Dictionaries

Looking up English words in the dictionary

**Sequence**: a group of things that come one after the other

**Collection**: a group of (interesting) things brought together for some purpose

Is a sequence a collection?  
Is a collection a sequence?

Python sequences and collections

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>a mutable sequence of arbitrary objects</td>
<td>[-100, &quot;blue&quot;, (1, 10), True]</td>
</tr>
<tr>
<td>tuple</td>
<td>an immutable sequence of arbitrary objects</td>
<td>(2017, &quot;Mar&quot;, 2)</td>
</tr>
<tr>
<td>string</td>
<td>an immutable sequence of characters</td>
<td>&quot;Go Wellesley!&quot;</td>
</tr>
<tr>
<td>set</td>
<td>an unordered collection of distinct objects.</td>
<td>{1, 4, 5, 23}</td>
</tr>
<tr>
<td>dict</td>
<td>An unordered collection of key:value pairs, where keys are immutable and values are any Python objects</td>
<td>{&quot;orange&quot;: &quot;fruit&quot;, 3: &quot;March&quot;, &quot;even&quot;: [2, 4, 6, 8]}</td>
</tr>
</tbody>
</table>

Properties of sequences & collections

- **Collections** (list, tuple, string, set, dict)
  - Find their length with `len`
  - Check membership in the collection with `in`
  - Are iterables (one can iterate over their elements with a loop)

- **Sequences** (list, tuple, string)
  - Use indices to access elements, e.g. `myList[2]`
  - Use slice operations to access subsequences, e.g. `myList[2:5]`

- **Mutable** (list, set, dict)
- **Immutable** (tuple, string)
Dictionaries

A Python dictionary is a mutable collection that maps keys to values.

A dictionary is enclosed with curly brackets and contains comma-separated pairs. A pair is a colon-separated key and value.

daysOfMonth = { 'Jan': 31, 'Feb': 28, 'Mar': 31, 'Apr': 30, ...

A dictionary is a mutable collection that maps keys to values.

keys and values

keys: any immutable type (or hashable value), such as numbers, strings, or tuples.
values: any Python object (numbers, strings, lists, tuples, dicts, sets, even functions)

students = { 'Jan': 31, 'Feb': 28, 'Mar': 31, 'Apr': 30, ...

Dictionaries

A dictionary is a mutable collection that maps keys to values.

A dictionary is enclosed with curly brackets and contains comma-separated pairs. A pair is a colon-separated key and value.

daysOfMonth = { 'Jan': 31, 'Feb': 28, 'Mar': 31, 'Apr': 30, ...

Dictionary Operations: subscripting

The value associated with a key is accessed using the same subscripting notation with square brackets used for list indexing:

In [6]: daysOfMonth['Oct']
Out[6]: 31

In [7]: student['dorm']
Out[7]: 'Munger Hall'

In [8]: student['year']
Out[8]: 2019

In [9]: student['CSMajor?']
Out[9]: True

In [10]: phones[9996541212]
Out[10]: 'Shakira'

In [11]: friends[('Oprah', 'Winfrey')]
Out[11]: [7897654321, 'O@winfrey.com']

Dictionary Operations: check with in before accessing

In [12]: daysOfMonth['October']

Subscripting a dictionary with an invalid key raises a KeyError:

In [13]: 'Oct' in daysOfMonth
Out[13]: True

In [14]: 'October' in daysOfMonth
Out[14]: False
Creating dictionaries

1. Literal dict: provide keys and pairs together

```
In [1]: scrabbleDict = {'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': 4, 'w': 4, 'x': 8, 'y': 4, 'z': 10}
```

2. Start with empty dict and add key/pairs

```
In [2]: cart = {}
In [3]: cart['oreos'] = 3.99
In [4]: cart['kiwis'] = 2.54
In [5]: cart
Out[5]: {'kiwis': 2.54, 'oreos': 3.99}
```

The order of key/pairs doesn't reflect the insertion order.

Exercise 1: The scrabble score

```
def scrabblePoints(letter):
    if letter in 'aeilnorstu':
        return 1
    elif letter in 'dg':
        return 2
    elif letter in 'bcmp':
        return 3
    elif letter in 'fhvwy':
        return 4
    elif letter in 'k':
        return 5
    elif letter in 'jx':
        return 8
    elif letter in 'qz':
        return 10
    return 0
```

Let's rewrite the function with the dict, `scrabbleDict`. Details about how `scrabbleDict` was created are in the next slide.

```
def scrabblePoints2(letter):
    # Let's rewrite the function with the dict, scrabbleDict. Details about how scrabbleDict was created are in the next slide.
    from string import lowercase
    letters = list(lowercase)
    scores = [1, 3, 3, 2, 1, 4, 2, 4, 1, 8, 5, 1, 3, 1, 1, 3, 10, 1, 1, 1, 4, 4, 8, 4, 10]
    zip is explained in slide 10-29.
    scrabbleDict = dict(zip(letters, scores))
In [12]: scrabbleDict
Out[12]: {'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': 4, 'w': 4, 'x': 8, 'y': 4, 'z': 10}
```

Creating dictionaries from a list of tuples

```
In [4]: dict([('a', 1), ('b', 3), ('c', 3)])
Out[4]: {'a': 1, 'b': 3, 'c': 3}
```

```
In [7]: from string import lowercase
In [8]: letters = list(lowercase)
In [9]: letters
Out[9]: ['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']
In [10]: scores = [1, 3, 3, 2, 1, 4, 2, 4, 1, 8, 5, 1, 3, 1, 1, 3, 10, 1, 1, 1, 4, 4, 8, 4, 10]
In [11]: scrabbleDict = dict(zip(letters, scores))
In [12]: scrabbleDict
Out[12]: {'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': 4, 'w': 4, 'x': 8, 'y': 4, 'z': 10}
```

Mutability in Dictionaries

**Dictionaries are mutable**

- Can add or remove key-value pairs.
- Can change the value associated with a key.

```
daysOfMonth['Feb'] = 29  # change for leap year
daysOfMonth['Dec'] = 31  # add new key and value
```

**However, dictionary keys must be immutable**

Eg: a list or a dict cannot be a key (only immutable values such as numbers, strings, and tuples)
**Mutability with hash**

A hash value is an integer number that is returned by the `hash` function.

```
In [25]: hash("Wellesley")
Out[25]: 1371402960993349759
```

```
In [26]: hash((2015, 10))
Out[26]: 3711745792089893406
```

```
In [27]: hash(1234)
Out[27]: 1234
```

Only immutable objects have hash values. We’ll get an error for mutable objects.

```
In [25]: hash([1, 2, 3])
```

```
In [26]: daysOfMonth[['Feb', 2015]] = 28
```

```
TypeError: unhashable type: 'list'
```

**Dictionaries are Mutable:**

The value associated with a key can be changed by combining subscript and assignment notation:

```
In [21]: student['dorm'] = 'Cazenove Hall'
```

```
In [22]: student
Out[22]: {'CSMajor?': True, 'dorm': 'Cazenove Hall', 'name': 'Georgia Dome', 'section': 2, 'year': 2019}
```

**Dictionaries are Mutable:**

A new key/value pair can be added by assigning to a key not already in the dictionary:

```
In [23]: student['psets'] = [95, 87, 73, 92]
```

```
In [24]: student
Out[24]: {'CSMajor?': True, 'dorm': 'Cazenove Hall', 'name': 'Georgia Dome', 'psets': [95, 87, 73, 92], 'section': 2, 'year': 2019}
```
Dictionaries are Mutable: remove key/value pair

A key/value pair can be removed by the `pop` method, which returns the old value in addition to removing the key/value pair:

```python
student = {'name': 'Georgia Dome', 'dorm': 'Cazenove Hall', 'year': 2019, 'CSMajor?': True, 'psets': [95, 87, 73, 92], 'section': 'student'}
```

In [25]: student.pop('section')
Out[25]: 2

In [26]: student
Out[26]: {'name': 'Georgia Dome', 'dorm': 'Cazenove Hall', 'year': 2019, 'CSMajor?': True, 'psets': [95, 87, 73, 92]}
```

Dictionaries are Mutable: update

An existing dictionary can be updated with new key/pair values through the `update` method.

```python
friends = {('Angelina', 'Jolie'): [242559191, 'jolie@pitt.org'], ('Oprah', 'Winfrey'): [7897654321, 'O@winfrey.com'], ('Stephen', 'Colbert'): [9993334444, 'host@cb.com']}
```

In [24]: newFriends = {('Harry', 'Potter'): [6172839531, 'hpotter@hogwarts.edu'], ('Stephen', 'Colbert'): [5553218674, 'host@cb.com']}

In [25]: friends.update(newFriends)

Entries with the same key will be replaced by the ones in the argument dictionary.

```python
In [26]: friends
Out[26]: {('Angelina', 'Jolie'): [242559191, 'jolie@pitt.org'], ('Harry', 'Potter'): [6172839531, 'hpotter@hogwarts.edu'], ('Oprah', 'Winfrey'): [7897654321, 'O@winfrey.com'], ('Stephen', 'Colbert'): [5553218674, 'host@cb.com']}
```

Dictionary Methods: keys, values, items

The `keys`, `values`, and `items` method invocations on a dictionary return, respectively, lists of keys, values, and key/value pairs for a dictionary:

```python
In [15]: daysOfMonth.keys()
```

# Note the lack of order to the keys!

```python
In [16]: daysOfMonth.values()
```

# Values have same order as keys from keys method invocation

```python
In [17]: daysOfMonth.items()
```

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Result</th>
<th>Mutates dict?</th>
</tr>
</thead>
<tbody>
<tr>
<td>keys</td>
<td>Returns all keys as a list</td>
<td>No</td>
</tr>
<tr>
<td>values</td>
<td>Returns all values as a list</td>
<td>No</td>
</tr>
<tr>
<td>items</td>
<td>Returns a list of (key, value) pairs</td>
<td>No</td>
</tr>
<tr>
<td>get(key[, val])</td>
<td>Returns corresponding value if key in dict, else returns val.</td>
<td>No</td>
</tr>
<tr>
<td>has_key(key)</td>
<td>Returns true if key in dict, else false</td>
<td>No</td>
</tr>
<tr>
<td>pop(key)</td>
<td>Removes key+val from dict and return val.</td>
<td>Yes</td>
</tr>
<tr>
<td>update(dict1)</td>
<td>Adds new key-value pairs to the dict, replaces existing key-value pairs.</td>
<td>Yes</td>
</tr>
<tr>
<td>clear</td>
<td>Removes all items from the dict.</td>
<td>Yes</td>
</tr>
<tr>
<td>iterkeys, iteritems, itervalues</td>
<td>Similar to keys, values, items, but more efficient for big dictionaries, because use “iterators”.</td>
<td>No</td>
</tr>
</tbody>
</table>

*The notation [, val] means an optional second parameter. This can be omitted.*
Iterating over a Dictionary [1]

keys, items, and values are useful for manipulating dictionaries with for loops

In [18]: for num in phones.keys():
   print num, phones[num]

5558671234 Justin Bieber
9996541212 Shakira
7811234567 Paula A. Johnson

Notice the tuple assignment. Look at the Lecture’s notebook for more info.

In [20]: for number, name in phones.items():
   print "Call", name, "at", number

Call Justin Bieber at 5558671234
Call Shakira at 9996541212
Call Paula A. Johnson at 7811234567

Application: Word Frequencies

A standard use case for dictionaries is to count the frequencies (num. of occurrences) of words in a list.

The algorithm for solving this problem (the needed steps) is this:

Create an empty dictionary.
For each word in the list of words:
  o If the word has not been encountered before, i.e., it is not yet a key in the dictionary, insert it as a key in with value 1.
  o If the word has been encountered before, i.e., it is already a key, increment the value associated with it.

Exercise 2: Word Frequencies

Define a function frequencies that, when given a list of words, returns a dictionary of word frequencies as described previously.

In [23]: frequencies(['I', 'know', 'that', 'I', 'said',
               'that', 'I', 'did'])
Out[27]: {'I': 3, 'did': 1, 'know': 1, 'said': 1, 'that': 2}

In [24]: frequencies('I know that I said that I did'.split())
Out[28]: {'I': 3, 'did': 1, 'know': 1, 'said': 1, 'that': 2}

Iterating over a Dictionary [2]

This behaves the same as for num in phones.keys() in slide 10-21. It is the recommended way to iterate over a dictionary and we will use it from now on. Don’t make it a habit to use keys with a for loop. That method is useful in some situations, but not during iteration. For more see 10-31.
Dictionary Operations: `get`

The `get` method is an alternative to using subscripting to get the value associated with a key in a dictionary. It takes two arguments:

1. the key
2. a default value to use if the key is not in the dictionary

In [25]: daysOfMonth.get('Oct', 'unknown')
Out[25]: 31

In [26]: daysOfMonth.get('OCT', 'unknown')
Out[26]: 'unknown'

Exercise 3: rewrite `frequency` with `get`

Rewrite the function `frequencies` from Exercise 2, so that it does not use an `if` statement.

Instead, it uses the `get` method, with the default value 0, for words that haven’t been seen before.

Exercise 4: Dictionary Key with Largest Value

Define a function `getKeyWithMaxValue` that, when given a dictionary, returns a key that corresponds to the largest value in the dictionary.

In [27]: getKeyWithMaxValue({'A': 0.25, 'E': 0.36, 'I': 0.16, 'O': 0.18, 'U': 0.05})
Out[27]: 'E'

In [28]: getKeyWithMaxValue({'Romney': 60933500, 'Obama': 65915796, 'Other': 2113596})
Out[28]: 'Obama'

The function `max` will be useful to find the largest value. Remember to use an early return.

Exercise 5: Reverse Lookup

Define a function `reverseDictionary` that, when given a dictionary, returns a new dictionary mapping the value in the given dictionary to a list of its corresponding keys.

In [29]: disneyOlympicResults = {'mickey': 'silver', 'minnie': 'gold', 'donald': 'bronze', 'daisy': 'bronze', 'goofy': 'bronze', 'ariel': 'silver', 'nemo': 'gold', 'mulan': 'silver', 'elsa': 'gold'}

In [30]: reverseDictionary(disneyOlympicResults)
Out[30]: {'bronze': ['donald', 'goofy', 'daisy'], 'gold': ['elsa', 'nemo', 'minnie'], 'silver': ['mickey', 'mulan', 'ariel']}
**Built-in function `zip`**

If we want to group list items pairwise, we can use the built-in function `zip`. It returns a list of tuples.

In [31]: lst1 = `range`(10, 15)
In [32]: lst2 = `range`(30, 35)
In [33]: `zip`(lst1, lst2)
Out[33]: [(10, 30), (11, 31), (12, 32), (13, 33), (14, 34)]

In [34]: lst3 = `list`('daisy')
In [35]: `zip`(lst1, lst3, lst2)
Out[35]: [(10, 'd', 30), (11, 'a', 31), (12, 'i', 32), (13, 's', 33), (14, 'y', 34)]

In [36]: `zip`(lst1, lst1[1:])
Out[36]: [(10, 11), (11, 12), (12, 13), (13, 14)]

Notice how the length of returned list, depends on length of smallest list.

---

**Unpairing with `zip`**

In [37]: from random import randint, choice
In [38]: from string import uppercase
In [39]: uppercase
Out[39]: 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
In [40]: nums, chars = [], []
   # tuple assignment
In [41]: for i in `range`(5):
   ...:     nums.append(randint(0,100))
   ...:     chars.append(uppercase))
In [42]: pairs = `sorted`(`zip`(nums, chars))
In [43]: pairs
Out[43]: [(10, 'K'), (17, 'J'), (17, 'V'), (51, 'D'), (96, 'P')]
In [44]: nlst, clst = `zip`(*pairs)
In [45]: nlst
Out[45]: (16, 17, 17, 51, 96)
In [46]: clst
Out[46]: ('K', 'J', 'V', 'D', 'P')

---

**Iteration: Best Practices**

The methods `keys`, `items`, and `values` always generate a list. For a big dictionary, this will occupy additional space in memory.

In [47]: `type`(daysOfMonth.keys())
Out[47]: list

In such cases, it's better to use the special methods: `iterkeys`, `iteritems`, and `itervalues`. They don't create an explicit list (as we can notice by using `type` to check for its output).

In [48]: `type`(daysOfMonth.iterkeys())
Out[48]: `dictionary-keyiterator`

---

**Membership in Dictionaries**

When looking up a key in a dictionary, always use the following idiom:

```python
someKey in someDict
```

and never write the following:

```python
someKey in someDict.keys()  
```

A dictionary is optimized to find a key instantly, that's a reason for using dictionaries. In a list, you have to search to be able to establish membership. For a long list, that will take time, especially if you do it within a LOOP, where this search will be repeated many times.