Programming Principles in Python (CSCI 503/490)

Strings

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)
Generators

- Special functions that return **lazy** iterables
- Use less memory
- Change is that functions **yield** instead of **return**
- \[
\text{def square(it):}
\quad \text{for i in it:}
\quad \quad \text{yield i*i}
\]
- If we are iterating through a generator, we hit the first yield and immediately return that first computation
- Generator expressions just shorthand (remember no tuple comprehensions)
  - \((i * i \text{ for } i \text{ in } [1,2,3,4,5])\)
Efficient Evaluation

- Only compute when necessary, not beforehand

```python
u = compute_fast_function(s, t)
v = compute_slow_function(s, t)
if s > t and s**2 + t**2 > 100:
    u = compute_fast_function(s, t)
    res = u / 100
else:
    v = compute_slow_function(s, t)
    res = v / 100
```

- slow function will not be executed unless the condition is true
Short-Circuit Evaluation

- Automatic, works left to right according to order of operations (and before or)
- Works for and and or

and:
- if any value is False, stop and return False
  - a, b = 2, 3
    a > 3 and b < 5

or:
- if any value is True, stop and return True
  - a, b, c = 2, 3, 7
    a > 3 or b < 5 or c > 8
Memoization

- memo_dict = {}
  def memoized_slow_function(s, t):
    if (s, t) not in memo_dict:
      memo_dict[(s, t)] = compute_slow_function(s, t)
    return memo_dict[(s, t)]

- for s, t in [(12, 10), (4, 5), (5, 4), (12, 10)]:
  if s > t and (c := memoized_slow_function(s, t) > 50):
    pass
  else:
    c = compute_fast_function(s, t)

- Second time executing for s=12, t=10, we don't need to compute!
- Tradeoff memory for compute time
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)`
Assignment 3

• Important for Test 1, but studying also should be a priority
• Use dictionaries, lists, and sets to analyze US Senate Stock Trades
• Due Today
Test 1

- This Wednesday, Feb. 21, 12:30pm-1:45pm
- In-Class, paper/pen & pencil
- Covers material through last week
- Format:
  - Multiple Choice
  - Free Response
- Information at the link above
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"
Unicode and 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)
Codes

• Characters are still stored as bits and thus can be represented by numbers
  - \texttt{ord} \rightarrow \text{character to integer}
  - \texttt{chr} \rightarrow \text{integer to character}
  - \\N{horse}: named emoji
Strings are Objects with Methods

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

• Doesn't matter if we have a variable or a literal
  - "Peter Piper picked a peck of pickled peppers".\text{find}("pick")
Finding & Counting Substrings

- `s.count(sub)`: Count the number of occurrences of `sub` in `s`
- `s.find(sub)`: Find the first position where `sub` occurs in `s`, else -1
- `s.rfind(sub)`: Like `find`, but returns the right-most position
- `s.index(sub)`: Like `find`, but raises a `ValueError` if not found
- `s.rindex(sub)`: Like `index`, but returns right-most position
- `sub in s`: Returns `True` if `s` contains `sub`
- `s.startswith(sub)`: Returns `True` if `s` starts with `sub`
- `s.endswith(sub)`: Returns `True` if `s` ends with `sub`
Removing Leading and Trailing Strings

- `s.strip()`: Copy of `s` with leading and trailing whitespace removed
- `s.lstrip()`: Copy of `s` with leading whitespace removed
- `s.rstrip()`: Copy of `s` with trailing whitespace removed
- `s.removeprefix(prefix)`: Copy of `s` with `prefix` removed (if it exists)
- `s.removesuffix(suffix)`: Copy of `s` with `suffix` removed (if it exists)
Transforming Text

• `s.replace(oldsub, newsub)`: Copy of `s` with occurrences of `oldsub` in `s` with `newsub`

• `s.upper()`: Copy of `s` with all uppercase characters

• `s.lower()`: Copy of `s` with all lowercase characters

• `s.capitalize()`: Copy of `s` with first character capitalized

• `s.title()`: Copy of `s` with first character of each word capitalized
## Checking String Composition

<table>
<thead>
<tr>
<th>String Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isalnum()</code></td>
<td>Returns True if the string contains only alphanumeric characters (i.e., digits &amp; letters).</td>
</tr>
<tr>
<td><code>isalpha()</code></td>
<td>Returns True if the string contains only alphabetic characters (i.e., letters).</td>
</tr>
<tr>
<td><code>isdecimal()</code></td>
<td>Returns True if the string contains only decimal integer characters</td>
</tr>
<tr>
<td><code>isdigit()</code></td>
<td>Returns True if the string contains only digits (e.g., '0', '1', '2').</td>
</tr>
<tr>
<td><code>isidentifier()</code></td>
<td>Returns True if the string represents a valid identifier.</td>
</tr>
<tr>
<td><code>islower()</code></td>
<td>Returns True if all alphabetic characters in the string are lowercase characters</td>
</tr>
<tr>
<td><code>isnumeric()</code></td>
<td>Returns True if the characters in the string represent a numeric value w/o a + or - or .</td>
</tr>
<tr>
<td><code>isspace()</code></td>
<td>Returns True if the string contains only whitespace characters.</td>
</tr>
<tr>
<td><code>istitle()</code></td>
<td>Returns True if the first character of each word is the only uppercase character in it.</td>
</tr>
<tr>
<td><code>isupper()</code></td>
<td>Returns True if all alphabetic characters in the string are uppercase characters</td>
</tr>
</tbody>
</table>
Splitting

- \( s = "Venkata, Ranjit, Pankaj, Ali, Karthika" \)
- `names = s.split(',')` # names is a list
- `names = s.split(',', 3)` # split by commas, split <= 3 times
- separator may be multiple characters
- if no separator is supplied (`sep=None`), runs of consecutive whitespace delimit elements
- `rsplit` works in reverse, from the right of the string
- `partition` and `rpartition` for a single split with before, sep, and after
- `splitlines` splits at line boundaries, optional parameter to keep endings
Joining

- `join` is a method on the `separator` used to join a list of strings
- `','.join(names)`
  - `names` is a list of strings, `','` is the separator used to join them

Example:
- `def orbit(n):
  # ...
  return orbit_as_list
  print(','.join(orbit_as_list))`
Formatting

- **s.ljust, s.rjust**: justify strings by adding fill characters to obtain a string with specified width
- **s.zfill**: `ljust` with zeroes
- **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
Format Strings

• Formatted string literals (f-strings) prefix the starting delimiter with \texttt{f}
• Reference variables \texttt{directly!}
  - \texttt{f"My name is \{first\_name\} \{last\_name\}"}
• Can include expressions, too:
  - \texttt{f"My name is \{name[0].capitalize()\} \{name[1].capitalize()\}"}
• Same \texttt{format mini-language} is available
Format Mini-Language Presentation Types

• Not usually required for obvious types
• :d for integers
• :c for characters
• :s for strings
• :e or :f for floating point
  - :e: scientific notation (all but one digit after decimal point)
  - :f: fixed-point notation (decimal number)
Field Widths and Alignments

• After : but before presentation type
  - f'[{27:10d}]' # ' [       27 ]'
  - f'[{"hello":10}]' # ' [hello       ]'

• Shift alignment using < or >:
  - f'[{"hello":>15}]' # ' [          hello ]'

• Center align using ^:
  - f'[{"hello":^7}]' # ' [ hello ]'
Numeric Formatting

- Add positive sign:
  - `f'[{27:+10d}]'` # `[       +27]`

- Add space but only show negative numbers:
  - `print(f'{27: d}\n{-27: d}')` # note the space in front of 27

- Separators:
  - `f'{12345678:,d}'` # '12,345,678'
Raw Strings

• Raw strings prefix the starting delimiter with `r`
•Disallow escaped characters
  • `'\\n is the way you write a newline, ``` `for "`'
  • `r"\\n is the way you write a newline, ` `for \``"`
• Useful for regular expressions
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>\d</td>
<td>Any digit (0–9).</td>
</tr>
<tr>
<td>\D</td>
<td>Any character that is <em>not</em> a digit.</td>
</tr>
<tr>
<td>\s</td>
<td>Any whitespace character (such as spaces, tabs and newlines).</td>
</tr>
<tr>
<td>\S</td>
<td>Any character that is <em>not</em> a whitespace character.</td>
</tr>
<tr>
<td>\w</td>
<td>Any <strong>word character</strong> (also called an <strong>alphanumeric character</strong>).</td>
</tr>
<tr>
<td>\W</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 <em>iterator</em>.</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