CS 134:
Dictionaries & Comparison to Lists
Announcements & Logistics

• **Practice midterm** on Glow
  • Two versions: with and without solutions
  • Midterm from F18 with slight modifications to fit our syllabus
• **Lab 5** will be a short debugging lab released today
  • Expect most people to finish it during scheduled lab period
• **Midterm**: Thu Mar 17th. Slots: 6 - 7:30 pm, 8 - 9:30 pm in **Wachenheim B11/002**
  • One room reserved for reduced distractions/extra time
• **Midterm review**: Tue Mar 15th, 7 - 8:30 pm in **TPL 203**
  • Try to review practice midterm before then!

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, loops)
- Homeworks 2-5
- Lectures 1-15 + 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:
  • Operators: +, [ ], [ : ], *, in/not in, etc
  • Strings: string methods, iteration, etc
  • Lists: list methods (append, extend), iteration, lists of lists, etc
  • Ranges and tuples
• File reading: with … as block
• Mutability and aliasing implications
• 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:
    {“Harry”: 12, “Hermione”: 12, “Hagrid”: 60}
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 an **immutable** type such as ints, strings, or tuples
- Keys of a dictionary must be **unique**: no duplicates allowed!
- **Values** can any Python object (numbers, strings, lists, tuples, etc.)
Accessing Items in a Dictionary

- Dictionaries are unordered so we cannot index into them: no notion of first or second item, etc.
- We access a dictionary using its keys as the subscript
  - If the key exists, its corresponding value is returned
  - If the key does not exist, it leads to a **KeyError**

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

# what US city has this zip code?
zipCodes['60606']
```

```
Out[2]: 'Chicago'
```

```
# what US city has this zip code?
zipCodes['48202']
```

```
Out[3]: 'Detroit'
```
Adding a Key, Value Pair

- Dictionaries are mutable, so we can add items or remove items from it.
- To add a new key, value pair, we can simply assign the key to the value using: `dictName[key] = value`

```python
In [5]: zipCodes['11777'] = 'Port Jefferson'
```

```python
In [6]: zipCodes
```

```
Out[6]: {'01267': 'Williamstown',
        '60606': 'Chicago',
        '48202': 'Detroit',
        '97210': 'Portland',
        '11777': 'Port Jefferson'}
```

- If the key already exists, an assignment operation as above will **overwrite** its value and assign it the new value.
Operations on Dictionaries

• Just like sequences, we can use the `len()` function on dictionaries to find out the number of keys it contains.

• To check if a key exists (or does not exist) in a dictionary, we can use the `in` (`not in`) operator respectively.

| In [6]: | zipCodes |
| Out[6]: | {'01267': 'Williamstown', '60606': 'Chicago', '48202': 'Detroit', '97210': 'Portland', '11777': 'Port Jefferson'} |

| In [8]: | '90210' in zipCodes |
| Out[8]: | False |

| In [9]: | '01267' in zipCodes |
| Out[9]: | True |

Should always check if a key exists before accessing its value in a dictionary.
Creating Dictionaries

• Several ways to create dictionaries:
  • **Direct assignment**: provide key, value pairs delimited with `{ }`
  • Start with empty dict and add key, value pairs
    • Empty dict is `{}` or `dict()`
  • Apply the built-in function `dict()` to a list of tuples

In [1]:  # direct assignment
    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}

**Note:** keys may be listed in any order
Creating Dictionaries

- Direct assignment: provide key, value pairs delimited with `{ }`
- Start with empty dict and add key, value pairs
  - Empty dict is `{}` or `dict()`
- Apply the built-in function `dict()` to a list of tuples

```python
In [2]: # accumulate in a dictionary
verse = "let it be, let it be, let it be, let it be, there will be an answer, let it be"
counts = {} # empty dictionary
for line in verse.split(',', '):  
    if line not in counts:  
        counts[line] = 1 # initialize count
    else:  
        counts[line] += 1 # update count

Out[2]: {'let it be': 5, 'there will be an answer': 1}
```

```python
In [3]: # use dict() function
dict([(\'a\', 5), (\'b\', 7), (\'c\', 10)])

Out[3]: {\'a\': 5, \'b\': 7, \'c\': 10}
```

Note: keys may be listed in any order
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**.

• In general, this behavior may vary across different Python versions, and it depends on the dictionary’s history of insertions and deletions.

```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
Dictionary Example: frequency

• Let’s write a function `frequency` that takes as input a list of words `wordList` and returns a dictionary `freqDict` with the unique words in `wordList` as keys, and their number of occurrences 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}
```
Dictionary Example: frequency

- Let's write a function `frequency` that takes as input a list of words `wordList` and returns a dictionary `freqDict` with the unique words in `wordList` as keys, and their number of occurrences in `wordList` as values.

```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:
        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 extremely common when using dictionaries:

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

- Instead of using `if, else` to do above, it is preferable to use the `.get()` method for dictionaries instead
Useful Dictionary Method: `.get()`

- `.get()` method is an alternative to using subscript notation `[]` to get the value associated with a key in a dictionary *without* checking for its existence.
- It 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.
- If key does not exist it returns the default value (if given), otherwise returns `None`.
- Syntax: `val = myDict.get(aKey, defaultVal)`

  **key whose value we are looking for in** `myDict`

  **if key doesn't exist, return this default value**
Useful Dictionary Method: `.get()`

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

```python
ids = {'rb17': 'Rohit', 'jral': 'Jeannie',
       'sfreund': 'Steve', 'lpd2': 'Lida'}

ids.get('lpd2', 'Ephelia')
'Lida'

ids.get('ss32', 'Ephelia')
'Ephelia'

ids # .get does not change the dictionary
{'rb17': 'Rohit', 'jral': 'Jeannie', 'sfreund': 'Steve', 'lpd2': 'Lida'}

print(ids.get('ks123'))
None
```
Example: `frequency` with `.get()`

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

```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:
        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:
        # what should we write instead?
        freqDict[word] = freqDict.get(word, 0) + 1
    return freqDict
```
Dictionary Methods: keys(), values(), items()

- Dictionary methods keys(), values(), items(): return a (list like) object containing only the keys, values, and items, respectively.

```python

calendar.keys()

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

calendar.items()
```
Note: Iterating over/membership in Dicts

By default loops and membership operators iterate over **keys** in the dictionary. Hence, we rarely need to use `.keys()` explicitly.

When iterating over the keys in a dictionary, just write

```
for someKey in someDict:
```

rather than

```
for someKey in someDict.keys():
```

because they have a similar meaning, but the latter creates an unnecessary object.

Similarly, when testing if a key is in a dictionary, just write

```
if someKey in someDict:
```

rather than

```
if someKey in someDict.keys():
```

Image Source: (http://cs111.wellesley.edu/spring19)
# Summary of Dictionary Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Result</th>
<th>Mutates dict?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.keys()</code></td>
<td>Returns all keys as a <code>dict_keys</code> object</td>
<td>No</td>
</tr>
<tr>
<td><code>.values()</code></td>
<td>Returns all values as a <code>dict_values</code> object</td>
<td>No</td>
</tr>
<tr>
<td><code>.items()</code></td>
<td>Returns (key, value) pairs as a <code>dict_items</code> object</td>
<td>No</td>
</tr>
<tr>
<td><code>.get(key [, val])</code></td>
<td>Returns corresponding value if <code>key</code> in dict, else returns <code>val</code>. The notation <code>[ , val]</code> means that the second argument <code>val</code> is optional and can be omitted. If it is not specified, it defaults to <code>None</code>.</td>
<td>No</td>
</tr>
<tr>
<td><code>.pop(key)</code></td>
<td>Removes key:val pair with given <code>key</code> from dict and returns associated val. Signals <code>Key Error</code> if key not in dict.</td>
<td>Yes</td>
</tr>
<tr>
<td><code>.update(dict2)</code></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><code>.clear()</code></td>
<td>Removes all items from the dict.</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Image Source: [http://cs111.wellesley.edu/spring19](http://cs111.wellesley.edu/spring19)
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

- 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
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}

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

days30

{'Apr': 30, 'Jun': 30, 'Sep': 30, 'Nov': 30}
```
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}
```

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

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

```bash
['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

• However, this behavior isn’t super interesting in our case. What if we wanted to sort 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 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]

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

[['q', 10], ['z', 10], ['j', 8]] ... [['s', 1], ['t', 1], ['u', 1]]
```

• We can also use a list comprehension after to extract just the keys if desired.
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 when 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:])
```

- 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']
```

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

```python
# 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, 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 \texttt{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
```
Benefits of Dictionaries

• Dictionaries can be a **more efficient** alternative to lists 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 queries** on average (if keys are "hashes" of values)
• To learn more about about efficiency of data structures, take CS136/CS256!