Some General Python Skills

Command-line scripts;
working with disk files

Automating Tasks, Working With Datasets

- Scripts are *good* for repeated activities
- Python programs saved as standalone scripts
  - Use from command line
  - Use from a GUI?
- Scripts are *good* for working with large datasets
- Getting input from diskfiles
- Sending output to diskfiles, for later use

Standalone Scripts

- Basics
  - Just save your python statements in a file
    - any text editor will do
      - some good choices: SciTE, geany, notepad++
- Run from command line
  - Start shell ("command prompt") first
  - Provide script file as argument to "python"
    - or maybe "py"
- Run from GUI
  - Find icon for file, double-click it
  - PROBLEM: output doesn't stay long enough to read!

Standalone Scripts 2

- Shell/command line:
  - Linux, MacOS:
    Initial comment line specifies use as a command
    - #!/usr/bin/env python3
  - Windows: comment has no effect (no harm)
- GUI usage (Windows) :
  - Add a final "input()" statement that makes the program wait for garbage input before closing

try it:

- A trivial program:

```
y = input('y? ')
z = x + y
print(z)

z2 = float(x) + float(y)
print(z2)
```

- Try with the *beginning* comment:

```
#!/usr/bin/env python3
```

- Try with the *ending* input:

```
input('waiting...')
```

The sys Module

- Interface to Python interpreter, OS shell
  - I/O objects:
    - sys.stdin
    - sys.stdout
    - sys.stderr
  - Command-line arguments:
    - sys.argv
  - Standard termination method:
    - sys.exit()
- Other utility functions, variables

Supporting Standalone Scripts –

"if __name__ == '__main__':"

- Python scripts can be executed from the command line, with optional arguments added
  - The system variable "__name__" is automatically set to the value "__main__"
- If script is imported, `__name__` is set to the filename instead of "__main__"
- Save this one-line script as "testname.py", and try **import**ing versus **run**ning it:
  -
```
print('__name__ is', __name__)
```

"if __name__ == '__main__':"

Typical Usage

- Define functions that do all desired actions when invoked
- Define special function (named "`main()`" by convention and long history) that does things, including invoking the other functions, as appropriate
- If `(__name__ == '__main__')` then invoke "`main()`" and give it the command-line arguments that are found in `sys.argv`

Seeing, Using Command-Line Arguments

- Short script:

```
def main(argv=[__name__]):
    for arg in argv:
        print(arg)
    if len(argv) > 1:
        spec_dir = argv[1]

if __name__ == '__main__':
    import sys
    sys.exit( main(sys.argv) )
```

main() runs as a standalone script, or can be invoked interactively in "jupyter qtconsole"

- This shows all supplied arguments, and chooses the 1st option as the desired directory

" `main()` " – *Why do it this way?*

- *Seems clumsy, lots of fiddling about for no purpose....*

- *an answer*: This is "dividing the task into separate functions", carried to its logical conclusion
  - Permits each piece of the script to be tested interactively
  - Allows easy reuse of parts of the script

  - Typically, the `main()` function handles any command-line arguments, calls other functions to perform the script's task with given parameters

Where Is the Script Located?

- Script file is in a *directory* (folder) on the computer
- Any data files are in a directory on the computer
- Where?
  What if they're in different directories?

The os Module

- Many OS-independent operations
  - Works on Windows, Mac, Linux, whatever else...
  - **os.getcwd()** – get current working directory
  - **os.curdir** – reference to the current directory
    - (*a.k.a.* **'.'**)
  - **os.listdir()** – contents of a directory
    - **os.listdir('/')** – contents of root directory
    - **os.listdir(os.getcwd())** – contents of current dir
    - **os.listdir(os.curdir)** – also contents of current dir

Getting Data From a File

- Disk files are represented in programs by named **handles**
- A file *name* is associated with a handle by *opening* it
  - Example:
    ```
    the_handle = open('filename.txt', 'r')
    ```
  - the `'r'` specifies that file is opened for *reading*
  - the `'filename.txt'` can include a *path* if the file is in a different directory
- File handles must be *closed* when finished
  - `the_handle.close()`

Reading the Data File

- Opened file handle has reading *methods*
- Read the entire file at once
  - Example:
    ```
    all_contents = the_handle.read()
    ```
- Read each line of text, one at a time
  - Example:
    ```
    a_line = the_handle.readline()
    ```
- Read all lines of text at once
  - Example:
    ```
    lines_list = the_handle.readlines()
    ```

try it:

- Download a data file:
  - https://montcs.bloomu.edu/Datasets/wordlist.txt
  - Save it someplace
- Python program:
  - Find current working directory
  - Find location of saved data file
- Open and read data file
  - One line at a time
  - Count lines
- Close the handle

(Review) File Reading

- Open a file for reading and use it:
  - Open the file, make a *filehandle*
  - Use the filehandle
  - Close the filehandle

```
filehandle = open('filename', 'r')
alldata = filehandle.read()  # or other ops
filehandle.close()
```

- Or, create an "opened-file" block:
  -

```
with open('filename', 'r') as filehandle:
   alldata = filehandle.read() # or other ops
```

The os.walk() Function

- How to *find* the source directory?
- os.walk() recursively explores a filesystem directory
- At each step, provides
  - a root – full path from starting point to current directory
  - a list of subdirectories – these will get explored in subsequent steps
  - a list of files

"*websearch-inputs*" *Lost in a Directory Tree*

- Desired source directory is buried in a "tree" of subdirectories starting at "t"
- (Nonstandard) "tree" program depicts directory structures graphically

```
519] tree t/
t/
├── dA
│   ├── ddaa
│   └── ddab
├── dB
│   ├── ddba
│   │   └── ddbaa
│   └── ddbb
│       └── websearch-inputs
│           ├── online-sources.txt
│           └── school-URLs.txt
└── dC
    └── ddca

10 directories, 2 files
```

"Top-down" Directory Walk

```
for root, dirs, files in os.walk('t'):
    print('{:30s} {}'.format(root, dirs))
```

```
t                              ['dA', 'dB', 'dC']
t/dA                           ['ddaa', 'ddab']
t/dA/ddaa                      []
t/dA/ddab                      []
t/dB                           ['ddba', 'ddbb']
t/dB/ddba                      ['ddbaa']
t/dB/ddba/ddbaa                []
t/dB/ddbb                      ['websearch-inputs']
t/dB/ddbb/websearch-inputs     []
t/dC                           ['ddca']
t/dC/ddca                      []
```

- "`os.walk('t')`" accesses each subdirectory in turn, showing subdir's contents

Use os.walk() To Find Source Directory

```
starting_path = '/home/bobmon/mushroom/2018-01/215/t'
spec_dir = 'websearch-inputs'
spec_path = None
for root,dirs,files in os.walk(starting_path):
    if spec_dir in dirs:
        spec_path = os.path.join(root, spec_dir)
        break

print(spec_path)
contents = os.listdir(spec_path)
print(contents)
```

```
/home/bobmon/mushroom/2018-01/215/t/dB/ddbb/websearch-inputs
['online-sources.txt', 'school-URLs.txt']
```

try it:

- List the files and directories on your P: drive
- List the files and directories on your home directory

(Review) The try-except Block

- Some legal operations may fail due to external problems
  - *e.g.* missing data, crashed network, etc.
- Use "try-except" to attempt operations, capture any errors ("exceptions")
- The "else" clause performs processing only if the "try" succeeds
  - Compare to "else" in an "if-else" block
- The "finally" clause provides clean-up operations regardless of success or failure

Silly Example: try-except-else-finally

```
x = 7
for y in range(3, -2, -1):
    try:
        z = x / y
    except:
        print('Oops.')
        z = 9e999
    else:
        print('Yay!')
    finally:
        print(x, y, z)
print('Done')
```

- Code on left gives results below:

```
Yay!
7 3 2.3333333333333335
Yay!
7 2 3.5
Yay!
7 1 7.0
Oops.
7 0 inf
Yay!
7 -1 -7.0
Done
```

Put things together:

- Download and save a data file
- Write, run a script to:
  - Get the filename from the command line
  - try: open the file
    - » if the name can't be opened, abandon the program
  - Read the file's lines
  - Count the lines and the lengths of the lines
  - Calculate the average line length, and report it
  - Close the file handle

Done