Programming Principles in Python (CSCI 503)

Strings

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)
Sets & Operations

- \( s = \{ 'DeKalb', 'Kane', 'Cook', 'Will' \} \)
  \( t = \{ 'DeKalb', 'Winnebago', 'Will' \} \)

- **Union:** \( s \cup t \equiv \{ 'DeKalb', 'Kane', 'Cook', 'Will', 'Winnebago' \} \)

- **Intersection:** \( s \cap t \equiv \{ 'DeKalb', 'Will' \} \)

- **Difference:** \( s \setminus t \equiv \{ 'Kane', 'Cook' \} \)

- **Symmetric Difference:** \( s \Delta t \equiv \{ 'Kane', 'Cook', 'Winnebago' \} \)

- **Object method variants:**
  - \( s.union(t), s.intersection(t), \)
  - \( s.difference(t), s.symmetric_difference(t) \)

- ***_update and augmented operator variants**
Comprehension

• Shortcut for loops that **transform** or **filter** collections
• Functional programming features this way of thinking:
  Pass functions to functions!
• Imperative: a loop with the actual functionality buried inside
• Functional: specify both functionality and data as inputs
List Comprehension

- output = []
  for d in range(5):
    output.append(d ** 2 - 1)

- Rewrite as a map:
  - output = [d ** 2 - 1 for d in range(5)]

- Can also filter:
  - output = [d for d in range(5) if d % 2 == 1]

- Combine map & filter:
  - output = [d ** 2 - 1 for d in range(5) if d % 2 == 1]
Comprehensions for other collections

• Dictionaries
  - \{k: v for (k, v) in other_dict.items() if k.startswith('a')\}
  - Example: one-to-one map inverses
    • \{v: k for (k, v) in other_dict.items()\}
    • Be careful that the dictionary is actually one-to-one!

• Sets:
  - \{s[0] for s in names\}

• Tuples? Not exactly
  - \(s[0] for s in names\)
  - Not a tuple, a generator expression
Iterators

• Key concept: iterators only need to have a way to get the next element
• To be iterable, an object must be able to produce an iterator
  - Technically, must implement the `__iter__` method
• An iterator must have two things:
  - a method to get the next item
  - a way to signal no more elements
• In Python, an iterator is an object that must
  - have a defined `__next__` method
  - raise `StopException` if no more elements available
Generators

• Special functions that return lazy iterables
• Use less memory
• Change is that functions yield instead of return
  
  ```python
  def square(it):
      for i in it:
          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 for i 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
Assignment 3

• Due Thursday
• US Senate Stock Trading Data
• Lots of iteration and dictionary access
• Also create new lists and dictionaries
• Last Part is CSCI 503 Only
Test 1

• Covers material through today'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)
Example

• Suppose I want to write Python code to print the numbers from 1 to 100. What errors do you see? How could you improve the code?

```python
// print the numbers from 1 to 100
int counter = 1
while counter < 100 {
    print counter
    counter++
}
```
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
Map

- Generator function (lazy evaluation)
- First argument is a **function**, second argument is the **iterable**
  ```python
def upper(s):
    return s.upper()
```
- `map(upper, ['sentence', 'fragment'])` # generator
- Similar comprehension:
  ```python
  - [upper(s) for s in ['sentence', 'fragment']] # comprehension
  ```
- This only calls `upper` once
  ```python
  for word in map(upper, ['sentence', 'fragment']):
    if word == "SENTENCE":
      break
  ```
Filter

• Also a generator
• def is_even(x):
    return (x % 2) == 0
• filter(is_even, range(10)) # generator

• Similar comprehension:
  - [d for d in range(10) if is_even(d)] # comprehension
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
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)
Strings are Objects with Methods

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

• Doesn't matter if we have a variable or a literal
  - "Peter Piper picked a peck of pickled peppers".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>isalnum()</td>
<td>Returns True if the string contains only alphanumeric characters (i.e., digits &amp; letters).</td>
</tr>
<tr>
<td>isalpha()</td>
<td>Returns True if the string contains only alphabetic characters (i.e., letters).</td>
</tr>
<tr>
<td>isdecimal()</td>
<td>Returns True if the string contains only decimal integer characters</td>
</tr>
<tr>
<td>isdigit()</td>
<td>Returns True if the string contains only digits (e.g., '0', '1', '2').</td>
</tr>
<tr>
<td>isidentifier()</td>
<td>Returns True if the string represents a valid identifier.</td>
</tr>
<tr>
<td>islower()</td>
<td>Returns True if all alphabetic characters in the string are lowercase characters</td>
</tr>
<tr>
<td>isnumeric()</td>
<td>Returns True if the characters in the string represent a numeric value w/o a + or - or .</td>
</tr>
<tr>
<td>isspace()</td>
<td>Returns True if the string contains only whitespace characters.</td>
</tr>
<tr>
<td>istitle()</td>
<td>Returns True if the first character of each word is the only uppercase character in it.</td>
</tr>
<tr>
<td>isupper()</td>
<td>Returns True if all alphabetic characters in the string are uppercase characters</td>
</tr>
</tbody>
</table>
Splitting

- \( s = "Venkata, Ranjit, Pankaj, Ali, Karthika" \)
- \( \text{names} = s\text{.split(',')} \) # names is a list
- \( \text{names} = s\text{.split(',}', 3)} \) # split by commas, split <= 3 times
- separator may be multiple characters
- if no separator is supplied (\( \text{sep=None} \)), runs of consecutive whitespace delimit elements
- \( \text{rsplit} \) works in reverse, from the right of the string
- \( \text{partition} \) and \( \text{rpartition} \) for a single split with before, sep, and after
- \( \text{splitlines} \) 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 numbers(n):
    # ...
    return numbers_as_list
print(','.join(numbers_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 f

• 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()}"

• Same **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:
  - \texttt{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