Bonus: Dictionaries and Sets

Our video series covered the basic programming topics you need to get started writing Python programs for scientific and engineering applications, but there are many more aspects of Python we left untouched. The most important built-in data structure we did not cover is the *dictionary*. The *set* is another useful datatype often used in conjunction with dictionaries.

A dictionary is similar to a list, but a list is *ordered* and a dictionary is *unordered*. We cannot use an integer to identify an element of a dictionary; instead we use a *key*. A key may be of any immutable type. Thus a tuple is allowed as a key, but whereas a general tuple can contain elements that are mutable, such as a list, any tuple used as a dictionary key may not contain any mutable elements.

The values can be of any type, mutable or immutable. The values can even be another dictionary, giving us a *nested* dictionary. Values can be repeated, but keys *must* be unique.

The dictionary itself is a mutable type, and can be of any size up to the limit of the computer's memory. We designate dictionaries with curly braces `{}` and we can create an empty dictionary with a set of empty braces:

```
words={}
```

We can also create a dictionary with a group of explicit key-value pairs:

```
Animals={'bear':'panda', 'cat':"leopard", "dog":"wolf"}
```

We refer to values in the dictionary with a syntax similar to that of lists, but using the key instead of an index.

```
In  [1]: Animals['bear']
Out [1]: 'panda'
```

To add elements we simply add the key and assign a value to it:

```
Animals['parrot']='African grey'
```

Operations on dictionaries include deleting key-value pairs

```
del Animals['bear']
```

Deleting an entire dictionary

```
del Animals
```

Clearing the dictionary (preserving the name but removing all content)

```
Animals.clear()
```

The length of a dictionary is the number of key-value pairs (i.e. the number of keys)

```
len(Animals)
```

A list of keys can be returned with

```
keys_list=Animals.keys()
```

Note: in Python 3 this must be explicitly converted if you need the list itself, rather than just an iterable:

```
keys_list=list(Animals.keys())    #Python 3
```

The `in` operator tests whether a key is present in a dictionary:

```
In  [2]: 'dog' in Animals
Out [2]: True
```

If you attempt to access a key not in the dictionary, a `KeyError` exception is thrown.

```
In  [3]: Animals['ungulate']
KeyError                                  Traceback (most recent
call last)
<ipython-input-3-d5f25ca351a4> in <module>()
----> 1 Animals['ungulate']

KeyError: 'ungulate'
```

If you need to access a key but do not know it's in the dictionary, use
`D.get(key)`
This returns `None` if the key is not found. It can also accept an optional argument to return a number if the key is absent

```
D.get(key,0)
```

You can also use `in` and `not in`

```
if this_key in D:
if this_key not in D:
```

Exercise
Type into Spyder and run

```
capitals={"Alabama":"Montgomery"}
capitals["Alaska"]="Juneau"
capitals["Arizona"]="Phoenix"
```

```
capitals["Arkansas"]="Little Rock"
print capitals.keys()
print "Virginia" in capitals
print "Arkansas" in capitals
```

Now add

```
newstate="Connecticut"
newcapital="Hartford"
if newcapital not in capitals:
  capitals[newstate]=newcapital
for key in capitals:
  print "The capital of ", key,\
        "is ",capitals[key]
```

We can iterate over the list of keys

```
for k in D.keys():
```

or just

```
for k in D:
```

Iterating over values is more difficult, since dictionaries are inherently unordered. We must convert it to an *iterable*. In Python 2.7 we can use

```
for k,v in D.iteritems():
    print k, v
```

In both Python 2.7 and 3 we can use

```
for k,v in D.items():
    print k, v
```

though this may be slow in 2.7.

Uses for Dictionaries

The dictionary is a very powerful data structure, but its applications in scientific and engineering codes may not be immediately obvious, especially to a beginner. Lists and arrays often feel more natural to students, since they may seem to be more "mathy." But there are at least two situations where dictionaries are a good choice of data structure. The first is when we need to find values by specific keys quickly. If you know the key there is no need to loop through the entire dictionary.

Wrong:

```
for k in capitals:
    if capitals[k]=='Arizona'
       print capitals[k]
```

Right:

```
print capitals['Arizona']
```

More generally, wherever it is more natural to retrieve a value from a non-integer reference, a dictionary is the appropriate data structure. For example, suppose you have a DNA sequence and you wish to count the number of occurrences of each nucleotide. You could set up a list
`nucleotides=['T','A','G','C']`
You would need a corresponding list
`n_count=[0,0,0,0]`
Then when writing your code, you would have to keep track of the count by index:

```
for nucleotide in sequence  #assume sequence is the string
    for i in range(3):
        if nucleotide==nucleotides[i]:
        n_count[i]+=1
```

We'd then have to remember the correct order of the nucleotides list to do further processing.

Using a dictionary we would simply write

```
nucleotides={'T':0,'A':0,'G':0,'C':0}
for nucleotide in sequence:
    if nucleotide in nucleotides:
        nucleotides[nucleotide]+=1
    else:
        print "Illegal nucleotide symbol encountered"
```

The second version is simpler, easier to manage, and likely faster.

Sets

Sets are another unordered type. This data structure is designed to have properties similar to its mathematical namesake. Sets are mutable but all elements must be immutable. The elements of a set must be unique; none may be duplicated. A set can be created with the `set()` function, but it can take at most one item, so a tuple is allowed:

```
myset=set((2,3,4,5))

empty_set=set()
```

We can also create a set by enclosing the elements in curly braces; this is similar to a dictionary but without any keys.

```
myset={2,3,4,5}
```

We add an element to a set with `add`

```
myset.add(6)
```

We extend it with a sequence using `update`
```
myset.update(7,8,9)
```

The `discard` method will fail silently if the item isn't present:

```
myset.discard(8)
myset.discard(11)
```

Whereas remove will throw an exception if the item isn't in the set
```
myset.remove(7)
myset.remove(12)
```

Clearing removes all elements
```
myset.clear()
```

Despite sets being unordered, the in operator can test for membership

```
if item in myset:
```

Sets have methods defined on them to imitate mathematical operations on sets.

Booleans:
`s2.issubset(s1)` or `s2<=s1` #True if s2 is a subset, or the same as, s1
`s2.issuperset(s1)` or `s2>=s1` #True if s2 is a superset, or the same as, s1
`s1==s2` #True if s1 has the same elements as s2

Information:
Len(s) #number of elements
Max(s) #maximum element
Min(s) #minimum element

Create new sets:
Intersection
`s1.intersection(s2)` or `s1&s2`
Union
`s1.union(s2)` or `s1|s2`
Symmetric difference, i.e. elements in `s1` or `s2` but not both

`s1.symmetric_difference(s2)` or `s1^s2`
Difference, i.e. elements in `s1` but not in `s2`
`s1.difference(s2)` or `s1-s2`

One common use of sets is to remove duplicates from a list:

```
nodupes=list(set(mylist))
```

This will result in a loss of order, however.

```
>>>L=[0,0,1,4,8,8,10]
>>>M=list(set(L))
>>>print M
[0, 1, 10, 4, 8]
```

Exercise

Type at your interpreter (no prompt shown here)

```
s=set()
s.update("California")
print s
```

What happened?? Lesson: be careful with strings since they are sequences.

```
s1={"Alabama","Arkansas","California","California"}
print s1
s2=set()
s2.add("California")
s2.add("Colorado")
s2.add("Oregon")
s1-s2
s1^s2
s1&s2
s1|s2
```