File Input/Output

Text File IO

- File IO is done in Python with the built-in `File` object which is returned by the built-in `open` function
- Use the 'w' open mode for writing

```
$ python
>>> f = open("hello.txt","w") # open for writing, create if necessary
>>> f.write("Hello, file!\n") # write string to file; notice \n ending
>>> f.close()                 # close file, causing it to write to disk
>>> exit()
$ cat hello.txt
Hello, file!
```

Use the 'r' open mode for reading

```
$ python
>>> f = open("hello.txt", "r") # open for reading in text mode
>>> contents = f.read()
# slurp the whole file into memory
>>> contents
'Hello, file!\n'
>>> exit()
```

Active Review

- Create a text file named `lines.txt` with three lines, `line 1`, `line 2`, and `line 3`.

Reading Lines from Text Files

The `readlines()` method reads all lines into memory as a list

```
>>> f = open("lines.txt", "r")
>>> f.readlines()
["line 1\n", "line 2\n", "line 3\n"]
```

- `readline()` reads one line at a time, returning empty string when fully read
- re-open file or use seek() to go back to beginning of file

```
>>> f = open("lines.txt", "r")
>>> f.readline()
'line 1\n'
>>> f.readline()
'line 2\n'
>>> f.readline()
'line 3\n'
>>> f.readline()
''
>>> f.seek(0)
>>> f.readline()
'line 1\n'
```

Active Review

- Use the walrus operator, `:=`, and a `while` loop to read each line of `lines.txt` and print it to stdout.

Processing Lines in a Text File

Could use readlines() and iterate through list it returns

```
>>> f = open("lines.txt", "r")
>>> for line in f.readlines():
...     print line
...
line 1
line 2
line 3
```

Better to take advantage of fact that a file object is `Iterable`

```
>>> for line in open("lines.txt", "r"):
...     print line
...
line 1
line 2
line 3
```

Files are Buffered

Try a little experiment. create a subdirectory named foo, cd to your new empty `foo` directory, launch a Python shell, create open a new file named bar, and write something to it:

```
$ mkdir foo
$ cd foo
$ python3
Python 3.4.0 (v3.4.0:04f714765c13, Mar 15 2014, 23:02:41) ...
>>> bar = open("bar", "w")
>>> bar.write("last call!")
10
>>>
```

Now open another command shell or use your graphical file explorer to view the contents of the bar file. It's empty. Now go back to your Python shell and do:

```
>>> bar.close()
```

Now view the contents of the `bar` file again. It has the text from the previous `write()` call. Files are buffered, and the buffer isn't (guaranteed to be) flushed to disk until the file object is closed or the `File` Object goes out of scope or the program terminates (gracefully).

Context Management with `with`

Python has context managers to close resources automatically. A context manager has the form

```
with expression as variable:
    block
```

which is equivalent to

```
variable = expression
block
variable.close()
```

For example, the previous bar example is:

```
>>> with open("bar", "w") as bar:
...     bar.write("last call!")
...
```

And the file is closed and flushed to disk automatically after the block under the with statement finishes.

Common File and Directory Tasks

Listing Files in a directory

```
1 import os
2 dir = 'some_dir'
3 for path in os.listdir(dir):
4     if os.path.isdir(path):
5         print(path + '/')
6     else:
7         print(path)
```

Moving and Copying Files

```
1 import shutil
2 shutil.move(source, destination)
3 shutil.copy(source, destination)
```

Making directories

```
1 import shutil
2 dir = 'some_dir'
3 shutil.mkdir(dir)
```

Conclusion

- Easy to write file processing utilities with Python.
- Many other libraries, like `pandas`, handle the file processing under the hood.
- Take a look at other built-in file-related standard modules, like gzip and tarfile.