OS and Software Interaction

Eric Franzosa (franzosa@hsph.harvard.edu)

http://franzosa.net/bst273

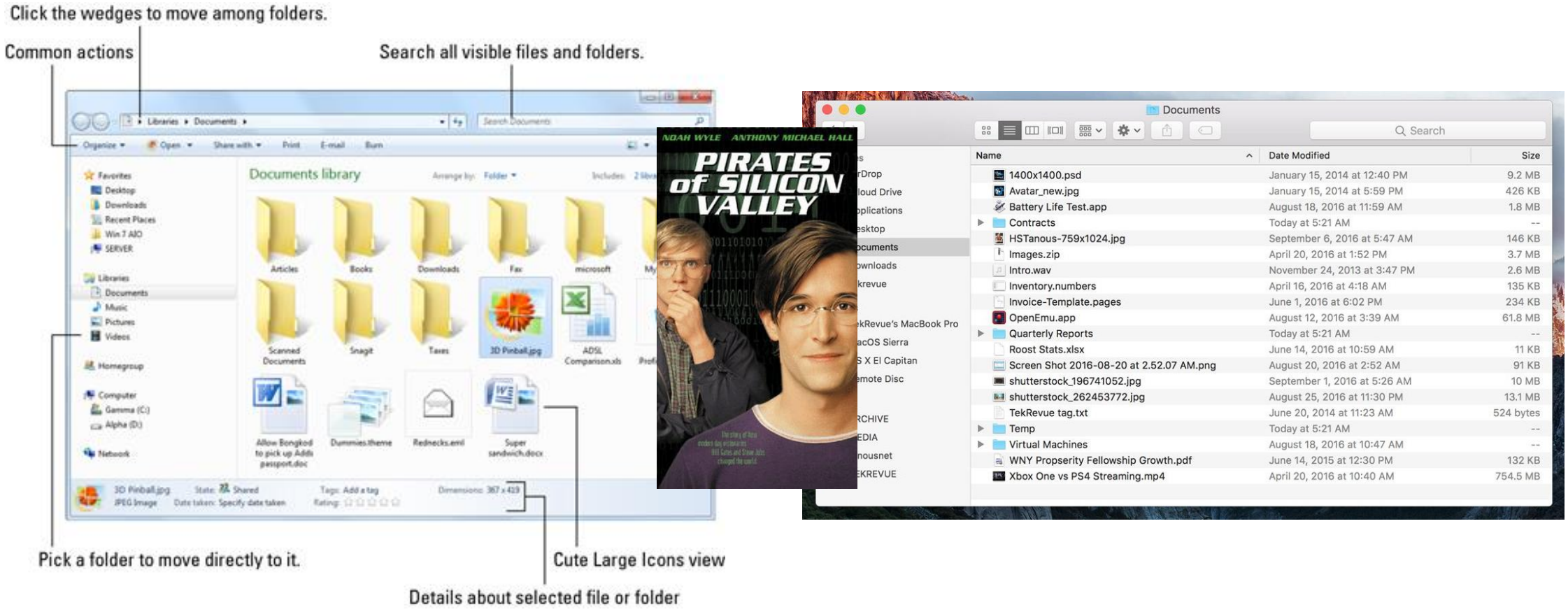
Overview

- Announcements
  - HW4 out and due today
  - Official Final Project documentation on Canvas (PDF)
- File systems review
- Interacting with your Operating System (OS)
  - The `os` and `shutil` modules
  - Short introduction to recursion
- Interacting with external programs
  - The `subprocess` module
- Activity

File systems

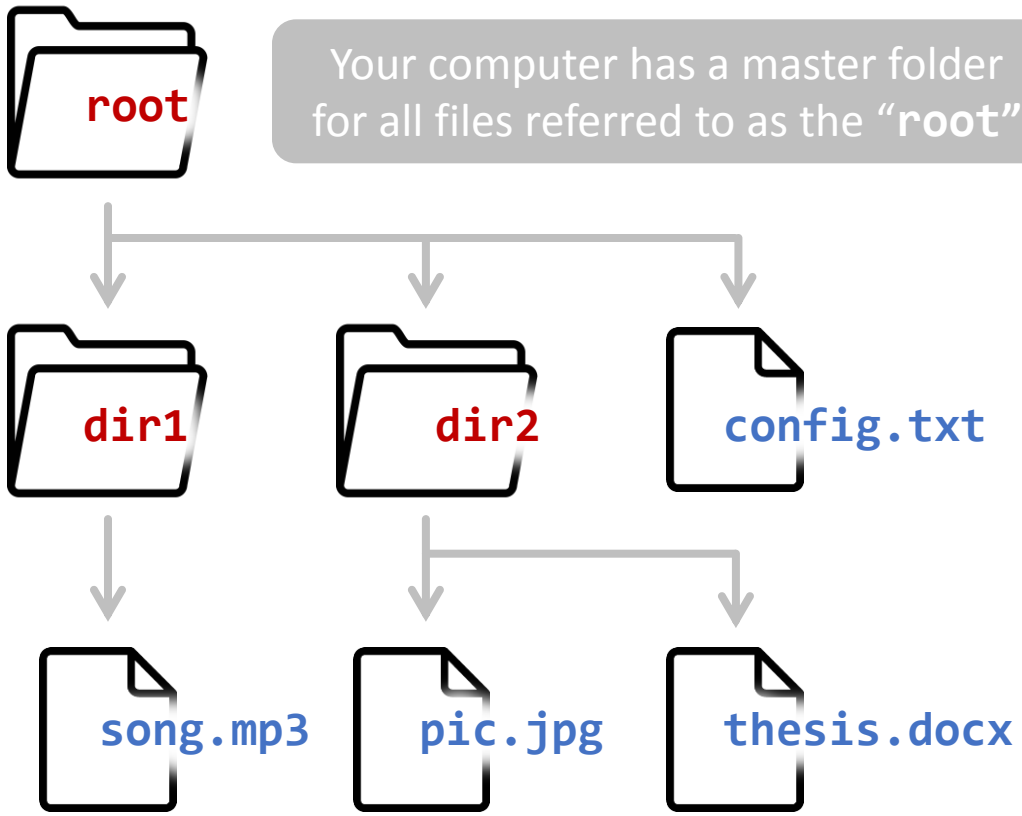
File systems

- The **MacOS Finder** and **Windows Explorer** provide graphical interfaces to Mac/Windows file systems, respectively.



File systems

- A file system is a hierarchical (tree) organization of data on your computer.
- Folders contain files (and other folders) in **parent-child relationships**.



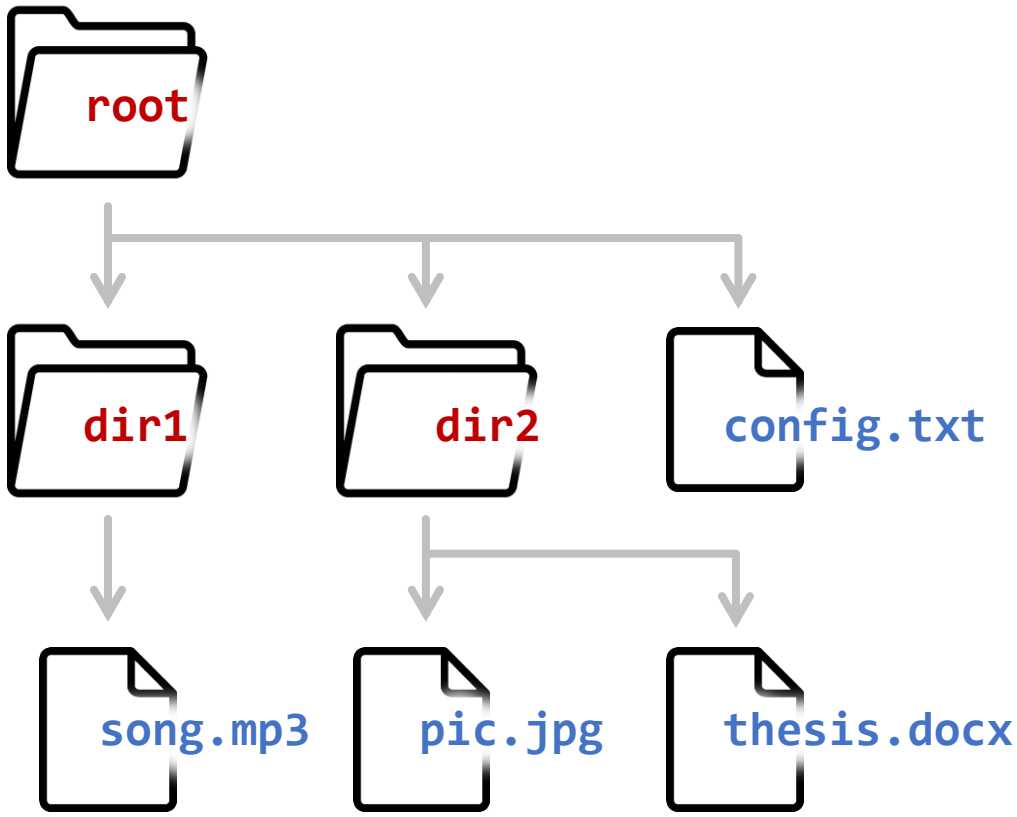
Your computer has a master folder for all files referred to as the "`root`"

The words "folder" and "directory" are used interchangeably

Files are often distinguished from folders by a file extension (`.xyz`): a convenience feature for hinting at file contents and associating them with appropriate programs

File systems

- Another representation of the same data:

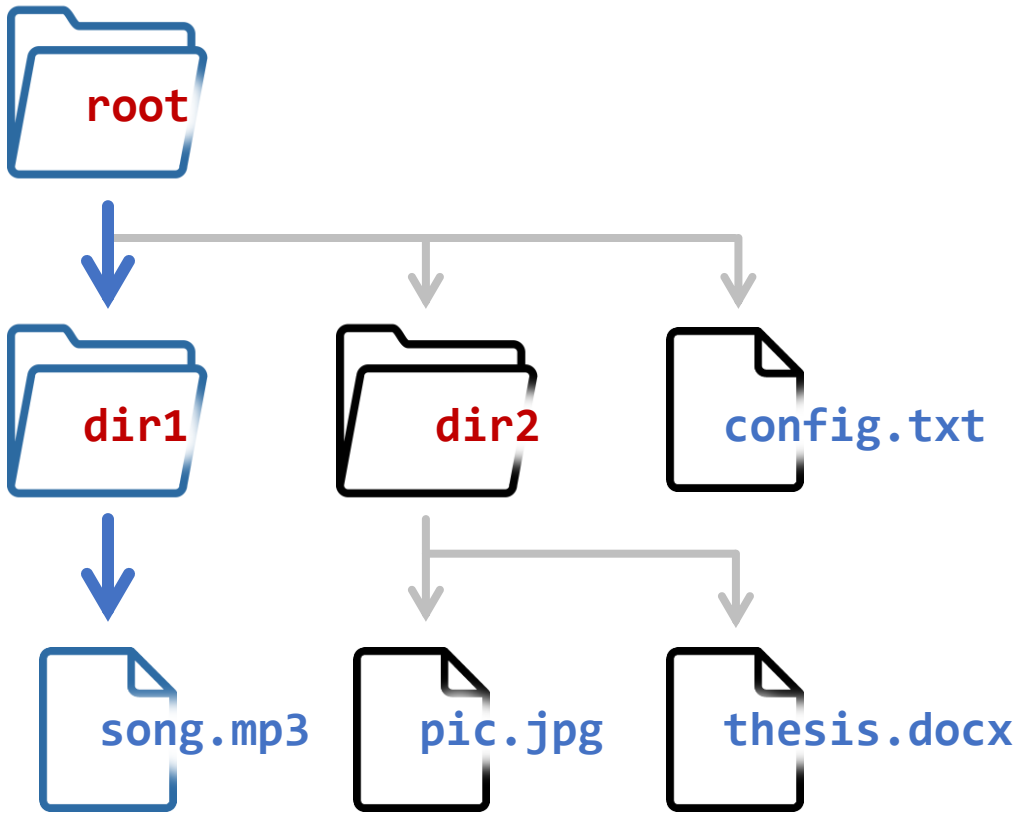


(we'll see that the "tree" command can actually sketch this out for us)

```
root
├─config.txt
├─dir1
│  └─song.mp3
└─dir2
   ├─pic.jpg
   └─thesis.docx
```

Paths

- There is a path in the tree to every folder or file on the computer.
- This path can be represented as a string of text.




```
/dir1/song.mp3
```

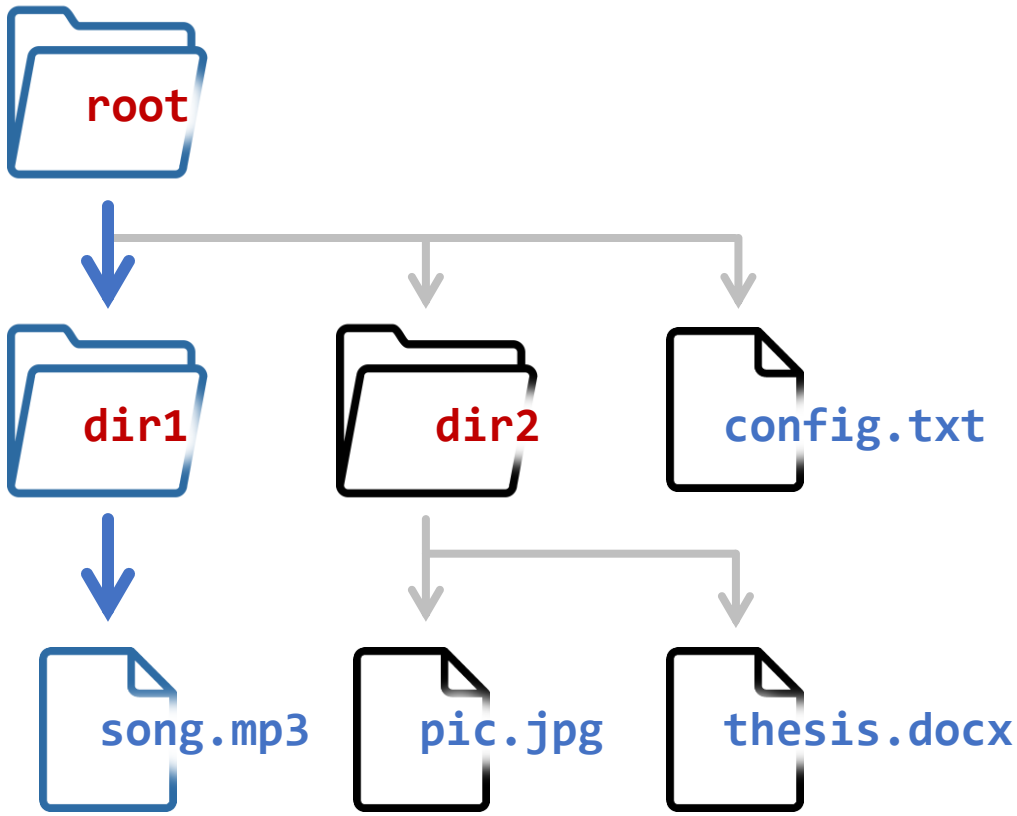
In most operating systems, the root of the hierarchy is represented by "/" (forward-slash) and the same symbol represents parent-child relationships.

```
C:\dir1\song.mp3
```

Windows has a separate root for each hard drive disk (HDD), with the default being "C:". Windows uses "\" (back-slash) for parent-child relationships.

Paths

- Paths that begin at the root are called “absolute paths”.
- Absolute paths are safer; they don’t depend on “frame of reference”.

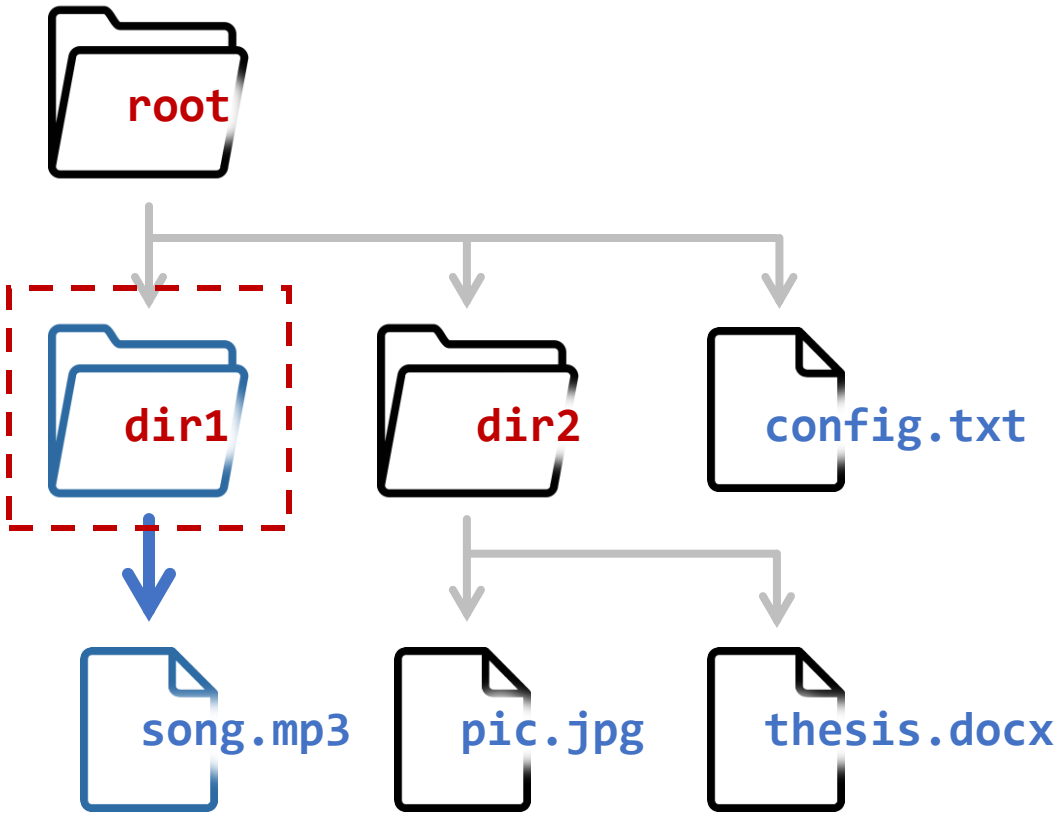



```
/dir1/song.mp3
```

Absolute path to song.mp3

Paths

- All other paths are “relative paths”.
- “Relative” → “Relative to your working directory” (more on that shortly).

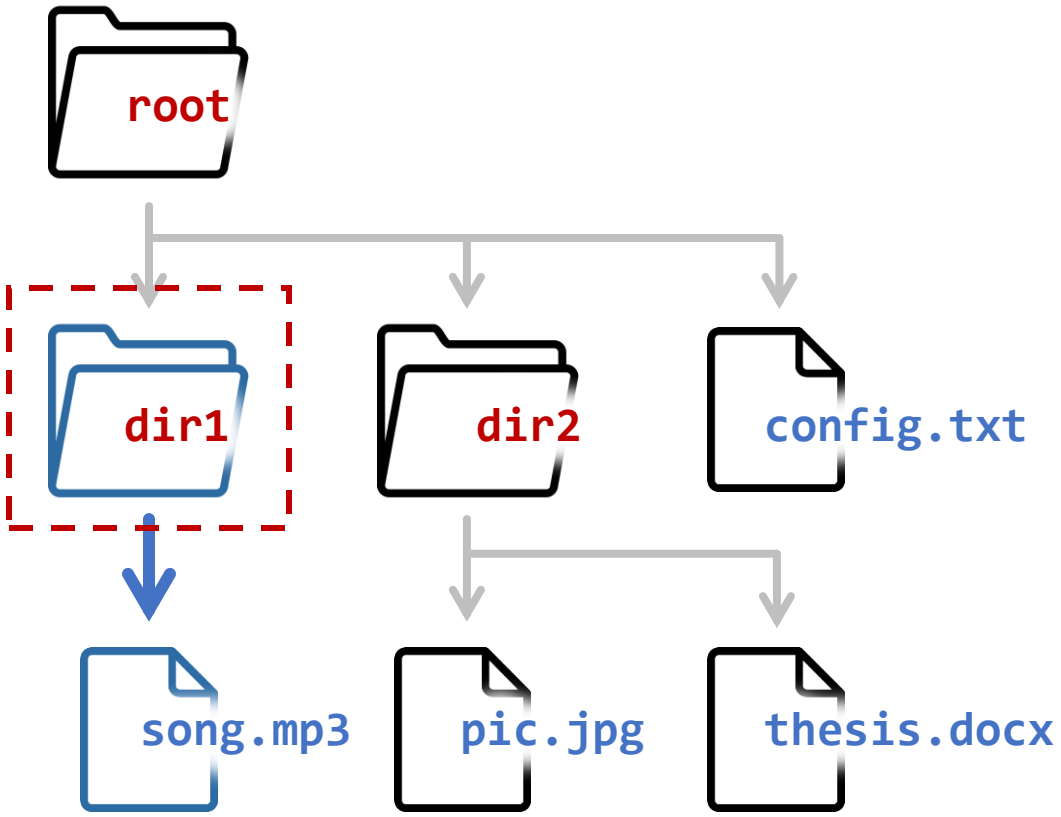


`song.mp3`

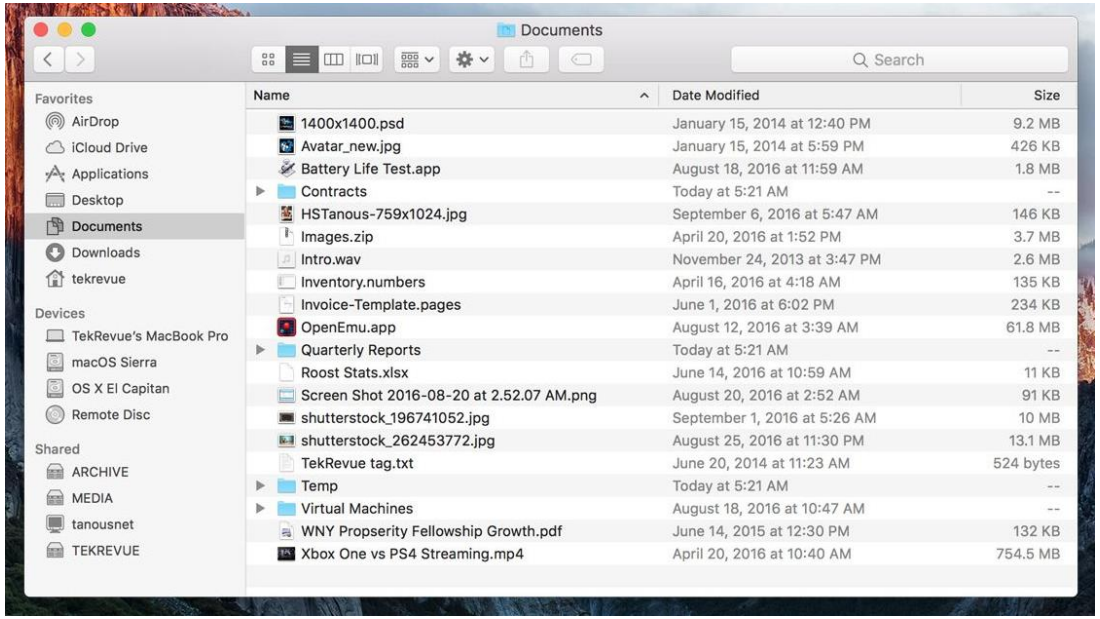
Relative path to song.mp3 if I’m in `dir1/`

Paths

- Relative paths exist because the terminal is “in” a given directory at any specific time (the default is usually your home folder, not the root).



This directory is called the “Working Directory”



Analogous to having finder “open” to `Documents/`

Misc. notes about file systems / paths

- Mac/Linux distinguish between upper vs. lowercase letters (Windows doesn't)
- Your **home folder** is usually a step below the root
  - Windows: `C:\Users\Franzosa`
  - MacOS: `/Users/franzosa`
  - Linux: `/home/franzosa`
- On Mac/Linux, `~` is a shortcut for the absolute path to your home directory
  - `~/Downloads` is equivalent to `/home/franzosa/Downloads`
- On all systems, `..` is a shortcut for "**my parent folder**"
  - Working in `~/Downloads/`, `..` is equivalent to `/home/franzosa` or `~`
- On all systems, `.` is a shortcut for "**here**" (i.e. the **working directory**)
  - Working in `~/Downloads/`, `.` is equivalent to `~/Downloads`

Interacting with operating systems

Interacting with Operating Systems (OSes)

- We've seen some of this already with **`sys`**
  - Read command line arguments with `sys.argv`
  - Use system I/O streams with from `sys.stdin` and `sys.stdout`
- Other options are managed by the modules **`os`** and **`shutil`**
  - https://docs.python.org/3/library/os.html
  - https://docs.python.org/3/library/shutil.html

The `os` module

- Access it like any other module
  - `>>> import os`
- Use it to figure out where your script is running
  - `>>> os.getcwd( )`
  - `‘/home/franzosa/’`
- Or work in a different location
  - `>>> os.chdir( ‘Downloads’ )`
  - `>>> os.getcwd( )`
  - `‘/home/franzosa/Downloads’`

The `os` module

- `os` enables command-line maneuvers and queries from within Python code
  - `os.getcwd( )` is an analog of `pwd` on the command line
  - `os.chdir( )` is an analog of `cd` on the command line
- Python code is the same regardless of your computer's OS
  - Very convenient for writing universal code

os.stat

- Returns information about a file as a **stat_result** object
  - `>>> my_stats = os.stat( “iris.tsv” )`
  - `>>> my_stats.st_size`
  - `4629 # file size in bytes as an int`
  - `>>> my_stats.ctime`
  - `1539036926 # creation time`
- Note, OSes measure time in seconds since Jan 1, 1970
  - The beginning of the “Unix Epoch”
  - Unix times can be conveniently subtracted from one another
  - Can be converted to normal dates and times with the **datetime** module

`os.listdir( path )`

- Returns the file and directory names present in *path* (default="./") as a list
  - `>>> os.list_dir( )`
  - `['iris.tsv', 'iris.txt', 'my_plot.png', 'scatter.py']`
- Similar to running the `ls` command (or `dir` on Windows)

`os.path`

- Contains a collection of useful functions for working with file paths
- A “module within a module”
- Use nested dot syntax to access functions
  - e.g. `os.path.`*`function`*

os.path.join & split

- os.path.join
  - *# joins paths on OS-specific parent-child path separator*
  - >>> os.path.join(“dirname”, “filename”)
  - ‘dirname/filename’
- os.path.split
  - *# splits path on OS-specific parent-child path separator*
  - >>> os.path.split(“dir1/dir2/filename”)
  - [“dir1/dir2”, “filename”]
- Always use these over e.g. “/”.join(“dirname”, “filename”)
  - *Any thoughts why?*

os.path.exists & isdir

- os.path.exists(*path*)
  - *# returns True/False if <path> exists/doesn't exist*
  - >>> os.path.exists("recipe_for_immortality.txt")
  - False
- os.path.isdir(*path*)
  - *# returns True/False if <path> is/isn't a folder*
  - >>> os.path.isdir("/home/efranzosa/Downloads")
  - True

`os.walk( path )`

- One of my favorite Python functions
- ***Yields*** triples of three items:
  - The path to a folder on your computer
  - A list of folders within that folder
  - A list of files within that folder
- Does this recursively for the *path* folder and all folders below *path*

os.walk(*path*)

- Recall our demo folder from Module 0
  - `>>> for items in os.walk( “demo” )`
  - `...  print( items )`

```
demo/
├── README.txt
├── hmp2012
│   └── metadata.tsv
├── gov
│   └── us_constitution.txt
└── words
    ├── words.txt
    └── more_words
        └── wikipedia_top100.txt
```

os.walk(*path*)

- Recall our demo folder from Module 0
  - `>>> for items in os.walk( “demo” )`
  - `...  print( items )`

```
demo/
├── README.txt
├── hmp2012
│   └── metadata.tsv
├── gov
│   └── us_constitution.txt
└── words
    ├── words.txt
    └── more_words
        └── wikipedia_top100.txt
```

```
1st iteration: ('demo', ['gov', 'hmp2012', 'words'], ['README.txt'])
```

os.walk(*path*)

- Recall our demo folder from Module 0
  - `>>> for items in os.walk( “demo” )`
  - `...  print( items )`

```
demo/
├── README.txt
├── hmp2012
│   └── metadata.tsv
├── gov
│   └── us_constitution.txt
└── words
    ├── words.txt
    └── more_words
        └── wikipedia_top100.txt
```

```
1st iteration: ('demo', ['gov', 'hmp2012', 'words'], ['README.txt'])
2nd iteration: ('demo/gov', [], ['us_constitution.txt'])
```

os.walk(*path*)

- Recall our demo folder from Module 0
  - `>>> for items in os.walk( “demo” )`
  - `...  print( items )`

```
demo/
├── README.txt
├── hmp2012
│   └── metadata.tsv
├── gov
│   └── us_constitution.txt
└── words
    ├── words.txt
    └── more_words
        └── wikipedia_top100.txt
```

```
1st iteration: ('demo', ['gov', 'hmp2012', 'words'], ['README.txt'])
2nd iteration: ('demo/gov', [], ['us_constitution.txt'])
3rd iteration: ('demo/hmp2012', [], ['metadata.tsv'])
```

os.walk(*path*)

- Recall our demo folder from Module 0
  - >>> for items in os.walk(“demo”)
  - ... print(items)

```
demo/
├── README.txt
├── hmp2012
│   └── metadata.tsv
├── gov
│   └── us_constitution.txt
└── words
    ├── words.txt
    └── more_words
        └── wikipedia_top100.txt
```

```
1st iteration: ('demo', ['gov', 'hmp2012', 'words'], ['README.txt'])
2nd iteration: ('demo/gov', [], ['us_constitution.txt'])
3rd iteration: ('demo/hmp2012', [], ['metadata.tsv'])
4th iteration: ('demo/words', ['more_words'], ['words.txt'])
```

os.walk(*path*)

- Recall our demo folder from Module 0
  - `>>> for items in os.walk( “demo” )`
  - `...  print( items )`

```
demo/
├── README.txt
├── hmp2012
│   └── metadata.tsv
├── gov
│   └── us_constitution.txt
└── words
    ├── words.txt
    └── more_words
        └── wikipedia_top100.txt
```

```
1st iteration: ('demo', ['gov', 'hmp2012', 'words'], ['README.txt'])
2nd iteration: ('demo/gov', [], ['us_constitution.txt'])
3rd iteration: ('demo/hmp2012', [], ['metadata.tsv'])
4th iteration: ('demo/words', ['more_words'], ['words.txt'])
5th iteration: ('demo/words/more_words', [], ['wikipedia_top100.txt'])
```

The 5th iteration is the FINAL iteration because we have explored every folder inside of the path we initially provided to the `os.walk( )` function.

`os.walk( path )`

- ***Yields*** triples of three items:
  - The path to a folder on your computer
  - A list of folders within that folder
  - A list of files within that folder
- Note that I said "**yields**" not "**returns**"
- `os.walk( )` is a special type of function called a *generator*
  - Generators return multiple values one-at-a-time
  - This lets us iterate over them in a `for` loop

Example of a generator

script.py (*open in Atom*)

```
import sys

def square_values( values ):
    for v in values:
        yield float( v )**2

for x in square_values( sys.argv ):
    print( x )
```

Turn a function into a generator with the `yield` operator

(*a terminal*)

```
$ python script.py 1.0 2.0 3.0

  1.0
  4.0
  9.0
```

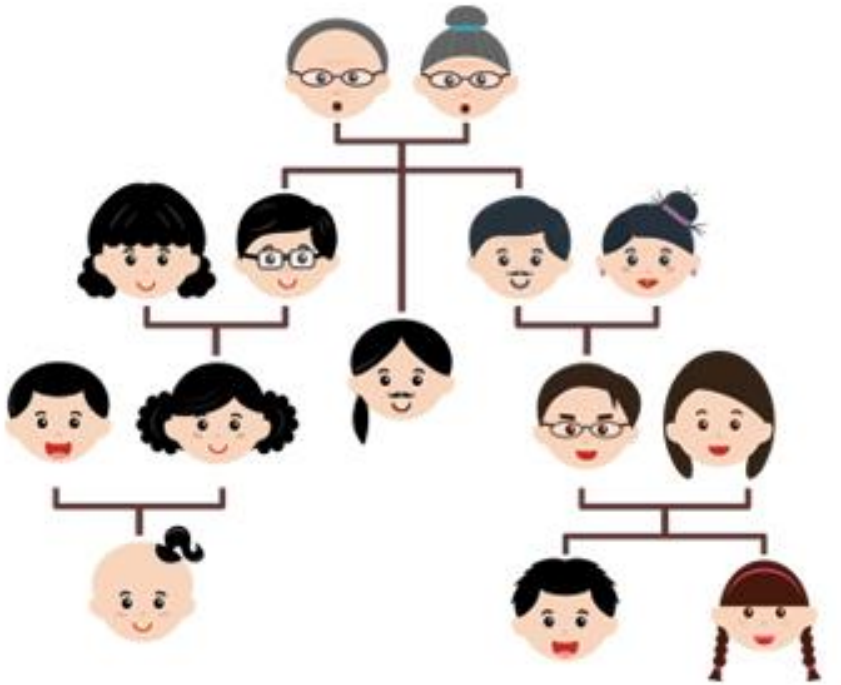
A brief introduction to recursion

- `os.walk( )` also happens to be an excellent example of *recursion*:
  - An important (general) concept in computer programming
- Recursion is a way to solve problems that can be divided into similar *sub-problems* whose solutions are not yet known.
- Properties of a recursive function:
  - Calls itself to solve sub-problems of a non-trivial problem
  - Returns an exact solution for trivial problems (so-called “base cases”)

```
def recursive( problem ):
    if not is_trivial( problem ):
        return recursive( subproblem1 ) + recursive( subproblem2 )
    else:
        return trivial_solution
```

A brief introduction to recursion

- How many descendants does a given person have (counting him/herself)?



```
def count_descendants( person ):
    children = person.get_children( )
    if len( children ) > 0:
        count = 1
        for child in children:
            count += count_descendants( child )
        return count
    else:
        return 1
```

A brief introduction to recursion

script.py (*open in Atom*)

```
import os
import sys

# recursively find all files below <path>
def find_files( path ):
    files = []
    for name in os.listdir( path ):
        full = os.path.join( path, name )
        if os.path.isdir( full ):
            files += find_files( full )
        else:
            files += [full]
    return files

for p in find_files( sys.argv[1] ):
    print( p )
```

(*a terminal*)

```
$ python script.py demo/

  demo/gov/us_constitution.txt
  demo/hmp2012/metadata.tsv
  demo/README.txt
  demo/words/more_words/wikipedia_top100.txt
  demo/words/words.txt
```

File manipulation overview

From the `os` module itself	
Function call	*What it does*
`os.rename( name1, name2 )`	Rename the file/folder with *name1* to *name2*
`os.remove( path )`	Remove the file (not folder) located at *path*
`os.mkdir( name )`	Make a folder named *name*
`os.rmdir( path )`	Remove the folder located at *path*

From the `shutil` module (import `shutil` to use these functions)	
Function call	*What it does*
`shutil.copy( path1, path2 )`	Copy (recursively) everything at *path1* to *path2*
`shutil.move( path1, path2 )`	Move (recursively) everything at *path1* to *path2*

Interacting with software

Interacting with software

- The `subprocess` module provides three important capabilities:
  - Make *any* command-line call from within a Python program
  - Determine if the command finished successfully
  - Capture the output of the command (for subsequent processing)
- Centered on a single function, `subprocess.run`, with many options
- Convenience functions call `subprocess.run` with different defaults
  - `subprocess.call`
  - `Subprocess.check_output`
- Can read more online at:
  - `https://docs.python.org/3/library/subprocess.html`

subprocess.call

- Runs a command at the command-line and returns an “exit code”
- “0” indicates success
  - `>>> subprocess.call( “ls” )`
  - `0`
- Any other number (often 1-255) indicates some error occurred
  - `>>> subprocess.call( “wc recipe_for_immortality.txt” )`
  - `1`

subprocess.call

- Setting `shell=True` allows us to perform complex piped commands and use system variables (such as “$HOME”)
  - `>>> subprocess.call( “ls $HOME | wc -l”, shell=True )`
  - `0`
- Note: this is frowned upon in professional code for security reasons, but is OK for things you’re writing and executing yourself

subprocess.call

- If you're NOT using `shell=True`, you can provide your command as a list of strings which will be automatically joined
  - `>>> my_path = "iris.tsv"`
  - `>>> subprocess.call( ["wc", "-l", my_path] )`
  - `1`
- This makes interspersing commands and variables a bit easier

subprocess.check_output

- Runs the command and returns the standard output
  - `>>> subprocess.check_output( “wc -l iris.tsv”, encoding=“utf-8” )`
  - `‘152 iris.tsv\n’`
- By default, subprocess.check_output returns individual “bytes”
  - Setting `encoding=“utf-8”` provides more traditional string formatting
- Output is provided as one, long string (with newlines)
  - `>>> subprocess.check_output( “ls demo”, encoding=“utf-8” )`
  - `‘gov\nhmp2012\nREADME.txt\nwords\n’`
- *How could we process the output line by line?*

subprocess.check_output

- A common (if verbose) coding motif:

```
import subprocess

for line in subprocess.check_output( command, encoding="utf-8" ).split( "\n" ):
    # do something with <line>
```

Activity

`code_count.py`

- On Canvas you'll find an almost-complete script called `code_count.py`
- This script uses concepts from today's lecture to count the number of lines in Python scripts located below a certain directory.
  - In case you want to tell someone "I coded *N* lines of Python in BST 273"
- The script needs a few more lines of code in order to function properly.
  - All things you've seen before today, albeit in other contexts.
- Take a few minutes to look over the script amongst yourselves, then we'll discuss it (and the necessary changes) together.

`code_count.py`: Extension 1

- See if you can modify the script to compute the total SIZE of all Python files below a given folder.
- HINT: *You can do this without making any special system calls using one of the features of the Python* `os` *module.*

`code_count.py`: Extension 2

- Modify the script so that it is not specific to Python files. Instead, have the user pass file-extensions-of-interest as arguments of the program.

`code_count.py`: Extension 3

- Modify the script to only count lines of Python code that include an `import` statement; do not open/parse any of the files using Python.
- HINT: *How would you count the* `import` *statements in a single Python file using a command-line chain?*

Extras

A brief introduction to recursion

- Consider the factorial of a number, n, written n!
  - n! = n × (n-1) × (n-1) × … 2 × 1
  - n! = n × (n-1)!
- This is a recursive function for computing factorials:

```
def factorial( n ):
    answer = n
    if n > 1:
        answer *= factorial( n - 1 )
    return answer
```