Programming Principles in Python (CSCI 503/490)

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**

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

- \texttt{s0 = "No full dates here, just 02/15"}
  \texttt{s1 = "02/14/2021 is a date"}
  \texttt{s2 = "Another date is 12/25/2020"}
  \texttt{s3 = "April Fools' Day is 4/1/2021 & May the Fourth is 5/4/2021"}

- \texttt{re.match(r'\d+/\d+/\d+',s1)} # returns match object
- \texttt{re.match(r'\d+/\d+/\d+',s2)} # None!
- \texttt{re.search(r'\d+/\d+/\d+',s2)} # returns 1 match object
- \texttt{re.search(r'\d+/\d+/\d+',s3)} # returns 1! match object
- \texttt{re.findall(r'\d+/\d+/\d+',s3)} # returns list of strings
- \texttt{re.finditer(r'\d+/\d+/\d+',s3)} # returns iterable of matches
- \texttt{re.sub(r'(?P<month>\d+)/(?P<day>\d+)/(?P<year>\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>`
Mask Policy

• Masks not required in this class
• Respect all
• Office hours or other interactions: you may ask me to wear a mask
Assignment 4

- Books in Different Languages
- Reading & Writing Files
- Iterators
- Statistics
- String Formatting
- CSCI 503 students compute and output two additional fields
Reading Files

- Use the `open()` method to open a file for reading

  ```python
  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):

  ```python
  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()`
Remember Encoding?

- Unicode, ASCII and others
- `all_lines = open('huck-finn.txt').readlines()`
  `all_lines[0] # '\ufeef\n'`
- `\ufeef` is the UTF Byte-Order-Mark (BOM)
- Optional for UTF-8, but if added, need to read it
- `a = open('huck-finn.txt', encoding='utf-8-sig').readlines()`
  `a[0] # '\n'`
- No need to specify UTF-8 (or ascii since it is a subset)
- Other possible encodings:
  - cp1252, utf-16, iso-8859-1
Other Methods for Reading Files

- **read()**: read the entire file
- **read(<num>)**: read <num> characters (bytes)
  - `open('huck-finn.txt', encoding='utf-8-sig').read(100)`
- **readlines()**: read the entire file as a list of lines
  - `lines = open('huck-finn.txt', encoding='utf-8-sig').readlines()`
Reading a Text File

- Try to read a file at most **once**
- \( f = \text{open('huck-finn.txt', 'r')} \)
  
  ```python
  for i, line in enumerate(f):
      if 'Huckleberry' in line:
          print(line.strip())
  for i, line in enumerate(f):
      if "George" in line:
          print(line.strip())
  ```

- Can't iterate twice!
- Best: do both checks when reading the file once
- Otherwise: either reopen the file or seek to beginning \( f.\text{seek(0)} \)
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
Comma-separated values (CSV) Format

• Comma is a field separator, newlines denote records
  - a,b,c,d,message
    1,2,3,4,hello
    5,6,7,8,world
    9,10,11,12,foo

• May have a header (a,b,c,d,message), but not required

• No type information: we do not know what the columns are (numbers, strings, floating point, etc.)
  - Default: just keep everything as a string
  - Type inference: Figure out the type to make each column based on values

• What about commas in a value? ➔ double quotes
Python csv module

- Help reading csv files using the csv module
  - import csv
    with open('persons_of_concern.csv', 'r') as f:
      for i in range(3): # skip first three lines
        next(f)
    reader = csv.reader(f)
    records = [r for r in reader] # r is a list

- or
  - import csv
    with open('persons_of_concern.csv', 'r') as f:
      for i in range(3): # skip first three lines
        next(f)
    reader = csv.DictReader(f)
    records = [r for r in reader] # r is a dict
Writing Files

- `outf = open("mydata.txt", "w")`
- If you open an existing file for writing, you wipe out the file’s contents. If the named file does not exist, a new one is created.
- Methods for writing to a file:
  - `print(<expressions>, file=outf)`
  - `outf.write(<string>)`
  - `outf.writelines(<list of strings>)`
- If you use write, no newlines are added automatically
  - Also, remember we can change print's ending: `print(..., end="", "`)`
- Make sure you close the file! Otherwise, content may be lost (buffering)
  - `outf.close()`
With Statement: Improved File Handling

• With statement does "enter" and "exit" handling:

• In the previous example, we need to remember to call `outf.close()`

• Using a with statement, this is done automatically:

  ```python
  with open('huck-finn.txt', 'r') as f:
    for line in f:
      if 'Huckleberry' in line:
        print(line.strip())
  ```

• This is important for **writing** files!

  ```python
  with open('output.txt', 'w') as f:
    for k, v in counts.items():
      f.write(k + ': ' + v + '
')
  ```

• Without *with*, we need `f.close()`
Context Manager

• The with statement is used with contexts
• A context manager's **enter** method is called at the beginning
• …and **exit** method at the end, even if there is an exception!

```python
• outf = open('huck-finn-lines.txt','w')
  for i, line in enumerate(huckleberry):
    outf.write(line)
    if i > 3:
      raise Exception("Failure")

• with open('huck-finn-lines.txt','w') as outf:
  for i, line in enumerate(huckleberry):
    outf.write(line)
    if i > 3:
      raise Exception("Failure")
```
Context Manager

• The with statement is used with contexts
• A context manager's **enter** method is called at the beginning
• …and **exit** method at the end, even if there is an exception!
• `outf = open('huck-finn-lines.txt','w')`
  
  ```python
  for i, line in enumerate(huckleberry):
    outf.write(line)
    if i > 3:
      raise Exception("Failure")
  ```

• `with open('huck-finn-lines.txt','w') as outf:`
  
  ```python
  for i, line in enumerate(huckleberry):
    outf.write(line)
    if i > 3:
      raise Exception("Failure")
  ```
JavaScript Object Notation (JSON)

- A format for web data
- Looks very similar to python dictionaries and lists
- Example:
  ```
  {"name": "Wes",
   "places_lived": ["United States", "Spain", "Germany"],
   "pet": null,
   "siblings": [{"name": "Scott", "age": 25, "pet": "Zuko"},
               {"name": "Katie", "age": 33, "pet": "Cisco"}]
  }
  ```
- Only contains literals (no variables) but allows null
- Values: strings, arrays, dictionaries, numbers, booleans, or null
  - Dictionary keys must be strings
  - Quotation marks help differentiate string or numeric values
Reading JSON data

- Python has a built-in `json` module
  - with open('example.json') as f:
    data = json.load(f)
  - with open('example-out.json', 'w') as f:
    json.dump(data, f)

- Can also load/dump to strings:
  - `json.loads`, `json.dumps`
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] ... <command> [<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
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>