CS134:
Dictionaries & Comparison to Lists
Announcements & Logistics

- **Practice midterm** on Glow
  - Midterm from F18 with slight modifications to fit our syllabus
- **Lab 5** is a short debugging lab, due Friday at noon for everyone
  - Expect most people to finish it during scheduled lab period
- **Midterm**: Thu Oct 20 6 - 7:30pm, 8 - 9:30pm in TCL 123 (Wege)
  - TCL 206 reserved for reduced distractions/extra time (pick up exam in TCL 123)
- **Midterm review**: Tue Oct 18 8-9:30pm in TCL 123
  - Try to review practice midterm before then!
- **No class** Fri Oct 21st

Do You Have Any Questions?
Midterm Material

- Labs 1-4
  - Lab 1: Intro to Python
  - Lab 2: Day of the week (if else statements)
  - Lab 3: Word puzzles (strings and loops)
  - Lab 4: Every vote counts (lists, strings, lists of lists, loops)
- Homeworks 2-5
- Lectures 1-15 (up to dictionaries) + Jupyter notebooks
- Book: parts of Ch 1, 2, 3, 5, 8, 9, 10, 12 (we won’t ask questions directly from the book)
Midterm Topics

• Variables, Types & Arithmetic Operators ( %, //, /, etc)
• Functions, Booleans and Conditionals (if elif else)
• Iteration: for loops, while loops, nested loops, list comprehensions
• Sequences:
  • Strings: string methods, iteration, etc
  • Lists: list methods (append, extend), iteration, lists of lists, etc
  • Ranges and tuples
  • Operators: +, [ ], [ : ], *, in/not in, etc
• File reading: with open(...) as
• Mutability and aliasing implications for lists
• Misc: doctests, simplification of verbose code
Last Time

• Discussed stable sorting and ways to override it using key function

• Introduced a new data structure: **dictionary**
  
  • Unordered, **mutable** key, value pairs

  • Keys must be **immutable** and **unique**, while values need not be

  • E.g., a dictionary storing key-value pairs of names and ages:
    
    ```json
    {"Charlie": 8, "Linus": 5, "Snoopy": 72}
    ```

• (No dictionaries on the midterm)
Today’s Plan

• Discuss dictionaries in more detail with examples
• Learn about dictionary methods such as `.get()`
• Use dictionaries to find the most frequent words from a wordList
• Examine differences between storing data as lists/nested lists vs. dictionaries
Recap: Dictionaries

- A **dictionary** is a **mutable** collection that maps **keys** to **values**
  - Enclosed with curly brackets, and contains comma-separated items
  - An item in the dictionary pair is a **colon-separated key, value pair**.
  - There is no ordering between the keys of a dictionary!

```python
# sample dictionary
zipCodes = {'01267': 'Williamstown', '60606': 'Chicago',
            '48202': 'Detroit', '97210': 'Portland'}
```

- **Keys** must be **immutable** and **unique**
- **Values** can any Python object (numbers, strings, lists, tuples, etc.)
Accessing/Adding Items in a Dictionary

• We access a dictionary using its keys as the “subscript”
  • If the key exists, its value is returned. Otherwise, we get a **KeyError**

```python
>>> # sample dictionary
>>> zipCodes = {
    "01267": "Williamstown",
    "60606": "Chicago",
    "48202": "Detroit",
    "97210": "Portland"
}
```

```python
>>> # what US city has this zip code?
>>> zipCodes["60606"]
'Chicago'
```

• To add a new key, value pair, we assign the key to the value using:
  ```python
dictName[key] = value
  ```
  • If the key already exists, an assignment will **overwrite** its value and assign it the new value to the existing key

```python
>>> zipCodes["11777"] = "Port Jefferson"
```

Add a new key, value pair '11777': 'Port Jefferson'
Iterating Over a Dictionary

- Can **iterate over the keys** of a dictionary directly in a for loop.
- Note: In Python 3.6 and beyond, the keys and values of a dictionary are **iterated over in the same order in which they were created**.

```python
>>> calendar = {
    "Jan": 31, "Feb": 28, "Mar": 31, "Apr": 30,
    "May": 31, "Jun": 30, "Jul": 31, "Aug": 31,
    "Sep": 30, "Oct": 31, "Nov": 30, "Dec": 31
}

>>> for day in calendar:
...   print(day, calendar[day], end=" ")
Jan 31 Feb 28 Mar 31 Apr 30 May 31 Jun 30
Jul 31 Aug 31 Sep 30 Oct 31 Nov 30 Dec 31
```

An aside: This changes the behavior of print to use spaces instead of new lines.
Computing Frequency
Computing a frequency

• One common use of a dictionary is to store **frequencies**.
• Let's write a function `frequency()` that takes as input a list of strings `wordList` and returns a dictionary `freqDict` with the unique strings in `wordList` as keys, and their number of occurrences (ints) in `wordList` as values.
• For example if `wordList` is:

  ```
  ['hello', 'world', 'hello', 'earth', 'hello', 'earth']
  ```

the function should return a dictionary with the following items:

  ```
  {'hello': 3, 'world': 1, 'earth': 2}
  ```
Computing a frequency

- One common use of a dictionary is to store frequencies.
- Let's write a function `frequency()` that takes as input a list of strings `wordList` and returns a dictionary `freqDict` with the unique strings in `wordList` as keys, and their number of occurrences (ints) in `wordList` as values

```python
def frequencyOld(wordList):
    """"""Given a list of words, returns a dictionary of word frequencies"""
    freqDict = {}  # initialize accumulator as empty dict
    for word in wordList:
        if word not in freqDict:
            freqDict[word] = 1  # add key with count 1
        else:
            freqDict[word] += 1  # update count
    return freqDict
```
Useful Dictionary Method: `.get()`

- The following code pattern is very common when using dictionaries:

  ```python
  if aKey not in myDict:
      myDict[aKey] = initVal + incrementVal  # add key
  else:  # if already exists
      myDict[aKey] += incrementVal  # update val
  ```

- Rather than writing the `if, else` block as shown above, we can use the `.get()` method for dictionaries
Useful Dictionary Method: `.get()`

- The following code pattern is very common when using dictionaries:

```python
if aKey not in myDict:
    myDict[aKey] = initVal + incrementVal  # add key
else:  # if already exists
    myDict[aKey] += incrementVal  # update val
```

- Rather than writing the `if, else` block as shown above, we can use the `.get()` method for dictionaries

```python
myDict[aKey] = myDict.get(aKey, initVal) + incrementVal
```
Useful Dictionary Method: `.get()`

- `.get()` method is an alternative to using `[]` to get the value associated with a key in a dictionary; eliminates the need to check for the key’s existence beforehand.

- `.get()` takes two arguments: a **key**, and an **optional** default **value** to use if the key is not in the dictionary.

- It returns the **value** associated with the given **key**, and if **key** does not exist, it returns the **default value** (if given), otherwise returns **None**.

- Syntax: `value = myDict.get(aKey, defaultVal)`
Useful Dictionary Method: `.get()`

- `.get()` method **does not modify the dictionary** it is called on

```python
>>> ids = {"ikh1": "Iris", "jra1": "Jeannie", "lpd2": "Lida"}
>>> ids.get("jra1", "Ephelia")
'Jeannie'

>>> ids.get("xyz1", "Ephelia")
'Ephelia'

>>> ids # .get(..) does not change the dictionary!
{'ikh1': 'Iris', 'jra1': 'Jeannie', 'lpd2': 'Lida'}

>>> print(ids.get("xyz1"))
None
```
Computing frequency Improved

- Let's rewrite our frequency function using .get() instead of if else

```python
def frequencyOld(wordList):
    """Given a list of words, returns a dictionary of word frequencies""
    freqDict = {}  # initialize accumulator as empty dict
    for word in wordList:
        if word not in freqDict:
            freqDict[word] = 1  # add key with count 1
        else:
            freqDict[word] += 1  # update count
    return freqDict
```

- What should we write instead inside the for loop?
Computing frequency Improved

- Let's rewrite our `frequency` function using `.get()` instead of `if else`

```python
def frequencyOld(wordList):
    """Given a list of words, returns a dictionary of word frequencies""
    freqDict = {}  # initialize accumulator as empty dict
    for word in wordList:
        if word not in freqDict:
            freqDict[word] = 1  # add key with count 1
        else:
            freqDict[word] += 1  # update count
    return freqDict
```

- What should we write instead inside the for loop?

```python
def frequency(wordList):
    """Given a list of words, returns a dictionary of word frequencies""
    freqDict = {}  # initialize accumulator as empty dict
    for word in wordList:
        freqDict[word] = freqDict.get(word, 0) + 1
    return freqDict
```
Other Dictionary Methods
Dictionary Methods: `keys()`, `values()`, `items()`

- Dictionary methods `keys()`, `values()`, `items()`: return a (list-like) object containing only the keys, values, and items, respectively.
- Note: We don't use these very often in practice

```python

>>> calendar.keys()

>>> calendar.values()
dict_values([31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31])

>>> calendar.items()
```
## Summary of Dictionary Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Result</th>
<th>Mutates dict?</th>
</tr>
</thead>
<tbody>
<tr>
<td>.keys()</td>
<td>Returns all keys as a <code>dict_keys</code> object</td>
<td>No</td>
</tr>
<tr>
<td>.values()</td>
<td>Returns all values as a <code>dict_values</code> object</td>
<td>No</td>
</tr>
<tr>
<td>.items()</td>
<td>Returns all (key, value) pairs as a <code>dict_items</code> object</td>
<td>No</td>
</tr>
<tr>
<td>.get(key, val)</td>
<td>Returns corresponding value if <code>key</code> in dict, else returns <code>val</code>. Second argument is optional, defaults to <code>None</code>.</td>
<td>No</td>
</tr>
<tr>
<td>.pop(key)</td>
<td>Removes key:value pair with given <code>key</code> from dict and returns associated <code>val</code>. <code>KeyError</code> if key not in dict.</td>
<td>Yes</td>
</tr>
<tr>
<td>.update(dict2)</td>
<td>Adds new key:value pairs from <code>dict2</code> to dict, replacing any key:value pairs with existing key</td>
<td>Yes</td>
</tr>
<tr>
<td>.clear()</td>
<td>Removes all items from the dict.</td>
<td>Yes</td>
</tr>
</tbody>
</table>
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)
Dictionary Comprehensions

- Like list comprehensions, dictionary comprehensions are 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}
>>> print(days30)
{'Apr': 30, 'Jun': 30, 'Sep': 30, 'Nov': 30}
```
Advantages of Using Dictionaries

- Easy access based on **keys** rather than **indices** (or position)
- For example, recall our Scrabble score example

```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}
```

- To access the Scrabble score for 'p' using a dictionary we simply ask for `scrabbleScore['p']`
- Difficult to accomplish with lists!
  - Store letters and scores are stored as two “parallel” ordered lists?
    Or a list of lists/tuples?
  - We have to 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']
```

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

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

- Finding the position of a letter in a list requires **looping** over each letter until we find the one we're looking for (this is what `index()` does!)

- 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
>>> randomLetter = ['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
```
Benefits of Dictionaries

- **Dictionaries** are *more efficient* than lists for *some common operations*
  - When we **insert** into an ordered sequence (e.g., 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 ordered sequence:
    - We might have to loop and check every item stored
  - Using a **dictionary** instead of a list means:
    - Can **insert** more efficiently (without having to move any other items)
    - Can support more efficient **searching** (just look up key, no loop required)
- To learn more about about efficiency of data structures, take CS136/CS256!
The end!