Programming Principles in Python (CSCI 503)

Scripts

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)

Regular Expressions

- AKA regex
- A syntax to better specify how to decompose strings
- Look for patterns rather than specific characters
- Metacharacters: `.` `^` `$` `*` `+` `?` `{` `}` `[` `]` `\` `|` `(` `)`
  - Repeat, one-of-these, optional
- Character Classes: `\d` (digit), `\s` (space), `\w` (word character), also `\D`, `\S`, `\W`
- Digits with slashes between them: `\d+/\d+/\d+`

Regular Expression Methods

Method/ Attribute	Purpose
`match()`	Determine if the RE matches at the beginning of the string.
`search()`	Scan through a string, looking for any location where this RE matches.
`findall()`	Find all substrings where the RE matches, and returns them as a list.
`finditer()`	Find all substrings where the RE matches, and returns them as an iterator.
`split()`	Split the string into a list, splitting it wherever the RE matches
`sub()`	Find all substrings where the RE matches, and replace them with a different string
`subn()`	Does the same thing as sub(), but returns the new string & number of replacements

[Deitel & Deitel]

Regular Expresion Examples

- ```
  s0 = "No full dates here, just 02/15"
  s1 = "02/14/2021 is a date"
  s2 = "Another date is 12/25/2020"
  s3 = "April Fools' Day is 4/1/2021 & May the Fourth is 5/4/2021"
  ```
- `re.match(r'\d+/\d+/\d+',s1) # returns match object`
- `re.match(r'\d+/\d+/\d+',s2) # None!`
- `re.search(r'\d+/\d+/\d+',s2) # returns 1 match object`
- `re.search(r'\d+/\d+/\d+',s3) # returns 1! match object`
- `re.findall(r'\d+/\d+/\d+',s3) # returns list of strings`
- `re.finditer(r'\d+/\d+/\d+',s3) # returns iterable of matches`
- ```
  re.sub(r'(\d+)/(\d+)/(\d+)',r'\3-\1-\2',s3)
                        # captures month, day, year, and reformats
  ```

Files

- A file is a sequence of data stored on disk.
- Python uses the standard Unix newline character (`\n`) to mark line breaks.
  - On Windows, end of line is marked by `\r\n`, i.e., carriage return + newline.
  - On old Macs, it was carriage return `\r` only.
  - Python **converts** these to `\n` when reading.

Files and Jupyter

- You can **double-click** a file to see its contents (and edit it manually)
- To see one as text, may need to right-click
- **Shell commands** also help show files in the notebook
- The `!` character indicates a shell command is being called
- These will work for Linux and macos but not necessarily for Windows
- `!cat <fname>`: print the entire contents of `<fname>`
- `!head -n <num> <fname>`: print the first `<num>` lines of `<fname>`
- `!tail -n <num> <fname>`: print the last `<num>` lines of `<fname>`

Reading Files

- Use the `open()` method to open a file for reading
  - `f = open('huck-finn.txt')`
- Usually, add an `'r'` as the second parameter to indicate read (default)
- Can iterate through the file (think of the file as a collection of lines):
  -
```
f = open('huck-finn.txt', 'r')
for line in f:
    if 'Huckleberry' in line:
        print(line.strip())
```
- Using `line.strip()` because the read includes the newline, and print writes a newline so we would have double-spaced text
- Closing the file: `f.close()`

Parsing Files

- Dealing with different formats, determining more meaningful data from files
- txt: text file
- csv: comma-separated values
- json: JavaScript object notation
- Jupyter also has viewers for these formats
- Look to use libraries to help possible
  - `import json`
  - `import csv`
  - `import pandas`
- Python also has pickle, but not used much anymore

Writing Files: Use with statement

- `outf = open("mydata.txt", "w")`
- Methods for writing to a file:
  - `print(<expressions>, file= outf)`
  - `outf.write(<string>)`
  - `outf.writelines(<list of strings>)`
- Make sure to **close** the file at the end: `outf.close()`
- With statement does "enter" and "exit": don't need to call `outf.close()`
  -
```
with open('output.txt', 'w') as outf:
    for k, v in counts.items():
        outf.write(k + ': ' + v + '\n')
```

Assignment 4

- Illinois Climate Data
- Reading & Writing Files
- Iterators
- Numeric Aggregation (think about comprehensions)
- Formatting Strings

Command Line Interfaces (CLIs)

- Prompt:
  - $
  - 

- Commands
  - `$ cat <filename>`
  - `$ git init`
- Arguments/Flags: (options)
  - `$ python -h`
  - `$ head -n 5 <filename>`
  - `$ git branch fix-parsing-bug`

Command Line Interfaces

- Many command-line tools work with stdin and stdout
  - `cat test.txt # writes test.txt's contents to stdout`
  - `cat # reads from stdin and writes back to stdout`
  - `cat > test.txt # writes user's text to test.txt`
- Redirecting input and output:
  - `<` use input from a file descriptor for stdin
  - `>` writes output on stdout to another file descriptor
  - `|` connects stdout of one command to stdin of another command
  - `cat < test.txt | cat > test-out.txt`

CLI Help/Usage

- No universal method
  - no arguments: `git`
  - `-h` or `--help`: `python -h`
  - help subcommand: `git help push`
- Usage strings often include information about <required> and [optional] arguments
  - cat: `usage: cat [-benstuv] [file …]`
  - python: `usage: python ... [-c cmd | -m mod | file | -] [arg]`
  - git: `usage: git [—version] … <comamnd> [<args>]`

Consoles, Terminals, and Shells

- Originally:
  - Console: hardware physically connected to host (e.g. maintenance)
  - Terminal: hardware that connects to the host (may be remote)
- Today: Consoles and terminals are **virtual**, effectively emulating the physical versions

- Shell: program that runs in the terminal
  - interacts with users
  - runs other programs
  - e.g. zsh, bash, tcsh

[StackOverflow]

Consoles, Terminals, and Shells in Jupyter

- Terminal mirrors the terminal in Linux terminals, Terminal.app (macOS), and PowerShell (Windows)
  - Runs more than just python
- Console provides IPython interface
  - Easier multi-line editing
  - Reference past outputs directly, other bells and whistles
- Shell will run in the Terminal app
- Can also use shell commands in the notebook using `!`
  - `!cat <filename>`
  - `!head -n 10 <filename>`

Python and CLIs

- Python can be used as a CLI program
  - Interactive mode: start the REPL
    - `$ python`
  - Non-interactive mode:
    - `$ python -c <command>`: Execute a command
    - `$ python -m <module>|<package>`: Execute a module
- Python can be used to create CLI programs
  - Scripts: `python my_script.py`
  - True command-line tools: `./command-written-in-python`

Interactive Python in the Shell

- Starting Python from the shell
  - `$ python`
- `>>>` is the Python interactive prompt
  -
```
>>> print("Hello, world")
Hello, world
```
  -
```
>>> print("2+3=", 2+3)
2+3= 5
```
- This is a REPL (Read, Evaluate, Print, Loop)

Interactive Python in the Shell

- `...` is the continuation prompt
-
```
>>> for i in range(5):
...     print(i)
...
```
- Still need to indent appropriately!
- Empty line indicates the suite (block) is finished
- This isn't always the easiest environment to edit in

Ending an Interactive Session

- `Ctrl-D` ends the input stream
  - Just as in other Unix programs
- Another way to get normal termination
  - `>>> quit()`
- `Ctrl-C` interrupts operation
  - Just as in in other Unix programs

Interactive Problems

- But standard interactive Python doesn't save programs!
- IPython does have some magic commands to help
  - `%history`: prints code
  - `%save`: saves a file with code
  - These are most useful outside the notebook, but you can type them in the notebook, too
- However, it is nice to be able to edit code in files and run it, too

Module Files

- A **module file** is a text file with the `.py` extension, usually name.py
- Python source on Unix is UTF-8
- Can use any text editor to write or edit…
- …but an editor that understands Python's spacing and indentation helps!
- Contents looks basically the same as what you would write in the cell(s) of a notebook
- There are also ways to write code in multiple files organized as a package, will cover this later

Scripts, Programs, and Libraries

- Often, interpreted ~ scripts and compiled code ~ programs/libraries
  - Python does compile **bytecode** for modules that are imported
- Modifying this usual definition a bit
  - Script: a one-off block of code meant to be run by itself, users **edit the code** if they wish to make changes
  - Program: code meant to be used in different situations, with **parameters** and **flags** to allow users to customize execution without editing the code
  - Library: code meant to be called from other scripts/programs
- In Python, can't always tell from the name what's expected, code can be both a library and a program

Program Execution

- Direct Unix execution of a program
  - Add the hashbang (`#!`) line as the **first line**, two approaches
  - `#!/usr/bin/python`
  - `#!/usr/bin/env python`
  - Sometimes specify `python3` to make sure we're running Python 3
  - File must be flagged as executable (chmod a+x) and have line endings
  - Then you can say: `$ ./filename.py arg1 ...`
- Executing the Python compiler/interpreter
  - `$ python filename.py arg1 ...`
- Same results either way

Writing CLI Programs

- Command Line Interface Guidelines
  - Accept flags/arguments
  - Human-readable output
  - Allow non-interactive use even if program can also be interactive
  - Add help/usage statements
  - Consider subcommand use for complex tools
  - Use simple, memorable name
  - …

Accepting Command-Line Parameters

- Parameters are received as a list of strings entitled `sys.argv`
- Need to `import sys` first
- `sys.argv[0]` is the name of the program as executed
  - Executing as `./hw01.py` or `hw01.py` will be passed as different strings
- `sys.argv[n]` is the nth argument
- `sys.executable` is the python executable being run

Using Parameters

- Test `len(sys.argv)` to make sure the correct number of parameters were passed
- Everything in `sys.argv` is a string, often need to cast arguments:
  - `my_value = int(sys.argv[1])`
- Guard against bad inputs
  - Test it before using or deal with errors
  - Use isnumeric or catch the exception
  - Printing help/usage statement on error can help users

The main function

- Convention: create a function named `main()`
- Customary, but not required
  -
```
def main():
    print("Running the main function")
```
- Nothing happens in a script with this definition!

The main function

- Convention: create a function named `main()`
- Customary, but not required

```
- def main():
      print("Running the main function")
```

- Nothing happens in a script with this definition!
- Need to call the function in our script!
-
```
def main():
    print("Running the main function")
main() # now, we're calling main
```

Using code as a module, too

- When we want to start a program once it's loaded, we include the line `main()` at the bottom of the code.
- Since Python evaluates the lines of the program during the import process, our current programs also run when they are imported into an interactive Python session or into another Python program.
- `import my_code # prints "Running the main function"`
- Generally, when we import a module, we **don't want it to execute**.

Knowing when the file is being used as a script

- Whenever a module is imported, Python creates a special variable in the module called `__name__` whose value is the name of the imported module.
- Example:

```
>>> import math
>>> math.__name__
'math'
```

- When Python code is run directly and not imported, the value of `__name__` is `'__main__'`.
- We can change the final lines of our programs to:

```
if __name__ == '__main__':
    main()
```