Programming Principles in Python (CSCI 503)

Files

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)
Functional Programming

• Programming without imperative statements like assignment
• In addition to comprehensions & iterators, have functions:
  - map: iterable of n values to an iterable of n transformed values
  - filter: iterable of n values to an iterable of m (m <= n) values
• Eliminates need for concrete looping constructs
Lambda Functions

• `def is_even(x):
    return (x % 2) == 0`

• `filter(is_even, range(10))` # generator

• Lots of code to write a simple check

• Lambda functions allow inline function definition

• Usually used for "one-liners": a simple data transform/expression

• `filter(lambda x: x % 2 == 0, range(10))`

• Parameters follow `lambda`, no parentheses

• No `return` keyword as this is implicit in the syntax

• JavaScript has similar functionality (arrow functions): `(d => d % 2 == 0)`
Strings

• Remember strings are sequences of characters
• Strings are collections so have `len`, `in`, and iteration
  
  - `s = "Huskies"
    - `len(s)`, "usk" in s, `[c for c in s if c == 's']`

• Strings are sequences so have
  - indexing and slicing: `s[0]`, `s[1:]`
  - concatenation and repetition: `s + " at NIU"; s * 2`
• Single or double quotes `'string1', "string2`'
• Triple double-quotes: `"""A string over many lines"""`
• Escaped characters: `\n` (newline) `\t` (tab)
Unicode and ASCII

• Conceptual systems
• ASCII:
  - old 7-bit system (only 128 characters)
  - English-centric
• Unicode:
  - modern system
  - 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"
String Methods

• We can call methods on strings like we can with lists
  
  ```python
  s = "Peter Piper picked a peck of pickled peppers"
  s.count('p')
  ```

• Categories of Methods
  
  - Finding and counting substrings
  - Removing leading and trailing whitespace and strings
  - Transforming text
  - Checking string composition
  - Splitting and joining strings
  - Formatting
Formatting

- `s.ljust, s.rjust, s.zfill`: justification/filling
- `s.format`: templating function
  - Replace fields indicated by curly braces with corresponding values
    - "My name is {} {}".format(first_name, last_name)
    - "My name is {1} {0}".format(last_name, first_name)
    - "My name is {first_name} {last_name}".format(first_name=name[0], last_name=name[1])
  - Braces can contain number or name of keyword argument
  - Whole `format mini-language` to control formatting
- `f-strings`: f"My name is {first_name} {last_name}"
Raw Strings

- Raw strings prefix the starting delimiter with \texttt{r}
- Disallow escaped characters
  - '\textbackslash\textbackslash\textbackslash\textbackslash is the way you write a newline, \textbackslash\textbackslash\textbackslash\textbackslash for \textbackslash\textbackslash\textbackslash\textbackslash.'
  - \texttt{r"\textbackslash\textbackslash\textbackslash\textbackslash is the way you write a newline, \textbackslash\textbackslash for \textbackslash\textbackslash\textbackslash\textbackslash."}
- Useful for regular expressions
Assignment 4

• Illinois Climate Data
• Reading & Writing Files
• Iterators
• Numeric Aggregation (think about comprehensions)
• Formatting Strings
Test 1

• Wednesday from 2:00-3:15pm on Blackboard
• Covers material through last Wednesday's class
• Content aligns with recommended text, but we covered more in lectures
• Format:
  - Multiple Choice
  - Free Response (see web page for examples)
• Questions related to principles and concepts as well as Python specifically (i.e. syntax)
Regular Expressions

- AKA regex
- A syntax to better specify how to decompose strings
- Look for patterns rather than specific characters
- "31" in "The last day of December is 12/31/2016."
- May work for some questions but now suppose I have other lines like: "The last day of September is 9/30/2016."
- …and I want to find dates that look like:
  - \{digits\}/\{digits\}/\{digits\}
- Cannot search for every combination!
  - \d+//\d+//\d+ # \d is a character class
Regular Expressions

• AKA regex
• A syntax to better specify how to decompose strings
• Look for patterns rather than specific characters
• "31" in "The last day of December is 12/31/2016."
• May work for some questions but now suppose I have other lines like: "The last day of September is 9/30/2016."
• …and I want to find dates that look like:
  • {digits}/{digits}/{digits}
• Cannot search for every combination!
  • \d+/%d+/%d+  # \d is a character class
Metacharacters

• Need to have some syntax to indicate things like repeat or one-of-these or this is optional.
  • .  ^  $  *  +  ?  { }  [ ]  \  |  ( )
  • [ ]: define character class
  • ^: complement (opposite)
  • \: escape, but now escapes metacharacters and references classes
  • *: repeat zero or more times
  • +: repeat one or more times
  • ?: zero or one time
  • {m, n}: at least m and at most n
### Predefined Character Classes

<table>
<thead>
<tr>
<th>Character class</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\d</code></td>
<td>Any digit (0–9).</td>
</tr>
<tr>
<td><code>\D</code></td>
<td>Any character that is <em>not</em> a digit.</td>
</tr>
<tr>
<td><code>\s</code></td>
<td>Any whitespace character (such as spaces, tabs and newlines).</td>
</tr>
<tr>
<td><code>\S</code></td>
<td>Any character that is <em>not</em> a whitespace character.</td>
</tr>
<tr>
<td><code>\w</code></td>
<td>Any <em>word character</em> (also called an <em>alphanumeric character</em>)</td>
</tr>
<tr>
<td><code>\W</code></td>
<td>Any character that is <em>not</em> a word character.</td>
</tr>
</tbody>
</table>
Performing Matches

<table>
<thead>
<tr>
<th>Method/Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>match()</code></td>
<td>Determine if the RE matches at the beginning of the string.</td>
</tr>
<tr>
<td><code>search()</code></td>
<td>Scan through a string, looking for any location where this RE matches.</td>
</tr>
<tr>
<td><code>findall()</code></td>
<td>Find all substrings where the RE matches, and returns them as a list.</td>
</tr>
<tr>
<td><code>finditer()</code></td>
<td>Find all substrings where the RE matches, and returns them as an iterator.</td>
</tr>
</tbody>
</table>
Regular Expressions in Python

- `import re`
- `re.match(<pattern>, <str_to_check>)`
  - Returns `None` if no match, information about the match otherwise
  - Starts at the **beginning** of the string
- `re.search(<pattern>, <str_to_check>)`
  - Finds **single** match **anywhere** in the string
- `re.findall(<pattern>, <str_to_check>)`
  - Finds **all** matches in the string, search only finds the first match
- Can pass in flags to alter methods: e.g. `re.IGNORECASE`
Examples

- s0 = "No full dates here, just 02/15"
  s1 = "02/14/2021 is a date"
  s2 = "Another date is 12/25/2020"
- re.match(r'\d+/>\d+/>\d+',s1) # returns match object
- re.match(r'\d+/>\d+/>\d+',s0) # None
- 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())
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>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 and the number of replacements</td>
</tr>
</tbody>
</table>
Substitution

• Do substitution in the middle of a string:
  \[\text{re.sub}(r' (\d+)/ (\d+)/ (\d+)', r'\3-\1-\2', s3)\]
• 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}'
  \[\text{re.sub}(r' (\d+)/ (\d+)/ (\d+)', to_date, s3)\]
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
  
  ```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] # '\ufeff
'`
- `\ufeff` 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? → 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
JavaScript Object Notation (JSON)

• A format for web data
• Looks very similar to python dictionaries and lists
• Example:

  ```json
  {"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`
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:
  - `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!
  - `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")
```

```python
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')`
  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")