Searching for Terms on Webpages

Develop a "web scraping and searching" script

Motivation

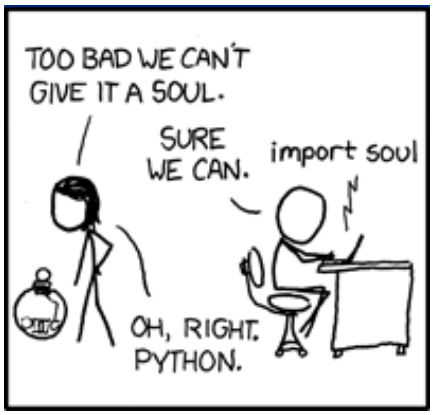
- This activity was originally developed at the request of a Digital Forensics professor who wished to track the appearance of certain “phrases of interest” in online news sources.
- The input and output formats were specified in a general way, and help to provide reproducibility.
- By the nature of the project, the answers found will change from week to week.

The Task:

- Copy some webpages of interest, and save them into text files
  - They happen to be online news sites
- Search each webpage for specific terms
  - Multiple terms
  - Might be regular expressions, or constant phrases
  - Stored in a file
- Save a report of the matches between webpages and search terms in a file
  - File should be readable into a spreadsheet
  - ".csv" (comma-separated-values) format is human-readable, convenient

Some Python Modules

- You should have seen these before:
  - os, os.path
  - sys
- These may be new:
  - urllib.request
  - ssl
  - csv



Obtaining and Saving Webpages

- Webpage "address" specified by URL
  - "Uniform Resource Locator"
- Some URLs start with "http" – normal webpage
- Some URLs start with "https" – webpage protected by SSL encryption

- All URLs will be listed in one or more text files, which are stored in a directory
  - Must (a) read the text files, and (b) collect the URLs of interest
  - Must (c) group the URLs according to text file

Building the task solution, a step at a time

- Following slides contain multiple steps
- Try them out in an interactive environment such as ipython/jupyter
- Final work in a script

Downloading and Copying the Webpages

- URLs imply Web access
  - Don't need a web browser
- A very useful Python module:

```
import urllib.request
```

  - High-level modules, requests URL contents and returns objects holding those contents

Accessing the Webpage

- `urllib.request.urlopen()`)
  - When successful, returns a "urllib.response" object
- Example:
  - `response = urllib.request.urlopen('http://bloomu.edu')`
  - Treat response like a file object:

```
lines = [ l for l in response.readlines() ]
for l in lines:
    print(l)
```

A URL Issue

- URLs must start with appropriate protocol
  - "`http://`" for normal webpages
  - "`https://`" for SSL-encrypted webpages
  - Other possible protocols
    - "`ftp://`" for File Transfer Protocol objects
    - "`file://`" for local files (but why bother?)
- Webpage URLs generally start with "`http://`" or "`https://`"
  - "`https://`" is becoming the norm
  - But "`https://`" downloads are *encrypted*

Encrypted Websites

- Communications with "https://" sites are encrypted while in transit
- Encryption handled by
  - SSL (older Secure Sockets Layer)
  - TLS (newer Transaction Level Security)
- Requires valid "SSL certificate"
- Try:

```
site = 'https://google.com'
response = urllib.request.urlopen(site)

site = 'https://montcs.bloomu.edu'
response = urllib.request.urlopen(site)
```

Handling SSL Encryption

- Another Python module to the rescue!

```
import ssl
```

- Used with `urllib.request.urlopen()`
  - Simple usage:
    don't try to verify SSL connections or most of the features that SSL and the `ssl` module provide
- Example:
  - ```
    site = 'https://montcs.bloomu.edu'
    response = urllib.request.urlopen( site, \
       context=ssl._create_unverified_context() )
    ```

Another Issue

- Not all webpages are successfully loaded
  - And maybe you can't tell until you try
- A Python feature for things that may fail:
  the `try – except` block
  -
    ```
    try:
       # do something that may fail
    except:
       # handle failure somehow, perhaps by
       # trying again, or by reporting an error
       # and continuing on, or …
    ```
    ```

Retrieving the Webpage's Contents

- Read the "response object", like a file object
  - Get the response contents as a single long string:

```
string = response.read()
lines = string.split(b'\n')
```

    - *What is `b'\n'` ??? --- see next slide*

  - Get the response contents as a list of lines:

```
lines = []
for l in response.readlines():
    lines.append(l)
```

  - ...or as a *list comprehension*:

```
lines = [l for l in response.readlines()]
```

Bytes versus Strings

- Python3: contents are read as a string of *bytes*
  - 8-bit values representing any desired data
- Byte strings must be *decoded* into text strings
  - `.decode()` method: convert byte strings into text
  - ...and `.encode()` converts strings to bytes...
- What kind of text encoding was used?
  - Not necessarily apparent at the beginning
  - Another use for a try-except block – try encodings until one works

Converting Bytes to Text

- Possibilities:
  - **utf-8** is very common on webpages
    - Handles non-English alphabets
  - **ascii** is the basic 7-bit encoding used by Python2 and many older programs
    - Limited to (modern) English text
  - **latin-1** is found on Microsoft-sourced webpages
  - **EBCDIC**, **utf-16**, others available but not common

Example – urllib.request.urlopen()

```
import urllib.request
response = urllib.request.urlopen('http://bloomu.edu')
print('response:', type(response))

pageb = response.read()
print('pageb:', type(pageb), len(pageb))

page = pageb.decode('utf-8')
print('page:', type(page), len(page))
```

```
response: <class 'http.client.HTTPResponse'>
pageb: <class 'bytes'> 40339
page: <class 'str'> 40324
```

(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
```

(Review) The os Module

- Many OS-independent operations
  - Works on Windows, Mac, Linux, etc…
  - **os.getcwd()** – get current working directory
    - **os.curdir** – the current directory (*a.k.a.* **'.'**)
  - **os.listdir('?')** – contents of a directory '?'
    - **os.listdir(os.getcwd())** – contents of current dir
  - **os.walk()** – recursively explore directory structure
    - not needed for this project

(Review) The os Module

- **os.path** – sub-module with directory-path-oriented operations
  - **os.path.join('?')** – join elements into a proper path
    - Because different OSes use different separators…
  - **os.path.abspath('?')** – full path to a specific name
    - **os.path.abspath(os.curdir)** – like os.getcwd()
  - **os.path.expanduser('?')** – home directory of specified user
    - Use "~" to specify the current user

Getting the URLs From the User

- User provides “spec_path”, a directory containing specification-files
  - Command-line option?
  - Use “`input()`” to ask for it?
  - Hardcode it into the script? (ugh)
- Get names of specification-files from directory
  - Use Python's "os" module:

```
files = os.listdir( spec_path )
```

Getting the URLs – B

- Next, read contents of each source file that's in the source directory
  - Path *to* file name must *include* the source directory
  - Format of "spec_path + filename" differs based on operating system
    - Linux, MacOS: "/" --- Windows: "\"
  - Let "os" handle the difference:

```
for f in files:
    filepath = os.path.join( spec_path, f )
```

Getting the URLs – C

- Read each source file's lines into a list
  - First line of source file specifies a "save" file for the captured webpages
  - Each remaining line is one of the desired URLs
  - Blank lines are ignored
- Reading lines from a text file includes the "newline" character at the line's end
  - Should be stripped off:

```
line = filehandle.readline()
cleanline = line.strip()
```

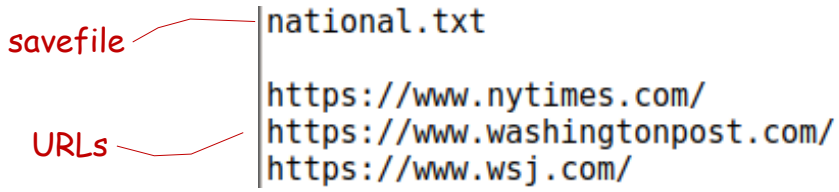
    - Also remove leading, trailing whitespace – not likely to actually be part of the desired input

(Review) Cleaning Text Strings

- Text strings may contain excess text at the beginning or end, or in the middle
  - whitespace, newline character, other text
- String methods **.lstrip()**, **.rstrip()** , **.strip()**
  - Remove unwanted text from *left* side/beginning, from *right* side/end, or from *both* sides
  - "unwanted text" defaults to “any whitespace” (including newlines)
- **.replace()** returns an edited copy of the string
  - Replace target string with some other string
  - Replacement string can be empty, to delete text

Getting the URLs – D

- Save the URL lists in a dictionary
  - Keyed by "savefile" name



- Challenge:
  what if multiple source files specify the *same* save file?
  - Don't want to overwrite one URL list with a different one, want to merge them together

Getting the URLs – E

- (actually, this is the script's *first* step)
- What is the source directory's name?
  - "Hardcode" a default value?
- It would be nice to allow a different location
  - Command-line options are often used for this
  - Python's "sys" module provides access to command-line options, in a list
    "`sys.argv` " supplied to script when it's run
  - Options list can be processed "by hand", or by using the "getopt" module – for now, "by hand" is easier

(Review) 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 after importing script as a module

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

The os.walk() Function

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

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

- Desired source directory is buried in a "tree" of subdirectories starting at "t"
- The 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']
```

(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
```

Lab Exercise Again

- Steps F-K of the exercise obtain and save the webpages
- Some new modules
- Deal with more "hair" – *i.e.*, real-world details that need to be handled
  - Old-time slang: "This is a hairy problem!"
- Final step, for now:
  save the webpages as text strings, to the desired output files