Experiment No. 7: Dictionaries

We've been learning about *sequences* in Python but now we're going to switch gears and learn about *mappings* in Python. This section will serve as a brief introduction to another built-in type called 'Dictionary', one of Python's best features and consist of:

1) Constructing a Dictionary
2) Accessing objects from a dictionary
3) Nesting Dictionaries
4) Basic Dictionary Methods

A dictionary is like a list, but more general. In a list, the indices have to be integers; in a dictionary they can be (almost) any type.

A dictionary contains a collection of indices, which are called **keys**, and a collection of **values**. Each key is associated with a single value. The association of a key and a value is called a *key-value pair* or sometimes an **item**.

A dictionary represents a mapping from *keys* to *values*, so you can also say that each key "maps to" a value. As an example, we'll build a dictionary that maps from English to Spanish words, so the keys and the values are all strings.

Constructing a Dictionary

Let's see how we can construct dictionaries to get a better understanding of how they work!

In [1]:
```
# Make a dictionary with {} and : to signify a key and a value
my_dict = {'key1':'value1','key2':'value2'}
```

In [2]:
```
# Call values by their key
my_dict['key2']
```

Out[2]:
```
'value2'
```

Its important to note that dictionaries are very flexible in the data types they can hold. For example:

In [3]:
```
my_dict = {'key1':123,'key2':[12,23,33],'key3':['item0','item1','item2']}
```

In [4]:
```
# Let's call items from the dictionary
my_dict['key3']
```

Out[4]:
```
['item0', 'item1', 'item2']
```

In [5]:
```
# Can call an index on that value
my_dict['key3'][0]
```

Out[5]:
```
'item0'
```

In [6]:
```
# Can then even call methods on that value
my_dict['key3'][0].upper()
```

Out[6]:
```
'ITEM0'
```

We can affect the values of a key as well. For instance:

In [7]:
```
my_dict['key1']
```

Out[7]:
```
123
```

In [8]:
```
# Subtract 123 from the value
my_dict['key1'] = my_dict['key1'] - 123
```

In [9]:
```
#Check
my_dict['key1']
```

Out[9]:
```
0
```

A quick note, Python has a built-in method of doing a self subtraction or addition (or multiplication or division). We could have also used += or -= for the above statement. For example:

In [10]:
```
# Set the object equal to itself minus 123
my_dict['key1'] -= 123
my_dict['key1']
```

Out[10]:
```
-123
```

We can also create keys by assignment. For instance if we started off with an empty dictionary, we could continually add to it:

In [11]:
```
# Create a new dictionary
d = {}
```

In [12]:
```
# Create a new key through assignment
d['animal'] = 'Dog'
```

In [13]:
```
# Can do this with any object
d['answer'] = 42
```

In [14]:
```
#Show
d
```

Out[14]:
```
{'animal': 'Dog', 'answer': 42}
```

Nesting with Dictionaries

Hopefully you're starting to see how powerful Python is with its flexibility of nesting objects and calling methods on them. Let's see a dictionary nested inside a dictionary:

In [15]:
```
# Dictionary nested inside a dictionary nested inside a dictionary
d = {'key1':{'nestkey':{'subnestkey':'value'}}}
```

Wow! That's a quite the inception of dictionaries! Let's see how we can grab that value:

In [16]:
```
# Keep calling the keys
d['key1']['nestkey']['subnestkey']
```

Out[16]:
```
'value'
```

A few Dictionary Methods

There are a few methods we can call on a dictionary. Let's get a quick introduction to a few of them:

In [17]:
```
# Create a typical dictionary
d = {'key1':1,'key2':2,'key3':3}
```

In [18]:
```
# Method to return a list of all keys
d.keys()
```

Out[18]:
```
dict_keys(['key1', 'key2', 'key3'])
```

In [19]:
```
# Method to grab all values
d.values()
```

Out[19]:
```
dict_values([1, 2, 3])
```

In [20]:
```
# Method to return tuples of all items  (we'll learn about tuples soon)
d.items()
```

Out[20]:
```
dict_items([('key1', 1), ('key2', 2), ('key3', 3)])
```

The function dict creates a new dictionary with no items. Because dict is the name of a built-in function, you should avoid using it as a variable name

In [21]:
```
eng2sp = dict()
eng2sp
```

Out[21]:
```
{}
```

The squiggly-brackets, {}, represent an empty dictionary. To add items to the dictionary, you can use square brackets:

In [22]:
```
eng2sp['one'] = 'uno'
```

This line creates an item that maps from the key 'one' to the value 'uno'. If we print the dictionary again, we see a key-value pair with a colon between the key and value:

In [23]:
```
eng2sp
```

Out[23]:
```
{'one': 'uno'}
```

This output format is also an input format. For example, you can create a new dictionary with three items:

In [24]:
```
eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

In [25]:
```
eng2sp
```

Out[25]:
```
{'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

The order of items in a dictionary is unpredictable. But that's not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

In [26]:
```
eng2sp['two']
```

Out[26]:
```
'dos'
```

The key 'two' always maps to the value 'dos' so the order of the items doesn't matter. If the key isn't in the dictionary, you get an exception:

In [27]:
```
eng2sp['four']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[27], line 1
----> 1 eng2sp['four']

KeyError: 'four'
```

The len function works on dictionaries; it returns the number of key-value pairs:

In [28]:
```
len(eng2sp)
```

Out[28]:
```
3
```

The in operator works on dictionaries, too; it tells you whether something appears as a key in the dictionary (appearing as a value is not good enough).

In [29]:
```
'one' in eng2sp
```

Out[29]: True

In [30]:
```
'uno' in eng2sp
```

Out[30]: False

To see whether something appears as a value in a dictionary, you can use the method values, which returns a collection of values, and then use the in operator:

In [31]:
```
vals = eng2sp.values()
'uno' in vals
```

Out[31]: True

Dictionary as a collection of counters

Suppose you are given a string and you want to count how many times each letter appears. There are several ways you could do it:

1. You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using a chained conditional.

2. You could create a list with 26 elements. Then you could convert each character to a number (using the built-in function ord), use the number as an index into the list, and increment the appropriate counter.

3. You could create a dictionary with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item.

Each of these options performs the same computation, but each of them implements that computation in a different way. An implementation is a way of performing a computation; some implementations are better than others. For example, an advantage of the dictionary implementation is that we don't have to know ahead of time which letters appear in the string and we only have

to make room for the letters that do appear. Here is what the code might look like:

In [32]:
```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

The name of the function is histogram, which is a statistical term for a collection of counters (or frequencies). The first line of the function creates an empty dictionary. The for loop traverses the string. Each time through the loop, if the character c is not in the dictionary, we create a new item with key c and the initial value 1 (since we have seen this letter once). If c is already in the dictionary we increment d[c]. Here's how it works:

In [33]:
```
h = histogram('brontosaurus')
h
```

Out[33]:
```
{'b': 1, 'r': 2, 'o': 2, 'n': 1, 't': 1, 's': 2, 'a': 1, 'u': 2}
```

The histogram indicates that the letters 'a' and 'b' appear once; 'o' appears twice, and so on. Dictionaries have a method called **get** that takes a key and a default value. If the key appears in the dictionary, get returns the corresponding value; otherwise it returns the default value. For example:

In [34]:
```
h = histogram('a')
h
```

Out[34]:
```
{'a': 1}
```

In [35]:
```
h.get('a', 0)
```

Out[35]:
```
1
```

In [36]:
```
h.get('c', 0)
```

Out[36]:
```
0
```

Looping and dictionaries

If you use a dictionary in a for statement, it traverses the keys of the dictionary. For example, print_hist prints each key and the corresponding value:

In [37]:
```
def print_hist(h):
    for c in h:
        print(c, h[c])
```

Here's what the output looks like:

In [38]:
```
h = histogram('parrot')
print_hist(h)
```

```
p 1
a 1
r 2
o 1
t 1
```

Again, the keys are in no particular order. To traverse the keys in sorted order, you can use the built-in function sorted:

In [39]:
```
for key in sorted(h):
    print(key, h[key])
```

```
a 1
o 1
p 1
r 2
t 1
```

Reverse lookup

Given a dictionary d and a key k, it is easy to find the corresponding value v = d[k]. This operation is called a lookup. But what if you have v and you want to find k? You have two problems: first, there might be more than one key that maps to the value v. Depending on the application, you might be able to pick one, or you might have to make a list that contains all of them. Second, there is no simple syntax to do a reverse lookup; you have to search. Here is a function that takes a value and returns the first key that maps to that value:

In [40]:
```
def reverse_lookup(d, v):
    for k in d:
        if d[k] == v:
            return k
    raise LookupError()
```

This function is yet another example of the search pattern, but it uses a feature we haven't seen before, raise. The raise statement causes an exception; in this case it causes a LookupError, which is a built-in exception used to indicate that a lookup operation failed. If we get to the end of the loop, that means v doesn't appear in the dictionary as a value, so we raise an exception. Here is an example of a successful reverse lookup:

In [41]:
```
h = histogram('parrot')
key = reverse_lookup(h, 2)
key
```

Out[41]:
```
'r'
```

And an unsuccessful one:

In [42]:
```
key = reverse_lookup(h, 3)
```

```
---------------------------------------------------------------------------
LookupError                               Traceback (most recent call last)
Cell In[42], line 1
----> 1 key = reverse_lookup(h, 3)

Cell In[40], line 5, in reverse_lookup(d, v)
      3     if d[k] == v:
      4         return k
----> 5 raise LookupError()

LookupError:
```

The effect when you raise an exception is the same as when Python raises one: it prints a traceback and an error message. When you raise an exception, you can provide a detailed error message as an optional argument. For example:

In [43]:
```
raise LookupError('value does not appear in the dictionary')
```

```
---------------------------------------------------------------------------
LookupError                               Traceback (most recent call last)
Cell In[43], line 1
----> 1 raise LookupError('value does not appear in the dictionary')

LookupError: value does not appear in the dictionary
```

A reverse lookup is much slower than a forward lookup; if you have to do it often, or if the dictionary gets big, the performance of your program will suffer.

Dictionaries and lists

Lists can appear as values in a dictionary. For example, if you are given a dictionary that maps from letters to frequencies, you might want to invert it; that is, create a dictionary that maps from frequencies to letters. Since there might be several letters with the same frequency, each value in the inverted dictionary should be a list of letters. Here is a function that inverts a dictionary:

In [44]:
```
def invert_dict(d):
    inverse = dict()
    for key in d:
        val = d[key]
        if val not in inverse:
            inverse[val] = [key]
        else:
            inverse[val].append(key)
    return inverse
```

Each time through the loop, key gets a key from d and val gets the corresponding value. If val is not in inverse, that means we haven't seen it before, so we create a new item and initialize it with a singleton (a list that contains a single element). Otherwise we have seen this value before, so we append the corresponding key to the list. Here is an example:

In [45]:
```
hist = histogram('parrot')
hist
```

Out[45]:
```
{'p': 1, 'a': 1, 'r': 2, 'o': 1, 't': 1}
```

In [46]:
```
inverse = invert_dict(hist)
inverse
```

Out[46]:
```
{1: ['p', 'a', 'o', 't'], 2: ['r']}
```

Lists can be values in a dictionary, as this example shows, but they cannot be keys. Here's what happens if you try:

In [47]:
```
t = [1, 2, 3]
d = dict()
d[t] = 'oops'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[47], line 3
      1 t = [1, 2, 3]
      2 d = dict()
----> 3 d[t] = 'oops'

TypeError: unhashable type: 'list'
```

A dictionary is implemented using a hashtable and that means that the keys have to be hashable.A hash is a function that takes a value (of any kind) and returns an integer. Dictionaries use these integers, called hash values, to store and look up key-value pairs. This system works fine if the keys are immutable. But if the keys are mutable, like lists, bad things happen. For example, when you create a key-value pair, Python hashes the key and stores it in the corresponding location. If you modify the

key and then hash it again, it would go to a different location. In that case you might have two entries for the same key, or you might not be able to find a key. Either way, the dictionary wouldn't work correctly. That's why keys have to be hashable, and why mutable types like lists aren't. The simplest way to get around this limitation is to use tuples, which we will see in the next experiment. Since dictionaries are mutable, they can't be used as keys, but they can be used as values.

Memos

If you played with the fibonacci function, you might have noticed that the bigger the argument you provide, the longer the function takes to run. Furthermore, the run time increases quickly. fibonacci with n=4 calls fibonacci with n=3 and n=2. In turn, fibonacci with n=3 calls fibonacci with n=2 and n=1. And so on.

Count how many times fibonacci(0) and fibonacci(1) are called. This is an inefficient solution to the problem, and it gets worse as the argument gets bigger. One solution is to keep track of values that have already been computed by storing them in a dictionary. A previously computed value that is stored for later use is called a memo.

Here is a "memoized" version of fibonacci:

In [48]:
```
known = {0:0, 1:1}
def fibonacci(n):
    if n in known:
        return known[n]
    res = fibonacci(n-1) + fibonacci(n-2)
    known[n] = res
    return res
```

known is a dictionary that keeps track of the Fibonacci numbers we already know. It starts with two items: 0 maps to 0 and 1 maps to 1. Whenever fibonacci is called, it checks known. If the result is already there, it can return immediately. Otherwise it has to compute the new value, add it to the dictionary, and return it. If you run this version of fibonacci and compare it with the original, you will find that it is much faster.