CSCI 134 Fall 2021:
Dictionaries & Plotting

October 18, 2021

Shikha Singh, 9AM
Jeannie Albrecht, 10AM
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 released today
  - Expect most people to finish it during scheduled lab period
  - (If you need more time, you have until Thur Oct 21 at 10pm)
- **No lecture on Wednesday Oct 20th!**
- **Midterm**: Wed Oct 20th: Slots: 6 - 7:30 pm, 8 - 9:30 pm in **TCL 202**
  - **TCL 206** will be available for reduced distractions/extra time
- **Midterm review**: Today Mon, Oct 18th 7 - 8 pm in **TPL 203**

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 1-14 + Jupyter notebooks
- Book: parts of Ch 1, 2, 3, 5, 8, 9, 10, 12
Midterm Topics

- Variables, Types & Operators ( %, //, /, +, *, etc)
- Functions, Booleans and Conditionals (if elif else)
- Iteration: for loops, while loops, nested loops, list comprehensions
- Sequences:
  - Strings: string methods, in/not in, iteration, etc
  - Lists: list methods (append, extend), in/not in, iteration, lists of lists, etc
  - Ranges, and tuples
- File reading: with … as block
- Mutability and aliasing
- Misc: special variables__all__, __name__, modules vs scripts, doctests
Last Time

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

• Introduced anonymous functions using **lambda expressions**

• E.g., `sorted(seq, key=lambda el: el[-1])`
  
  • Interpret as for `el in seq: use el[-1] to sort seq`
  
  • For **each element in the sequence**, `sorted` **calls the key function on element** to figure out the order of sorting

• Introduced a new data structure: **dictionary**
  
  • unordered, **mutable** key, value store
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
• Use matplotlib to plot these words
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
In [1]: # sample dictionary
    zipCodes = {'01267': 'Williamstown', '60606': 'Chicago',
                '48202': 'Detroit', '97210': 'Portland'}

In [2]: # what US city has this zip code?
    zipCodes['60606']

Out[2]: 'Chicago'

value associated with key '60606'

In [3]: # 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
```

```python
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.

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

In [7]: len(zipCodes)
Out[7]: 5

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 it's 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}

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
In [3]: 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):
    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
```
Important 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
Important 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)`
Important Dictionary Method: `.get()`

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

```python
In [46]: ids = {'ss32': 'Shikha', 'jral': 'Jeannie',
               'kas10': 'Kelly', 'lpd2': 'Lida'}

In [53]: ids.get('kas10', 'Ephelia')
Out[53]: 'Kelly'

In [54]: ids.get('srm2', 'Ephelia')
Out[54]: 'Ephelia'

In [49]: ids # .get does not change the dictionary
Out[49]: {'ss32': 'Shikha', 'jral': 'Jeannie', 'kas10': 'Kelly', 'lpd2': 'Lida'}

In [50]: print(ids.get('ks123'))
None
```
Important Dictionary Method: `.get()`

- We can now simplify this considerably using `get`:

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

- Simpler, more efficient, and preferred alternative:

```python
myDict[aKey] = myDict.get(aKey, initialVal - step) + step
```
Example: frequency Improved

- 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?
Example: `frequency` Improved

• 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

In [2]: calendar.keys()


In [3]: calendar.values()

Out[3]: dict_values([31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31])

In [4]: calendar.items()

Iterating over/membership in Dicts

When iterating over the keys in a dictionary, just write

```python
for someKey in someDict:
```

rather than

```python
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

```python
if someKey in someDict:
```

rather than

```python
if someKey in someDict.keys():
```
### 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 (key, value) pairs as a <code>dict_items</code> object</td>
<td>No</td>
</tr>
</tbody>
</table>
| .get(key [, val]) | Returns corresponding value if `key` in dict,  
                      |               |               |
|              | else returns `val`. The notation [, `val`] means that                |               |               |
|              | the second argument `val` is optional and can be omitted. If it is    |               |               |
|              | is not specified, it defaults to `None`.                              |               |               |
| .pop(key)    | Removes key:val pair with given `key` from dict and returns           | Yes           |
|              | associated val. Signals `KeyError` if key not in dict.                |               |               |
| .update(dict2) | Adds new key:value pairs from `dict2` to dict,  
                   |               |               |
|              | replacing any key:value pairs with existing key.                     | Yes           |
| .clear()     | Removes all items from the dict.                                      | Yes           |

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)


In [6]: days30 = {k: calendar[k] for k in calendar if calendar[k] == 30}

In [7]: days30

Out[7]: {'Apr': 30, 'Jun': 30, 'Sep': 30, 'Nov': 30}
Sorting a Dictionary

• We can use lambda sorting to sort dictionaries based on their key, or value, or any other function of either

• Recall that when iterating over a dictionary, we are iterating over its keys (as opposed to its values)

• This is important because sorted() runs an implicit for loop over the the data type being sorted

• Example:

```python
>>> sorted(myDict, key=lambda k: myDict[k])
```

for each key in myDict return its value in myDict
Plotting with **matplotlib**

- Let's plot the most frequent words in *Pride and Prejudice*
- A plot is a graphical technique for representing a data set, usually as a graph showing the relationship between two or more variables
- We'll be using Python's **matplotlib** library to make plots/graphs
- The best way to learn how to plot different types of graphs is to read the documentation and see examples

**Resources**

- **matplotlib examples**: [http://matplotlib.org/examples](http://matplotlib.org/examples)
- **pyplot documentation**: [http://matplotlib.org/api/pyplot_summary.html](http://matplotlib.org/api/pyplot_summary.html)
- **cool plots**: [https://matplotlib.org/gallery.html](https://matplotlib.org/gallery.html)
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4])
plt.show()

If only one list is provided, Python assumes it is as the points on the y axis (x values start at 0)
Plotting Basics: Plot function

```python
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4], [10, 14, 15, 18])
plt.show()
```

Equivalent to saying plot the points 
(1, 10), (2, 14), (3, 15), (4, 18)
Plotting Examples: See Notebook

• See lecture Jupyter notebook for variable `matplotlib` examples
• We will plot the bar chart for the most frequent words in Pride and Prejudice on the x axis, and their frequency on the y axis.

![Frequency of common words in Pride and Prejudice](image)
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!