```
import os

# Delete file test2.txt
os.remove("text2.txt")
```

DIRECTORIES

- The **os** module has several methods that help you create, remove and change directories.
- *mkdir()* method: creates directories in the current directory.
  - Syntax: os.mkdir(“new_dir”)
- *chdir()* method: changes the current directory.
  - Syntax: os.chdir(“new_dir”)
- *getcwd()* method: displays the current working directory.
  - Syntax: os.*getcwd()*
- *rmdir()* method deletes the directory
  - Syntax: os.rmdir(‘dir_name’)

MODULES

- A module is a Python object with arbitrarily named attributes that you can bind and reference.
- A module can define functions, classes and variables.
- A module can also include runnable code.
- A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use.

IMPORT STATEMENT

- Syntax: import module1, module2, …
- When the interpreter encounters an import statement, it imports the module if the module is present in the search path.
- A search path is a list of directories that the interpreter searches before importing a module.
- A module is loaded only once, regardless of the number of times it is imported.

FROM…IMPORT STATEMENT

- *from … import* statement: it lets you import specific attributes from a module into the current namespace
- Syntax: from module_name import name1,name2,…
- from … import * : import all names from a module into the current namespace

LOCATING MODULES

When you import a module, the Python interpreter searches for the module in the following sequences

- The current directory.
- If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
- If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.

The module search path is stored in the system module sys as the **sys.path** variable. The sys.path variable contains the current directory, PYTHONPATH, and the installation-dependent default.