Word association football

Dictionaries

A dictionary is similar to a sequence, as it can be indexed. The difference is that, instead of being indexed by 0, 1, 2, etc.; it is indexed by *keys*. (Dictionaries cannot be sliced.) A dictionary is really **a mapping from keys to values**. Dictionary **keys** can be any **immutable type** — for example strings or numbers, while **values** can be of **any type**. Dictionaries are used for storing and retrieving information based on a key. For this reason, there can be no duplicates in the keys, but there can be duplicates in the values.

Here are some examples of dictionaries in action using a phone book as an example.

```
>>> phone = {'Eric' : 7724, 'John' : 9224, 'Graham' : 8462}
>>> phone
{'John': 9224, 'Graham': 8462, 'Eric': 7724}
>>> type(phone)
<class 'dict'>
>>> phone['John']
9224
>>> phone['Terry']

Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    phone['Terry']
KeyError: 'Terry'
>>> phone['Terry'] = 6352
>>> phone['Terry']
6352
>>> phone
{'John': 9224, 'Graham': 8462, 'Eric': 7724, 'Terry': 6352}
```

The first example shows one way of constructing a dictionary — an open brace (curly bracket) followed by a sequence of `key : value` pairs followed by a closed brace. Notice how when the dictionary is printed, it is not in the same order that it was created in. This is due to the way dictionaries are stored in memory. They are not necessarily stored in any particular order. The third example shows accessing the dictionary using the square bracket notation. The next example shows that, using the square bracket notation, if a key is not in the dictionary it will raise an error. However, if a key is not in the dictionary it can be added (or if the key is in the dictionary then its value is updated) using the square bracket notation as shown in the last example.

The next couple of examples show other methods of creating a dictionary.

```
>>> phone = dict(Eric=7724, John=9224, Graham=8462)
>>> phone
{'John': 9224, 'Graham': 8462, 'Eric': 7724}
>>> phone = dict([('Eric', 7724), ('John', 9224), ('Graham', 8462)])
>>> phone
{'John': 9224, 'Graham': 8462, 'Eric': 7724}
```

The first example shows that a dictionary can be created using the dict function with a sequence of `key=value` pairs as arguments. The second example shows that the dict function can also be used with a single sequence of `(key, value)` pairs as an argument.

The next few examples show a few methods of dictionaries as well as a few examples of for loops on dictionaries.

```
>>> for key in phone :
    print(key, phone[key])

John 9224
Graham 8462
Eric 7724
>>> phone.keys()
dict_keys(['Graham', 'John', 'Eric'])
>>> phone.items()
dict_items([('Graham', 8462), ('John', 9224), ('Eric', 7724)])
>>> for item in phone.items() :
    print(item)

('Graham', 8462)
('John', 9224)
('Eric', 7724)
```

The first example shows using a for loop directly on a dictionary. The loop variable `key` becomes each key in the dictionary in turn. This prints out all the `key, value` pairs of the dictionary. This is the most common method of looping through dictionaries. The next two examples show the `keys` and `items` dictionary methods. These two methods return a special class (similar to that of range), but as can be seen they contain a sequence of all the keys or all the `key, value` pairs that are known as items. These methods (and a similar `values` are included to provide an efficient way of getting the keys, values, or both and looping through them, as shown in the last example. They are rarely used in any other way.

Using Dictionaries

Let’s start with a simple example of a function that takes a filename and creates a dictionary where the keys are the line numbers and the values are the corresponding lines. Below is the function definition.

```
def get_lines(filename) :
    """Return a dictionary containing each line in the file as values
    and the corresponding line number as keys.

    Parameters:
        filename (str): Name, including path, of the file to be opened.

    Return:
        dict: Dictionary containing the contents of the file.

    Preconditions:
        'filename' is the name of a file that can be opened for reading.
    """
    lines = {}
    f = open(filename, 'r')
    for i, line in enumerate(f) :
        lines[i] = line.strip()
    f.close()
    return lines
```

The first line creates an empty dictionary, `lines`, for us to store our lines in. We then open the `filename` in universal read mode. Using a for loop, along with the enumerate function seen before, the index and line is easily obtained. We then use the square bracket notation to added the stripped line as the value to the dictionary, with the index (being the line number) as the key. The line was stripped using the strip method of strings, as this is more useful if we were to do anything more with this dictionary. The dictionary is then returned.

Having saved this code as get_lines.py, it can be tested. The following is a test using the text.txt file use previously.

```
>>> lines = get_lines('text.txt')
>>> lines
{0: 'Python is fun,', 1: 'it lets me play with files.',
2: 'I like playing with files,', 3: 'I can do some really fun stuff.',
4: '', 5: 'I like Python!'}
>>> lines[5]
'I like Python!'
```

Let’s now look at a slightly more complex example. This example will look at determining the frequency count of characters in a file. We will need to open the file for reading, read the contents of the file and count how many times each character appears. Dictionaries are ideal for this — we can use the characters as the keys and the character counts as the associated values. We will need to make use of the dictionary method **`get`**.

```
>>> d = {}
>>> help(d.get)
Help on built-in function get:

get(...)
    D.get(k[,d]) -> D[k] if k in D, else d.  d defaults to None.

>>> d = {"one" : 1, "three" : 3 ,"many" : 99999999999}
>>> d["one"]
1
>>> d.get("one")
1
>>> d["two"]

Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    d["two"]
KeyError: 'two'
>>> print(d.get("two"))
None
>>> d.get("two", "that number does not exist")
'that number does not exist'
```

The `get` method is similar to the square bracket notation that we have used to look up values associated with a key. The biggest difference is that `get` does not raise an error if the key is not in the dictionary as the square bracket notation does.

`get` Syntax

```
dictionary.get(key)
dictionary.get(key, d)
```

Semantics

`get` takes in a key as an argument and either returns the value in *`dictionary`* associated with that *`key`* or, if that key is not in *`dictionary`*, then `get` returns *`d`*. *`d`* is an optional argument of `get` and defaults to `None`.

Now we can have a look at our example. The following is the required function definition.

```
def freq_count(filename) :
    """Return the frequency count of characters occuring in a file.

    Parameters:
        filename (str): Name, including path, of the file to be opened.

    Return:
        dict: Frequency of each character occuring in the file.

    Preconditions:
        'filename' is the name of a file that can be opened for reading.
    """
    freq = {}
    file = open(filename, 'r')
    for line in file :
        for char in line :
            freq[char] = freq.get(char, 0) + 1
    file.close()
    return freq
```

The function starts with creating an empty dictionary, `freq`. Then the function opens the file and iterates through the lines of the file with a for loop. Inside the for loop is another (nested) for loop to iterate through the line to get each character. Inside that for loop we start with the square bracket notation for adding a key, value pair to `freq` using the `char` as the key. We then use the `get` method to look up the same key but if `char` is not in `freq` then 0 is returned. The value returned by `get` is incremented by 1, thus incrementing the number of occurrences of `char`. `get` is used as it provides the perfect base for this problem. If the `char` is in `freq` then the value (the count) is returned and then has 1 added to it and then that new value is assigned to the key. If `char` is not in `freq` then `get` returns 0 which allows us to add 1 to it as this will be the first occurrence of `char` in the file.

After saving freq.py, it can be applied to a few files that we have used in previous sections.

```
>>> freq_count("sgame.txt")
{' ': 125, '\n': 9, '1': 5, '3': 3, '2': 3, '5': 3, '4': 3, '7': 2, '6': 3,
'9': 4, '8': 2}
>>> freq_count("text.txt")
{'\n': 6, '!': 1, ' ': 19, ',': 2, '.': 2, 'I': 3, 'P': 2, 'a': 4, 'c': 1,
'e': 8, 'd': 1, 'g': 1, 'f': 6, 'i': 9, 'h': 4, 'k': 2, 'm': 2, 'l': 9, 'o': 4,
'n': 6, 'p': 2, 's': 6, 'r': 1, 'u': 3, 't': 7, 'w': 2, 'y': 5}
>>> freq_count("words.txt")
{'\n': 252, '!': 12, ' ': 943, "'": 40, ')': 16, '(': 16, '-': 7, ',': 48,
'.': 211, '2': 1, '4': 1, '?': 10, 'A': 6, 'C': 7, 'B': 6, 'E': 3, 'D': 12,
'G': 7, 'F': 30, 'I': 15, 'H': 2, 'K': 1, 'J': 1, 'M': 31, 'L': 4, 'O': 16,
'N': 10, 'P': 13, 'S': 14, 'R': 5, 'U': 3, 'T': 11, 'W': 8, 'V': 4, 'Y': 10,
'a': 203, '`': 10, 'c': 51, 'b': 40, 'e': 324, 'd': 114, 'g': 78, 'f': 48,
'i': 134, 'h': 203, 'k': 21, 'j': 3, 'm': 38, 'l': 99, 'o': 298, 'n': 179, 'q': 5,
'p': 48, 's': 155, 'r': 250, 'u': 98, 't': 294, 'w': 50, 'v': 15, 'y': 72, 'x': 1,
'z': 3}
```

Note that, instead of reading the file line-by-line, we could have read the entire file into a single string, using `read` and processed that character-by-character. However, for a very large file this approach would generate a very large (especially in memory) string.

Formatting Strings

The dictionaries, especially when large, do not print out very nicely. Let's write a function that takes a dictionary and displays it in an easy to read format (also called 'pretty printing'). To be able to 'pretty print' we need to be able to print in a formatted way. We have seen simple examples of this already by simply printing one value after another separated by commas. Python has another approach using the **`format`** method of strings. `format` operates on a format string which has segments where substitutions are made. The items substituted into this format string are the arguments of the `format` method.

Following are a few examples using the `format` method.

```
>>> help(str.format)
Help on method_descriptor:

format(...)
    S.format(*args, **kwargs) -> string

    Return a formatted version of S, using substitutions from args and kwargs.
    The substitutions are identified by braces ('{' and '}').

>>> a = 10
>>> b = 22
>>> c = 42
>>> 'a = {0}, b = {1}, c = {2}'.format(a, b, c)
'a = 10, b = 22, c = 42'
>>> 'a = {0}, b = {1}, c = {2}'.format(b, c, a)
'a = 22, b = 42, c = 10'
>>> 'a = {0}, b = {1}, c = b + 2 * a = {1} + 2 * {0} = {2}'.format(a,b,c)
'a = 10, b = 22, c = b + 2 * a = 22 + 2 * 10 = 42'
>>> s = 'hello'
>>> '{0} world, {1}'.format(s, c)
'hello world, 42'
>>> 'a = {0}, b = {1}, c = {2}'.format(b, c)

Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    'a = {0}, b = {1}, c = {2}'.format(b, c)
IndexError: tuple index out of range
>>> 'a = {0}, b = {1}, c = {2}'.format(a, b, c, s)
'a = 10, b = 22, c = 42'
>>> our_format = 'print {0} everywhere, {1}'
>>> our_format.format(a, s)
'print 10 everywhere, hello'
```

The first example is of the help description of the `format` method. The next two examples use the `format` method to print three numbers out with descriptors as to what they are. Notice how the order of the arguments of `format` is associated with the numbers in the substitutions. The next example shows that substitutions can be made more than once into the format string. The next example shows that multiple types can be printed. The next example shows that if there are more substitutions to be made than there are arguments in `format` then an error is raised. However, the next example shows that if there are more arguments in `format` than substitution areas then there is no error, the extra values are ignored as the example shows that the format string can be assigned to a variable and used later to format print.

`format` Syntax

`string``.format(`*`sequence`*`)`

Semantics

Values from *sequence* are substituted into substitution place holders in *string*. These place holders are denoted by numbers starting from 0 inside braces (curly brackets). The index of the value in *sequence* is the corresponding number that the value is substituted. The string resulting from the substitutions is returned.

Aside: More Formatting options

The `format` method, along with the format string, has many options to enable different forms of formatted printing. Following are examples of some of the possible formatting options.

```
>>> 60.0/22
2.727272727272727
>>> "2 decimal places - {0:.2f}".format(60.0/22)
'2 decimal places - 2.73'
>>> "5 decimal places - {0:.5f}".format(60.0/22)
'5 decimal places - 2.72727'
```

It is possible to print to a certain number of decimal places by using a `:.nf` after the index number. `n` here is the number of decimal places that are to be used. The examples above show printing to 2 and 5 decimal places.

```
>>> "use indexing - {0[3]}, {0[1]}".format("hello")
'use indexing - l, e'
```

Indexing can be used on sequences inside the format string by indexing the substitution placeholder. The above example uses indexing twice during formatting.

```
>>> "give spacing - {0:10}, {1:7}".format("hi", "bye")
'give spacing - hi        , bye    '
>>> "give spacing - {0:10}, {1:7}".format("longer", "bye")
'give spacing - longer    , bye    '
```

Spacing can be made around the items that are to be printed by adding a `:n` after the substitution place holder. `n` here represents the number of places that are required. The item fills up the spacing to its length then blank spaces are added to fill the rest. Above is two examples of printing with spacing around a couple of strings.

These are just a few of the possible options available for format printing. More options and examples are available at the Python docs page for format strings.

Let’s return to our example. Below is the function definition for 'pretty printing our dictionary.

```
def display_dictionary(dictionary) :
    """Pretty print 'dictionary' in key sorted order.

    Parameters:
        dictionary dict: Dictionary to be pretty printed.
    """
    keys = dictionary.keys()
    keys = sorted(keys)
    for k in keys :
        print('{0}  :  {1}'.format(repr(k), dictionary[k]))
```

The function first gets the keys of the dictionary and sorts them. The sorted function returns a sorted list of the given sequence. Now, when the dictionary is printed, there is a nice order to the characters. Then it iterates through the keys list and prints them using `format`. The keys are printed using the repr function. This makes the strings print with the ' ' around them, if we had printed the strings directly we would have lost the quotes. repr returns the **representation** of the argument. This can be seen when the disp_dict.py file is saved and tested as below.

```
>>> freq = freq_count('sgame.txt')
>>> display_dictionary(freq)
'\n'  :  9
' '  :  125
'1'  :  5
'2'  :  3
'3'  :  3
'4'  :  3
'5'  :  3
'6'  :  3
'7'  :  2
'8'  :  2
'9'  :  4
```