Python File IO (Input-Output)

Python makes it very easy to read and write files to disk.

Keep in mind that it is almost always better to use a Python module for specific formats. For example, use the `json` module for JSON files or the `csv` module for `.csv` files. Better yet, use Pandas for table-like data.

What is a file?

A *file* is a segment of data, typically associated with a filename, that exists in a computer's persistent storage. This means that the data remains when the computer is turned off.

There are two main kinds of files: *text* and *binary*.

Text files are typically easier for humans to read and write.

Binary files (images, music files, etc.) are more efficient in terms of storage.

Python scripts are text files and by convention have a `.py` extension. On unix systems we can dump a text file to the terminal with:

```
$ cat hello.py
# run me from the command line with
# $ python3 hello.py

print("hello sweet world!")
```

For fun, try dumping a binary file to the terminal with `$ cat /bin/ls`. What happens?

In Python it is very easy to open, read from, and write to a text file. Let's see how it works.

See Chapter 9 in **Learning Python** for information on accessing files with Python. The relevant information starts on page 282.

The file object

- Interaction with the file system is pretty straightforward in Python.
- Done using *file objects*
- We can instantiate a file object using `open` or `file`

Opening a file

```
f = open(filename, option)
```

- `filename`: path to file on disk
- `option`: mode to open file (passed as a string)
- `'r'`: read file
- `'w'`: write to file (overwrites existing file)
- `'a'`: append to file
- We need to close a file after we are done: `f.close()`

Open a file:

```
f = open("humpty-dumpty.txt","r")
f
```

We can test if the file is closed:

```
f.closed
```

We can close the file:

```
f.close()
f.closed
```

Closing a file flushes any buffered data to disk and frees up operating system resources. If using a file in this manner, it is important to close files. *We will take off points if you neglect to do this.*

`with open() as f:`

It is good practice to use the `with` keyword when dealing with file objects. This has the advantage that the file is properly closed after its suite finishes, even if an exception is raised on the way.

```
with open('humpty-dumpty.txt', 'r') as f:
    print(f.read())

f.closed
```

If a file does not exist

```
bad_file = open("no-file.txt","r")
```

Reading data from a file

File object methods:

- `read()`: Read entire file (or first `n` characters, if supplied)
- `readline()`: Reads a single line per call
- `readlines()`: Returns a list with lines (splits at newline)

`readline()`

Use the `readline()` method to read lines from a file:

```
f = open("humpty-dumpty.txt","r")
print(f.readline())
print(f.readline())
f.close()
```

`read()`

You can read an entire file at once with the `read()` method:

```
f = open("humpty-dumpty.txt","r")
poem = f.read()
print(poem)
f.close()
```

Iterate over lines

You can very easily iterate over lines in a file with:

```
f = open("humpty-dumpty.txt","r")
for line in f:
    print(line)
f.close()
```

An example with `with`

```
with open('humpty-dumpty.txt', 'r') as f:
    for i, line in enumerate(f):
        print("line {}: {}".format(i,line))
```

Note the extra lines between each line of text. You can do this by specifying the `end` keyword parameter for the `print` function to be an empty string (`""`): `print(line, end='')` or slicing `line` with `print(line[:-1])`.

Iterate over words!

The string `split` method partitions a string into a list based on a delimiter. Space is the default delimiter. The `strip` method removes leading and trailing whitespace from a string.

```
f = open("humpty-dumpty.txt","r")
for line in f:
    for word in line.split():
        # use strip() method to remove extra newline characters
        print(word.strip())
f.close()
```

Writing to file

Use the `write()` method to write to a file. Make sure to open the file in write mode with `'w'` as the second argument to `open()`.

```
name = "Python learner"
with open('hello.txt', 'w') as f:
    f.write("Hello, {}!\n".format(name))
```

```
%cat hello.txt
```

More writing examples

Write elements of list to file:

```
xs = ["i", "am", 'a', 'fancy', 'list', 42]
with open('from_list.txt', 'w') as f:
    for x in xs:
        f.write('{}\n'.format(x))
```

```
%cat from_list.txt
```

To write multiple lines to a file at once, use the `writelines` method:

```
f = open("writelines.txt","w")
f.writelines(["a mighty fine day\n","ends with a great big party\n","thank you, its friday\n"])
f.close()
```

```
%cat writelines.txt
```

Note that the `write` and `writeliens` methods will not insert newline characters. To get a new line, you must add `'\n'` to the strings.

Buffering

For efficiency, the `file` object will temporarily store data from `write` or `writelines` methods in memory before actually writing to disk. This is known as buffering. It turns out that writing larger chunks of data to disk in fewer transactions is more efficient than many transactions of small chunks. If you attempt to open a text file created by Python and not closed, you may not see the data. Calling the `close()` method flushes all data to disk.

```
f = open('foo.txt','w')
f.write("this is some sweet text\n")
%cat foo.txt
```

(On my system `foo.txt` is empty at this point. Behavior may be different on your system.)

```
f.close()
%cat foo.txt
```