Programming Principles in Python (CSCI 503/490)

Sets, Comprehensions, and Generators

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)
Dictionary

• AKA associative array or map
• Collection of key-value pairs
  - Keys must be unique
  - Values need not be unique
• Syntax:
  - Curly brackets {} delineate start and end
  - Colons separate keys from values, commas separate pairs
  - d = {'DeKalb': 783, 'Kane': 134, 'Cook': 1274, 'Will': 546}
• No type constraints
  - d = {'abc': 25, 12: 'abc', ('Kane', 'IL'): 123.54}
Collections

- A dictionary is **not** a sequence
- Sequences are **ordered**
- Conceptually, dictionaries need no order
- A dictionary is a **collection**
- Sequences are also collections
- All collections have length (**len**), membership (**in**), and iteration (loop over values)
- Length for dictionaries counts number of key-value **pairs**
  - Pass dictionary to the **len** function
  - `d = {'abc': 25, 12: 'abc', ('Kane', 'IL'): 123.54}
    
    `len(d)` # 3
Mutability

- Dictionaries are **mutable**, key-value pairs can be added, removed, updated
- ```
   d = {'DeKalb': 783, 'Kane': 134, 'Cook': 1274, 'Will': 546}
   d['Winnebago'] = 1023 # add a new key-value pair
   d['Kane'] = 342       # update an existing key-value pair
   d.pop('Will')        # remove an existing key-value pair
   del d['Winnebago']   # remove an existing key-value pair
   d.update({'Winnebago': 1023, 'Kane': 324})
   d.update([('Winnebago', 1023), ('Kane', 324)])
   d.update(Winnebago=1023, Kane=324)
```
# Dictionary Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;dict&gt;.clear()</code></td>
<td>Remove all key-value pairs</td>
</tr>
<tr>
<td><code>&lt;dict&gt;.update(other)</code></td>
<td>Updates the dictionary with values from <code>other</code></td>
</tr>
<tr>
<td><code>&lt;dict&gt;.pop(k, d=None)</code></td>
<td>Removes the pair with key <code>k</code> and returns value or default <code>d</code> if no key</td>
</tr>
<tr>
<td><code>&lt;dict&gt;.get(k, d=None)</code></td>
<td>Returns the value for the key <code>k</code> or default <code>d</code> if no key</td>
</tr>
<tr>
<td><code>&lt;dict&gt;.items()</code></td>
<td>Returns iterable view over all pairs as (key, value) tuples</td>
</tr>
<tr>
<td><code>&lt;dict&gt;.keys()</code></td>
<td>Returns iterable view over all keys</td>
</tr>
<tr>
<td><code>&lt;dict&gt;.values()</code></td>
<td>Returns iterable view over all values</td>
</tr>
</tbody>
</table>
## Dictionary Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Meaning</th>
<th>Mutate</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;dict&gt;.clear()</code></td>
<td>Remove all key-value pairs</td>
<td></td>
</tr>
<tr>
<td><code>&lt;dict&gt;.update(other)</code></td>
<td>Updates the dictionary with values from <code>other</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;dict&gt;.pop(k, d=None)</code></td>
<td>Removes the pair with key <code>k</code> and returns value or default <code>d</code> if no key</td>
<td></td>
</tr>
<tr>
<td><code>&lt;dict&gt;.get(k, d=None)</code></td>
<td>Returns the value for the key <code>k</code> or default <code>d</code> if no key</td>
<td></td>
</tr>
<tr>
<td><code>&lt;dict&gt;.items()</code></td>
<td>Returns iterable view over all pairs as <code>(key, value)</code> tuples</td>
<td></td>
</tr>
<tr>
<td><code>&lt;dict&gt;.keys()</code></td>
<td>Returns iterable view over all keys</td>
<td></td>
</tr>
<tr>
<td><code>&lt;dict&gt;.values()</code></td>
<td>Returns iterable view over all values</td>
<td></td>
</tr>
</tbody>
</table>
Iteration

- Even though dictionaries are not sequences, we can still iterate through them.
- Principle: Don't depend on order.
- `for k in d:
    print(k, end=" ")`
  - This only iterates through the **keys**!
- `for k in d:
    print('key:', k, 'value:', d[k], end=" ")`
  - ...but this is kind of like counting through a sequence (not pythonic)
Iteration

• for k in d:  # iterate through keys
    print(k, end=" ")
• for k in d.keys():  # iterate through keys
    print('key:', k)
• for v in d.values():  # iterate through values
    print('value:', v)
• for k, v in d.items():  # iterate through key-value pairs
    print('key:', k, 'value:', v)
Dictionary Views

• for k in d.keys(): # iterate through keys
  print('key:', k)
• for v in d.values(): # iterate through values
  print('value:', v)
• for k, v in d.items(): # iterate through key-value pairs
  print('key:', k, 'value:', v)
• keys() is superfluous but is a bit clearer
• items() is the enumerate-like method
Exercise: Count Letters

- Write code to take a string and return the count of each letter that occurs in a dictionary
- `count_letters('illinois')`
  
  # returns {'i': 3, 'l': 2, 'n': 1, 'o': 1, 's': 1}
Exercise: Count Letters

- def count_letters(s):
  d = {}
  for c in s:
    if c not in d:
      d[c] = 1
    else:
      d[c] += 1
  return d

count_letters('illinois')
Exercise: Count Letters

• def count_letters(s):
    d = {}
    for c in s:
        d[c] = d.get(c, 0) + 1
    return d

count_letters('illinois')
Exercise: Count Letters (using collections)
Exercise: Count Letters (using collections)

- from collections import defaultdict
def count_letters(s):
    d = defaultdict(int)
    for c in s:
        d[c] += 1
    return d
count_letters('illinois')
Exercise: Count Letters (using collections)

- from collections import defaultdict
  def count_letters(s):
    d = defaultdict(int)
    for c in s:
      d[c] += 1
    return d
  count_letters('illinois')

- from collections import Counter
  def count_letters(s):
    return Counter(s)
  count_letters('illinois')
Sorting

- Order doesn't really mean anything in a dictionary
- There is no .sort or .reverse method
- We can iterate through items in sorted order using sorted
- `d = count_letters('illinois')`
  ```python
  for k, v in sorted(d.items()):
      print(k, ':', v)
  ```
- reversed also works on dictionary views
- sorted and reversed work on any iterable (thus all collections)
Assignment 3

- USDA Food Data
- Looking at branded data and nutrition information
- Start with the sample notebook (or copy its code) to download the data
- Data is a list of dictionaries
- Need to iterate through, update, and create new lists & dictionaries
- Part 6 is CSCI 503 students Only, but CSCI 490 students may complete for extra credit
Test 1

• Next Wednesday, Feb. 23
• In-class, 2:00-3:15pm in PM 153
• Format:
  - Multiple Choice
  - Free Response
• More information to be posted online soon
Sets
Sets

• Sets are dictionaries but without the values
• Same curly braces, no pairs
• \( s = \{ 'DeKalb', 'Kane', 'Cook', 'Will' \} \)
• Only one instance of a value is in a set—sets **eliminate duplicates**
• Adding multiple instances of the same value to a set doesn't do anything
  • \( s = \{ 'DeKalb', 'DeKalb', 'DeKalb', 'Kane', 'Cook', 'Will' \} \)
  • \# \{ 'Cook', 'DeKalb', 'Kane', 'Will' \}
• Watch out for the empty set
  - \( s = \{ \} \) \# not a set!
  - \( s = \text{set()} \) \# an empty set
Sets are Mutable Collections

- Sets are `mutable` like dictionaries: we can add, and delete
- Again, no type constraints
  - `s = {12, 'DeKalb', 22.34}`
- Like a dictionary, a set is a `collection` but not a sequence
- Q: What three things can we do for any collection?
Collection Operations on Sets

- $s = \{ 'DeKalb', 'Kane', 'Cook', 'Will' \}$
- Length
  - $\text{len}(s)$ # 4
- Membership: fast just like dictionaries
  - 'Kane' in s # True
  - 'Winnebago' not in s # True
- Iteration
  - for county in s:
    - print(county)
Mathematical Set Operations

- $s = \{\text{DeKalb}, \text{Kane}, \text{Cook}, \text{Will}\}$
  $t = \{\text{DeKalb}, \text{Winnebago}, \text{Will}\}$

- Union: $s \cup t \ # \ \{\text{DeKalb}, \text{Kane}, \text{Cook}, \text{Will}, \text{Winnebago}\}$
  - Unlike dictionaries, is commutative for sets ($s \cup t == t \cup s$)

- Intersection: $s \cap t \ # \ \{\text{DeKalb}, \text{Will}\}$

- Difference: $s - t \ # \ \{\text{Kane}, \text{Cook}\}$

- Symmetric Difference: $s \Delta t \ # \ \{\text{Kane}, \text{Cook}, \text{Winnebago}\}$

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

- Disjoint: $s$.isdisjoint(t) \ # \ False
Mutation Operations

- add: `s.add('Winnebago')`
- discard: `s.discard('Will')`
- remove: `s.remove('Will')` # generates KeyError if not exist
- clear: `s.clear()` # removes all elements

- Variants of the mathematical set operations (have augmented assignments)
  - `update (union): |=`
  - `intersection_update: &=`
  - `difference_update: -=`
  - `symmetric_difference_update: ^=`

- Methods take any iterable, operators require sets
Comprehensions
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 using other collections

- Comprehensions can use existing collections, too (not just ranges)
- Anything that is **iterable** can be used in the for construct (like for loop)
- `names = ['smith', 'Smith', 'John', 'mary', 'jan']`
- `names2 = [item.upper() for item in names]`
Any expression works as output items

• Tuples inside of comprehension
  - `[(s, s+2) for s in slist]`

• Dictionaries, too
  - `[{"i": i, "j": j} for (i, j) in tuple_list]`

• Function calls
  - `names = ['smith', 'Smith', 'John', 'mary', 'jan']`
    `names2 = [item.upper() for item in names]`
Multi-Level and Nested Comprehensions

• **Flattening** a list of lists
  - my_list = [[1,2,3],[4,5],[6,7,8,9,10]]
    - [v for vlist in my_list for v in vlist]
  - [1,2,3,4,5,6,7,8,9,10]

• Note that the for loops are in order

• Difference between **nested** comprehensions
  - [[v**2 for v in vlist] for vlist in my_list]
  - [[1,4,9],[16,25],[36,49,64,81,100]]
Comprehensions for other collections

• Dictionaries
  - {k: v for (k, v) in other_dict.items()
      if k.startswith('a')}
  - Sometimes used for one-to-one map inverses
• How?
Comprehensions for other collections

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

• Sets:
  - \( \{ s[0] \text{ for } s \text{ in names} \} \)
Tuple Comprehension?

- thing = (x ** 2 for x in numbers if x % 2 != 0)
  thing # not a tuple! <generator object <genexpr> ...>

- Actually a **generator**!

- This **delays** execution until we actually need each result