CS 134: Dictionaries and Sets
Announcements & Logistics

- **Lab 5** is today/tomorrow
  - Expect most people to finish it during scheduled lab period
- **Midterm**: Thu Mar 17th
  - Attend one slot: 6 - 7:30pm or 8 - 9:30pm in **Wachenheim B11**
  - **Wachenheim 002** at 6pm for reduced distractions/extra time
- **Midterm review**: Tue Mar 15th
  - 7 - 8:30 pm in **TPL 203** (bring your questions!)
- **Practice midterm** on Glow
- Please fill out the **CS134 TA feedback form** by Friday

Do You Have Any Questions?
Last Time

• A **dictionary** is a **mutable** collection that maps **keys** to **values**
  • **Keys** must be unique & **immutable**, **values** can any Python object
• Iterating over a dictionary: what do we iterate over?
  • Iterate over the **keys** of a dictionary directly (by default)
• Dictionary comprehensions: similar to list comprehensions
• Useful dictionary method:
  • `dict.get(key, defaultVal)`: returns `dict[key]` if key exists, else returns `defaultVal`. If no `defaultVal` provided: returns `None` if key does not exist.
Today’s Plan

• Wrap up dictionaries
• Investigate **sorting** with dictionaries
• Discuss a new unordered data structure: **sets**
• Review all data structures so far and when to use each
Recap: Dictionaries and Mutability

- Dictionaries are **mutable**
  - Has implications for aliasing!
    ```python
    >>> myDict = {1: 'a', 2: 'b', 3: 'c'}
    >>> newDict = myDict  # alias!
    >>> newDict[4] = 'd'
    >>> myDict  # changes as well
    {1: 'a', 2: 'b', 3: 'c', 4: 'd'}
    ```
  - Note: dictionary keys **must be immutable**
    - Cannot have keys of mutable types such as list
  - Dictionary values can be any type (mutable values such as lists)
Recap: Dictionary Comprehensions

- Similar to list comprehensions, useful for mapping and filtering
- Remember: when iterating over a dictionary, we are iterating over its **keys** (in the order of creation)

```python

days30 = {k: calendar[k] for k in calendar if calendar[k] == 30}

days30

{'Apr': 30, 'Jun': 30, 'Sep': 30, 'Nov': 30}
```
Sorting Operations with Dictionaries

• Let’s say we’re developing a Scrabble app

• We can store the score for each letter as a dictionary as below

```python
scrabbleScore = {'a':1, 'b':3, 'c':3, 'd':2, 'e':1, 'f':4, 'g':2, 'h':4, 'i':1, 'j':8, 'k':5, 'l':1, 'm':3, 'n':1, 'o':1, 'p':3, 'q':10, 'r':1, 's':1, 't':1, 'u':1, 'v':8, 'w':4, 'x':8, 'y':4, 'z':10}
```

• If we call the `sorted()` function on a dictionary, it returns an ordered list of all the keys.
Sorting Operations with Dictionaries

• Let’s say we’re developing a Scrabble app

• We can store the score for each letter as a dictionary as below

```python
scrabbleScore = {'a': 1, 'b': 3, 'c': 3, 'd': 2, 'e': 1, 'f': 4, 'g': 2, 'h': 4, 'i': 1, 'j': 8, 'k': 5, 'l': 1, 'm': 3, 'n': 1, 'o': 1, 'p': 3, 'q': 10, 'r': 1, 's': 1, 't': 1, 'u': 1, 'v': 8, 'w': 4, 'x': 8, 'y': 4, 'z': 10}
```

• If we call the `sorted()` function on a dictionary, it returns an ordered list of all the keys.

```python
print(sorted(scrabbleScore))
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
```
Sorting By Value

• This behavior isn’t super useful for Scrabble
• What if we wanted to sort based on the scores of the letters (from highest to lowest) instead?
• This known as a **sort-by-value** as opposed to **sort-by-key**
• As before, using `sorted()` with a **key** function (not be confused with the keys in the dictionary!) comes in handy.
• We’ll need to spend just a little more effort to come up with a suitable **key** function
• Ex: Jupyter notebook
Sorting By Value

- We first use the `items()` method to generate a list of tuples, where each tuple is a key-value pair.

- We then sort this list based on value (`second` element of each tuple).

```python
def getScrabbleScore(letterScoreTuple):
    
    Takes a tuple corresponding to (letter, score) and returns the score
    
    return letterScoreTuple[1]
```

```python
# First use the `items` method to get a list of (key, value) tuples
# and then sort using a key function
scrabbleItems = getScrabbleScore.items()
sortedScrabbleItems = sorted(scrabbleItems, key=getScrabbleScore, reverse=True)
print(sortedScrabbleItems[0:3], '...', sortedScrabbleItems[-3:])
```

- Note that we can also use a list comprehension after to extract just the keys if desired.

```python
[('q', 10), ('z', 10), ('j', 8)] ... [('s', 1), ('t', 1), ('u', 1)]
```
Advantages of Using Dictionaries

• Easy access based on keys (some sort of named reference) rather than indices (referenced by position in the list)

• For example, to access the Scrabble score for ‘p’ using a dictionary we simply ask for `scrabbleScore['p']`

• In contrast suppose the letters and scores are stored as two ordered lists (or even as a list of lists) that looks like this:

```python
print(letters[0:3], '...', letters[-3:])
print(scores[0:3], '...', scores[-3:])
```

```python
['a', 'b', 'c'] ... ['x', 'y', 'z']
[1, 3, 3] ... [8, 4, 10]
```

• We now have to be able to “recall” or find where ‘p’ is located in these lists and then extract its corresponding score
Advantages of Using Dictionaries

• Side-by-side this is what that would look like

```python
# dictionary access
scoreDict = scrabbleScore['p']

# list access
indexP = letters.index('p')
scoreList = scores[indexP]

# confirm they're the same
scoreDict == scoreList
```

True

• Though list access seems like a minor notational inconvenience, it also has computational implications

• Every time we try to find the position of a letter in our list using the `index()` method, we are actually looping over each letter until we find the one we’re looking for (in fact, we could have re-written the list access explicitly using a loop.)

• The dictionary access on the other hand instantly knows what it’s looking for
Advantages of Using Dictionaries

• Let's see how this difference plays out when we ask the computer to do 6 million queries (people across the world play a lot of Scrabble!)

• We'll use our old friend the `time` module for this

```python
# random letters to query several times
randomLetters = ['a', 'l', 'q', 's', 'y', 'z']*1000000
print("Number of queries", len(randomLetters))

Number of queries 6000000
```

• Ex: Jupyter notebook
Advantages of Using Dictionaries

• Even in this really simple case, dictionaries give a 4x speed-up!

```python
# generate list of letters and scores
letters = list(scrabbleScore.keys())
scores = list(scrabbleScore.values())

# time using list operations to compute total score
startTime = time.time()
totalScore = 0

for query in randomLetters:
    index = letters.index(query)
    totalScore += scores[index]

endTime = time.time()
timeList = endTime - startTime
print("Time taken using a list", round(timeList, 3), "seconds")

Time taken using a list 2.219 seconds

# time using dictionaries to compute total score
startTime = time.time()
totalScore = 0

for query in randomLetters:
    totalScore += scrabbleScore[query]

endTime = time.time()
timeDict = endTime - startTime
print("Time taken using a dictionary", round(timeDict, 3), "seconds")

Time taken using a dictionary 0.589 seconds
```
Summary: Benefits of Dictionaries

• Dictionaries can be a more efficient alternative to sequences for some operations
  
• When we insert into an ordered sequence like a list
  
  • We need to "move over" all elements to make space
  
  • This is an expensive operation: worst case (insert at beginning of list) takes time proportional to number of items stored in list

• When we search for an item in an list:
  
  • If we are not careful we might have to compare to every item stored
  
• Using a dictionary instead of a list means:
  
  • Can insert more efficiently (without having to move any other item)
  
  • Can support more efficient (almost instantaneous!) queries on average (if keys are "hashes" of values)

• To learn more about about efficiency of data structures, take CS136/CS256!
Moving on...
New Unordered Data Structure: Sets

- Dictionaries are unordered **key, value** stores
- What if we only need an unordered "**collection**" of items?
  - We can use a new data structure: **sets**
- Sets are **mutable, unordered** collections of **immutable** objects
- Sets are written as comma separated values between curly braces
- Like keys in a dictionary, values in a set must be **unique** and **immutable**
  - Sets can be an effective way of **eliminating duplicate values**

```python
nums = {42, 17, 8, 57, 23}
flowers = {'tulips', 'daffodils', 'asters', 'daisies'}
potters = {('Ron', 'Weasley'), ('Luna', 'Lovegood'), ('Hermione', 'Granger')}
emptySet = set() # empty set
```
New Unordered Data Structure: Sets

- **Question:** What is the potential downside of removing duplicates w/sets?

```python
In [1]: firstChoice = ['a', 'b', 'a', 'a', 'b', 'c']

In [2]: uniquenesses = set(firstChoice)

Out[2]: {'a', 'b', 'c'}

In [3]: set("abra\kadabra")

Out[3]: {'a', 'b', 'd', 'k', 'r'}
```
New Unordered Data Structure: Sets

• **Question:** What is the potential downside of removing duplicates w/sets?
  • Loses ordering of elements

In [1]: firstChoice = ['a', 'b', 'a', 'a', 'b', 'c']

In [2]: uniqueness = set(firstChoice)
   uniqueness

Out[2]: {'a', 'b', 'c'}

In [3]: set("aabbrakadabra")

Out[3]: {'a', 'b', 'd', 'k', 'r'}
Sets: Membership and Iteration

- Can check membership in a set using `in` and `not in`.
- Can check length of a set using `len()`.
- Can iterate over values in a loop (order will be arbitrary).

```python
In [14]: nums = {42, 17, 8, 57, 23}
   flowers = {'tulips', 'daffodils', 'asters', 'daisies'}

In [15]: 16 in nums
Out[15]: False

In [16]: 'asters' in flowers
Out[16]: True

In [17]: len(flowers)
Out[17]: 4

In [18]: # iterable
   for f in flowers:
       print(f, end=" ")

   tulips daisies daffodils asters
```

end = """ prevents new line
Sets are Unordered

- Therefore we **cannot**:
  - Index into a set (no notion of “position”)
  - Concatenate two sets (concatenation implies ordering)
  - Create a set of **mutable** objects:
    - Such as lists, sets, and dictionaries

```python
In [21]: {{3, 2}, [1, 5, 4]}
```

```
--
TypeError

Traceback (most recent call last)
/var/folders/h8/n5myy3jd1d7cfv42cw42f1t80000gn/T/ipykernel_10595/3548805500.py in <module>
----> 1 {{3, 2}, [1, 5, 4]}

TypeError: unhashable type: 'list'
```
Set Methods Summary

- `s.add(item)`: changes the set `s` by adding item to it.
- `s.remove(item)`: changes the set `s` by removing item from `s`.
  - If item is not in `s`, a `KeyError` occurs.

The following operations return a new set.

- `s1.union(s2)` or `s1 | s2`: returns a new set that has all elements that are either in `s1` or `s2`.
- `s1.intersection(s2)` or `s1 & s2`: returns a new set that has all the elements that are in both sets.
- `s1.difference(s2)` or `s1 - s2`: returns a new set that has all the elements of `s1` that are not in `s2`.
- `s1 |= s2, s1 &= s2, s1 -= s2` are versions of `|, &`, `−` that mutate `s1` to become the result of the operation on the two sets.
An Overview of Python Data Structures (so far!)
# Python Data Structures at a Glance

<table>
<thead>
<tr>
<th></th>
<th>Lists</th>
<th>Tuples</th>
<th>Dictionaries</th>
<th>Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Order</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Mutability</strong></td>
<td>Yes</td>
<td>No</td>
<td>Yes (keys are immutable)</td>
<td>Yes (items are immutable)</td>
</tr>
<tr>
<td><strong>Iterable</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Comprehensions</strong></td>
<td>Yes</td>
<td>Yes (need to enclose in <code>tuple</code>)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Methods</strong></td>
<td>.append(), .extend(), .count(), .index(), etc</td>
<td>.count(), .index(), etc</td>
<td>.get(), .pop(), etc</td>
<td>.add(), .remove(), etc</td>
</tr>
</tbody>
</table>
# Python Data Structures at a Glance

<table>
<thead>
<tr>
<th></th>
<th>Lists</th>
<th>Tuples</th>
<th>Dictionaries</th>
<th>Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Order</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Mutability</strong></td>
<td>Yes</td>
<td>No</td>
<td>Yes (keys are immutable)</td>
<td>Yes (items are immutable)</td>
</tr>
<tr>
<td><strong>Iterable</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Comprehensions</strong></td>
<td>Yes (need to enclose in <code>tuple</code>)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Methods

- `append()`, `extend()`, `count()`, `index()`, etc
- `add()`, `remove()`, etc
Does Order Matter?

• Examples where order in data is important:
  • Ranked ballots
  • Queues
  • Words in a sentence
  • Tables/Matrices

• Tuples or lists?
  • Do we need to add/remove items dynamically?
    • If yes, use lists (they are mutable!)
  • If data stays same (no changes), use tuples (more space efficient)
  • Even though you can concatenate items to tuples, it is not efficient, as it requires “copying over all the data” and creating a new tuple
Unordered Collections

• When storing a collection of data with no implicit ordering:
  • Use dictionaries or sets
  • Dictionaries are more appropriate when there is a key, value pair
  • Better performance in general as compared to ordered structures

• Suppose we want to store student data in this course and quickly look up info for a given unix ID. Which data structure should we use?
  • Info may contain student name, class year, section, etc

• Can store a dictionary of dictionaries (just like lists of lists!)

```python
hpDict = { 'hp23': { 'name': 'Harry James Potter', 'house': 'Gryffindor', 'patronus': 'Stag'},
          'hg3': { 'name': 'Hermione Jean Granger', 'house': 'Gryffindor', 'patronus': 'Otter'},
          'll4': { 'name': 'Luna Lovegood', 'house': 'Ravenclaw', 'patronus': 'Hare'}}
```