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

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

Use the 'r' open mode for reading

```
1  $ python
2  >>> f = open("hello.txt", "r") # open for reading in text mode
3  >>> contents = f.read()
4  # slurp the whole file into memory
5  >>> contents
6  'Hello, file!\n'
7  >>> 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

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

```
1   >>> f = open("lines.txt", "r")
2   >>> f.readline()
3   'line 1\n'
4   >>> f.readline()
5   'line 2\n'
6   >>> f.readline()
7   'line 3\n'
8   >>> f.readline()
9   ''
10  >>> f.seek(0)
11  >>> f.readline()
12  '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

```
1  >>> f = open("lines.txt", "r")
2  >>> for line in f.readlines():
3  ...     print line
4  ...
5  line 1
6  line 2
7  line 3
```

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

```
1  >>> for line in open("lines.txt", "r"):
2  ...     print line
3  ...
4  line 1
5  line 2
6  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:

```
1  $ mkdir foo
2  $ cd foo
3  $ python3
4  Python 3.4.0 (v3.4.0:04f714765c13, Mar 15 2014, 23:02:41) ...
5  >>> bar = open("bar", "w")
6  >>> bar.write("last call!")
7  10
8  >>>
```

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:

```
1  >>> 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

```
1  with expression as variable:
2      block
```

which is equivalent to

```
1  variable = expression
2  block
3  variable.close()
```

For example, the previous bar example is:

```
1  >>> with open("bar", "w") as bar:
2  ...       bar.write("last call!")
3  ...
```

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

```python
import os
dir = 'some_dir'
for path in os.listdir(dir):
    if os.path.isdir(path):
        print(path + '/')
    else:
        print(path)
```

Moving and Copying Files

```python
import shutil
shutil.move(source, destination)
shutil.copy(source, destination)
```

Making directories

```python
import shutil
dir = 'some_dir'
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.