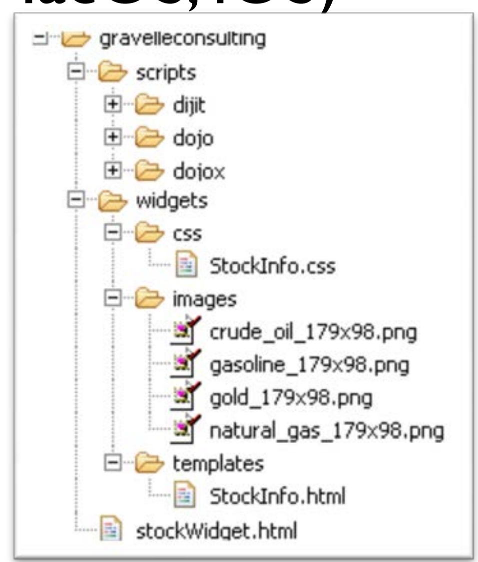
Files

- Variables are stored in RAM (Random Access Memory)
  - RAM is fast and easy to use and reuse
  - RAM is *volatile*: cleared when the program exits
- *Files* are stored in non-volatile memory
  - The computer's magnetic *hard drive*
  - A USB stick (a.k.a *flash memory*)
  - Optical media (CDs, DVDs)
  - Non-volatile memory is slow and cheap compared to RAM
  - Files include programs, your homework, web pages, etc.

Directory Structure

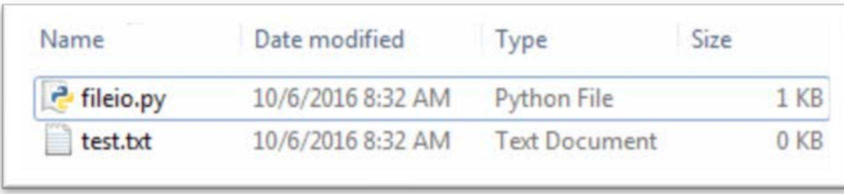
- Modern *operating systems* (e.g. Windows, MacOS, iOS) store files in a hierarchical tree
- A *path* describes the location of a file
  - E.g. \gravelleconsulting\widgets\css\
  - Different OSs use / or \ in path names
  - Windows paths start with the drive letter
    - E.g. C:\home\classes\1370\
  - *Absolute paths* start at the *root* of the file system
  - *Relative paths* are relative to some location in the tree
    - E.g. images/crude_oil_179x98.png
- When running a Python script, paths are relative to the location of that script



Writing to a File

- Conceptually, just like calling *print*, only we're sending characters to a file instead of to the Python console
- First, open the file
  - Must specify the file name to open, using a path
  - If the file is in the same directory as your program, then just the filename will suffice

```
f = open("test.txt", 'w')
```



Name	Date modified	Type	Size
fileio.py	10/6/2016 8:32 AM	Python File	1 KB
test.txt	10/6/2016 8:32 AM	Text Document	0 KB

  - The 'w' argument tells it we want to open the file for writing

Writing to a File

- Second, write strings to the file
  - *write* is a method you call on a file object (like f above)
  - Unlike *print*, *write* only takes strings and does not automatically add a new line

```
f = open(“test.txt”, ‘w’)
f.write(“This is a string in my file”)
f.write(“This is another string on the same line”)
f.write(“\nThat there is a new line character”)
```

- Third, close the file
  - If you don’t close, what you write may be lost!
  - Remember that files are slow? The system *buffers* writes (keeps them in memory) and does them all together in batches

```
f.close()
```

Writing to a File

- Files support *sequential access*
  - Start at the beginning, one character at a time to the end
  - Contrast to RAM which can put and get variables anywhere
- Files are opened in different modes
  - 'w' : open for writing, replace any existing file by that name
  - 'a' : open for appending, add to the end of any existing file
  - 'r' : open for reading
  - There are more, but we'll start there

Reading From a File

- Conceptually, just like calling *input*, only we're getting characters from a file instead of from the console
- Same pattern as writing
  - Open the file
  - Read from the file
  - Close the file
- Reading from a file is also sequential, you get each character in the file in order
- Reading is more complex, just like user input, because you can't guarantee what you'll get

Reading From a File

- *read* returns the whole file as a string

```
>>> f = open("test.txt", 'r')
>>> f.read()
'77\n88\n99 100 101\n102\n'
```

- *readline* returns the next line
  - All characters up to and including the next newline '\n'

```
>>> f = open("test.txt", 'r')
>>> f.readline()
'77\n'
>>> f.readline()
'88\n'
>>> f.readline()
'99 100 101\n'
```

- *readlines* returns all lines from the file in a list

```
>>> f = open("test.txt", 'r')
>>> f.readlines()
['77\n', '88\n', '99 100 101\n', '102\n']
```

Reading From a File

- Very common to loop over the lines in a file

```
f = open("test.txt", 'r')
for line in f.readlines():
  # do something
f.close()
```

- So common, there's a syntactic shortcut
  - Loop over the file object itself

```
f = open("test.txt", 'r')
for line in f:
  # do something
f.close()
```

Reading From a File

- Often have to convert file strings to data (e.g. numbers)
- Just like converting user input
  - Note that int() and float() are smart enough to drop whitespace
  - `int("        77      \n")` returns the number 77
- But no magic for multiple numbers in a string
  - `int("77 88 99")` throws an error
- The *split* method is quite useful here
  - Called on a string, pass in a delimiting character
  - Splits it on every instance of the delimiter into a list of strings
  - `"Me Myself I".split(' ')` returns the list `['Me', 'Myself', 'I']`
  - `"A, B, C".split(',')` returns the list `['A', ' B', ' C']`
    - Note the spaces are still there

Reading From a File

- The *strip* method can also be quite useful
  - Called on a string, removes all preceding and trailing whitespace
  - `“   Get Out    ”.strip()` returns the string `“Get Out”`