Programming Principles in Python (CSCI 503/490)

Files

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)
Unicode and ASCII

• Conceptual systems
• ASCII:
  - old, English-centric, 7-bit system (only 128 characters)
• Unicode:
  - Can represent over 1 million characters from all languages + emoji 🎉
  - Characters have hexadecimal representation: é = U+00E9 and name (LATIN SMALL LETTER E WITH ACUTE)
  - Python allows you to type "é" or represent via code "\u00e9"
• Codes: \texttt{ord} → character to integer, \texttt{chr} → integer to character
Strings

- Objects with methods
- Finding and counting substrings: count, find, startswith
- Removing leading & trailing substrings/whitespace: strip, removeprefix
- Transforming Text: replace, upper, lower, title
- Checking String Composition: isalnum, isnumeric, isupper
- Splitting & Joining:
  - names = str.split(', ')  
  - ', '.join(names)
Format and f-Strings

• `s.format`: templating function
  - Replace fields indicated by curly braces with corresponding values
    - "My name is {} {}".format(first_name, last_name)
    - "My name is {first_name} {last_name}".format(
      first_name=name[0], last_name=name[1])

• Formatted string literals (f-strings) reference variables **directly**!
  - f"My name is {first_name} {last_name}"

• Can include expressions, too:
  - f"My name is {name[0].capitalize()} {name[1].capitalize()}"

• **Format mini-language** allows specialized displays (alignment, numeric formatting)
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+
• Usually use raw strings (no backslash plague): r'\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

- 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
Grouping

- Parentheses capture a group that can be accessed or used later
- Access via `groups()` or `group(n)` where `n` is the number of the group, but numbering starts at 1
- Note: `group(0)` is the full matched string

```python
for match in re.finditer(r'\d+/(\d+)/\d+', s3):
    print(match.groups())
```

```python
for match in re.finditer(r'\d+/(\d+)/\d+', s3):
    print('{2}-{0:02d}-{1:02d}'.format(*[int(x) for x in match.groups()]))
```

- `*` operator expands a list into individual elements
# Modifying Strings

<table>
<thead>
<tr>
<th>Method/Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>split()</code></td>
<td>Split the string into a list, splitting it wherever the RE matches</td>
</tr>
<tr>
<td><code>sub()</code></td>
<td>Find all substrings where the RE matches, and replace them with a different string</td>
</tr>
<tr>
<td><code>subn()</code></td>
<td>Does the same thing as <code>sub()</code>, but returns the new string and the number of replacements</td>
</tr>
</tbody>
</table>
Substitution

- Do substitution in the middle of a string:
  - `re.sub(r'\d+/\d+/\d+', r'3-1-2', s3)`
- All matches are substituted
- First argument is the regular expression to **match**
- Second argument is the **substitution**
  - \1, \2, ... match up to the **captured groups** in the first argument
- Third argument is the **string** to perform substitution on
- Can also use a **function**:
  - `to_date = lambda m: f'{m.group(3)}-{int(m.group(1)):02d}-{int(m.group(2)):02d}'`
  - `re.sub(r'\d+/\d+/\d+', to_date, s3)`
Assignment 4

- Assignment will cover strings and files
- Reading & writing data to files
- Dealing with characters and encodings
Files
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.
Opening a File

• Opening associates a file on disk with an object in memory (file object or file handle).

• We access the file via the file object.

• `<filevar> = open(<name>, <mode>)`

• Mode `r` = read or `w` = write, `a` = append

• read is default

• Also add `b` to indicate the file should be opened in binary mode: `rb`, `wb`
Standard File Objects

• When Python begins, it associates three standard file objects:
  - `sys.stdin`: for input
  - `sys.stdout`: for output
  - `sys.stderr`: for errors

• In the notebook
  - `sys.stdin` isn't really used, `get_input` can be used if necessary
  - `sys.stdout` is the output shown after the code
  - `sys.stderr` is shown with a red background
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')`
    
    ```python
    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 Encodings (Unicode, ASCII)?

- Encoding: How things are actually stored
- ASCII "Extensions": how to represent characters for different languages
  - No universal extension for 256 characters (one byte), so…
  - ISO-8859-1, ISO-8859-2, CP-1252, etc.
- Unicode encoding:
  - UTF-8: used in Python and elsewhere (uses variable # of 1—4 bytes)
  - Also UTF-16 (2 or 4 bytes) and UTF-32 (4 bytes for everything)
  - Byte Order Mark (BOM) for files to indicate endianness (which byte first)
Encoding in Files

- `all_lines = open('huck-finn.txt').readlines()`
  `all_lines[0]` # `ﻯ
`

- `ﻯ` 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]` # `
`

- 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**

```python
f = open('huck-finn.txt', 'r')
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.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? \(\rightarrow\) 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!
• 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")
  ```