Dictionaries

Dictionary is typed with **braces**

Dictionary = {1,2,3,4}

List = [1,2,3,4]

Dictionaries can use integer values as keys, like this:

```
>>> spam = {12345: 'Luggage Combination', 42: 'The Answer'}
```

Dictionaries are not ordered, so they can't be sliced like lists

"There are three dictionary methods that will return list-like values of the dictionary's keys, values, or both keys and values: keys(), values(), and items()."

values()

```
>>> spam = {'color': 'red', 'age': 42}
>>> for v in spam.values():
        print(v)

red
42
```

items()

```
>>> for i in spam.items():
        print(i)

('color', 'red')
('age', 42)
```

keys()

```
>>> spam = {'color': 'red', 'age': 42}
>>> spam.keys()
dict_keys(['color', 'age'])
>>> list(spam.keys())
['color', 'age']
```

Checking whether a key or value exists in a dictionary:

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> 'name' in spam.keys()
True
>>> 'Zophie' in spam.values()
True
>>> 'color' in spam.keys()
False
>>> 'color' not in spam.keys()
True
>>> 'color' in spam
False
```

Get()

“It’s tedious to check whether a key exists in a dictionary before accessing that key’s value. Fortunately, dictionaries have a get() method that takes two arguments: the key of the value to retrieve and a fallback value to return if that key does not exist.”

```
>>> picnicItems = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnicItems.get('cups', 0)) + ' cups.'
'I am bringing 2 cups.'
>>> 'I am bringing ' + str(picnicItems.get('eggs', 0)) + ' eggs.'
'I am bringing 0 eggs.'
```

“Because there is no 'eggs' key in the picnicItems dictionary, the default value 0 is returned by the get() method. **Without using get(), the code would have caused an error message, such as in the following example:”**

```
>>> picnicItems = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'
Traceback (most recent call last):
  File "<pyshell#34>", line 1, in <module>
    'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'
KeyError: 'eggs'
```

setDefault()

You'll often have to set a value in a dictionary for a certain key only if that key does not already have a value. The code looks something like this:

```
spam = {'name': 'Pooka', 'age': 5}
if 'color' not in spam:
    spam['color'] = 'black'
```

The setdefault() method offers a way to do this in one line of code. The first argument passed to the method is the key to check for, and the second argument is the value to set at that key if the key does not exist. If the key does exist, the setdefault() method returns the key's value. Enter the following into the interactive shell:

```
>>> spam = {'name': 'Pooka', 'age': 5}
>>> spam.setdefault('color', 'black')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
>>> spam.setdefault('color', 'white')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
```

The first time setdefault() is called, the dictionary in spam changes to {'color': 'black', 'age': 5, 'name': 'Pooka'}. The method returns the value 'black' because this is now the value set for the key 'color'. When spam.setdefault('color', 'white') is called next, the value for that key is not changed to 'white' because spam already has a key named 'color'.

The setdefault() method is a nice shortcut to ensure that a key exists. Here is a short program that counts the number of occurrences of each letter in a string.

Nested dictionaries

“As you model more complicated things, you may find you need dictionaries and lists that contain other dictionaries and lists. Lists are useful to contain an ordered series of values, and dictionaries are useful for associating keys with values. For example, here’s a program that uses a dictionary that contains other dictionaries in order to see who is bringing what to a picnic. The totalBrought() function can read this data structure and calculate the total number of an item being brought by all the guests.”

```
   allGuests = {'Alice': {'apples': 5, 'pretzels': 12},
                'Bob': {'ham sandwiches': 3, 'apples': 2},
                'Carol': {'cups': 3, 'apple pies': 1}}

   def totalBrought(guests, item):
       numBrought = 0
       for k, v in guests.items():
          numBrought = numBrought + v.get(item, 0)
       return numBrought

   print('Number of things being brought:')
   print(' - Apples         ' + str(totalBrought(allGuests,
'apples')))
   print(' - Cups           ' + str(totalBrought(allGuests, 'cups')))
   print(' - Cakes          ' + str(totalBrought(allGuests,
'cakes')))
   print(' - Ham Sandwiches ' + str(totalBrought(allGuests, 'ham
sandwiches')))
   print(' - Apple Pies     ' + str(totalBrought(allGuests, 'apple
pies')))
```

‘Inside the totalBrought() function, the for loop iterates over the key-value pairs in guests ❶. Inside the loop, the string of the guest’s name is assigned to k, and the dictionary of picnic items they’re bringing is assigned to v. If the item parameter exists as a key in this dictionary, it’s value (the quantity) is added to numBrought ❷. If it does not exist as a key, the get() method returns 0 to be added to numBrought.’

The output of this program looks like this:

```
Number of things being brought:
- Apples 7
- Cups 3
- Cakes 0
- Ham Sandwiches 3
- Apple Pies     1
```

“This may seem like such a simple thing to model that you wouldn’t need to bother with writing a program to do it. But realize that this same totalBrought() function could easily handle a dictionary that contains thousands of guests, each bringing thousands of different picnic items. Then having this information in a data structure along with the totalBrought() function would save you a lot of time!

You can model things with data structures in whatever way you like, as long as the rest of the code in your program can work with the data model correctly. When you first begin programming, don’t worry so much about the “right” way to model data. As you gain more experience, you may come up with more efficient models, but the important thing is that the data model works for your program’s needs.’”