Intro to Computer Science

Previous

- Dictionaries

Next

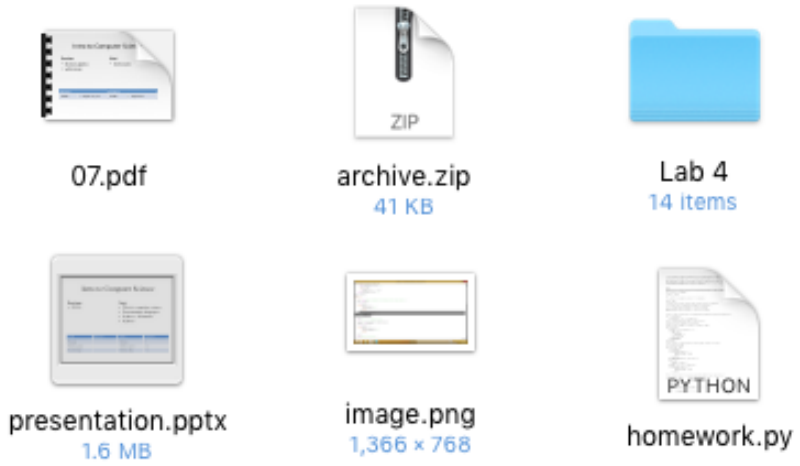
- File I/O

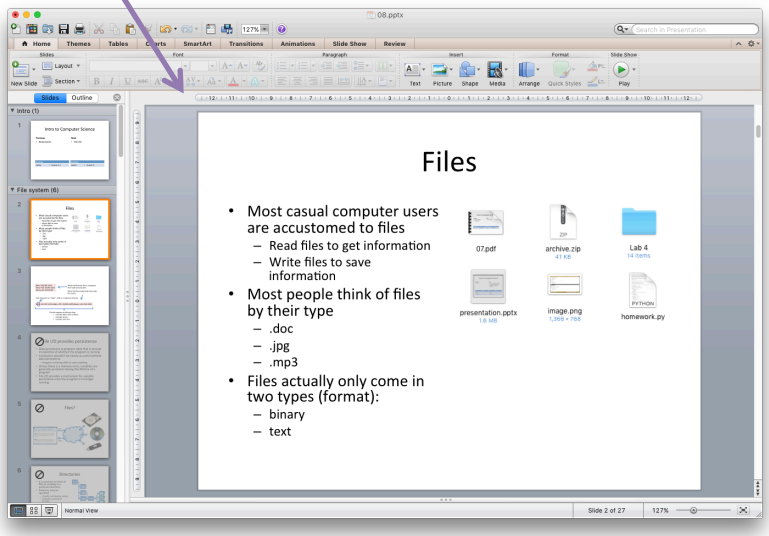
Readings	
Gaddis	• Chapter 9.1

Readings	
Gaddis	• Chapter 6

Files

- Most casual computer users are accustomed to files
  - Read files to get information
  - Write files to save information
- Most people think of files by their type
  - .doc
  - .jpg
  - .mp3
- Files actually only come in two types (format):
  - binary
  - text





Text files: the basics

```
Bob 1-212-555-1212
Mary +971-56-856-2245
Nancy +44-1523-4456
```

What the file looks like in a program that reads and presents

What the file actually looks like under the covers

Each character is a "byte"; a file is a *sequence* of bytes

```
Bob 1-212-555-1212\nMary +971-56-856-2245\nNancy +44-1523-4456
```

This file happens to 60 bytes long

- Includes letters and numbers
- Includes spaces
- Includes new lines

Files are meant to be opened

```
fp = open('words.txt')
```

To open a file call `open`!

- `open` requires the file name
- Either the *relative path* or the *absolute path*
- Python will throw an exception (error) if the file cannot be opened

- The open method returns a file object
- We've seen objects before:
  - strings
  - lists
  - dictionaries
- An object has a value associated with it, and a collection of methods available to it

Reading

- There are several methods that read data from files
  - Per byte
  - Per line
- Depends on your programming requirements
  - Per line covers most use cases
  - Python makes this easy!

Methods for reading files

Method	Notes
`read()`	• Reads an entire file (in one call) • Use with caution: if the file is bigger than your memory space you'll have problems
`read(n)`	• Read n bytes (or less) from a file
`readline()`	• Read a single line from a given file • Subsequent calls read subsequent lines (maintains internal pointer) • *Of all reading methods, this is likely to be the most useful*
`readlines()`	• Returns the entire file as a list of strings • Each element in the list corresponds to a line in the file • List order corresponds to line order

Example:

```
fp = open('file.txt')
single_line = fp.readline()
entire_file = fp.read()
```

Methods return empty string to signify end-of-file (EOF)

You have to strip

- The readline methods return a string that includes the newline terminator
- You almost always want to get rid of this!

file.txt

This is the first line
This is the second line

```
>>> fp = open('file.txt')
>>> line = fp.readline()
>>> print(line[-1])
```

The last character is probably not what you think!

```
>>> line
'This is the first line\n'
>>> print(line.strip())
This is the first line
>>>
```

The newline character is part of the string!

File objects are sequences!

- Recall a sequence: data that's treated like an array
  - strings
  - lists
- Recall why sequences are awesome:
  - *Iteration using a for-loop*
- In a vast majority of cases, this is how you will process a file

Let Python do the work

file.txt

This is the first line
This is the second line
This is the third line

```
fp = open('file.txt')
for line in fp:
    sLine = line.strip()
    print(sLine)
```

Why are we stripping?

iteration 1 → This is the first line

iteration 2 → This is the second line

iteration 3 → This is the third line

Writing

- Two common ways of writing files: write, print
- Assume a writable open file,

```
fp = open('file.txt', 'w')
```

methods are as follows:

Method	Usage	Notes
`write`	`fp.write('Hello file!')`	Writes the string to the file using the object method
`print`	`print('Hello file!', file=fp)`	Writes a string to a file using the print function • Advantage: casting (to string) is implicit • Disadvantage: must remember to use the "file" parameter • ?vantage: newline is written implicitly

Close your file!

- After you are finished with a file object you ~~should~~ **must** close it
- Not closing a file object will not result in an immediate error
- But there are various problems you will experience if you get into the habit of not closing
  - Reaching the OS limit on the number of files open
  - I/O operations not actually being committed
- Going forward: *we will always deduct points for not closing*

```
fp = open('file.txt')
# your reading/writing code
fp.close()
```

That's it! (So there's no reason not to do it)

The file may not be saved!

(Depeche) Mode

- Files can be opened with various modes
  - Reading, writing, appending
  - Several options: see help(open)
- Specified using an optional parameter to open

Assume file is a string containing a valid filename:

```
fp = open(file)

fp = open(file, mode='r')
```

- Open the file for reading
- These two lines are equivalent

```
fp = open(file, mode='w')
```

- Open the file for writing
- Truncate the file if it already exists

```
fp = open(file, mode='a')
```

- Open the file for writing (appending)
- Existing data is preserved

Persistence: realized

write_state.py

```
name = 'John'
age = 10
shoes = 12.5
filename = 'john.txt'
fp = open(filename, 'w')
print(name, file=fp)
print(age, file=fp)
print(shoes, file=fp)
```

reread_state.py

- Stop execution in one place
- Resume execution in another

```
filename = 'john.txt'
fp = open(file, 'r')
name = fp.readline()
age = int(fp.readline())
shoes = float(fp.readline())
```