Data Structures

First we covered the basics of Python and what it means to have modules in files, run scripts from the command line, and what basic Python sytax looks like.

We then covered primitives like numerics, booleans, `None`s, and strings. So let's now jump into more complex data structures that will help us organize and process information.

- Lists
- Sets
- Dictionaries
- Tuples

Lists

Probably the most used data structure in Python. You'll want to pay attention to this section. Lists in Python support:

- list.**append**(x) - adding item to list
- list.**extend**(L) - adding lists together
- list.**insert**(i, x) - insert item at given position
- list.**remove**(x) - search and remove
- list.**pop**([i]) - search, return, and remove
- list.**index**(x) - search and return position
- list.**sort**(cmp=None, key=None, reverse=False) - sort list
- list.**reverse**() - reverse list

Declaring a New (Empty) List

In [ ]:
```
# to create an empty list
items1 = []
items2 = list()

# prove they are the same
print items1 == items2
print items1
```

Declaring a New List (with Elements)

```
In [ ]: my_items = [0, 1, 2, 3]

        # count number of items in list
        print "Number of elements in this list:", len(my_items)

        # iterate through and print each element
        for item in my_items:
            print item

        # we can actually delcare this list another way! ...
        print "Same list, using range():", range(4)

        # an idiom you'll use ALL the time
        for n in range(4):
            print n
```

Deleting a List with the `del` Keyword

```
In [141]: my_list = ['a', 'b', 'c', None]
          print "Still here:", my_list

          # delete it
          del my_list
          print "Should trigger a NameError:", my_list
```

```
Still here: ['a', 'b', 'c', None]
Should trigger a NameError:

-------------------------------------------------------------------
---------
NameError                                 Traceback (most recent c
all last)
<ipython-input-141-9ceaaeadbec2> in <module>()
      4 # delete it
      5 del my_list
----> 6 print "Should trigger a NameError:", my_list

NameError: name 'my_list' is not defined
```

Adding Items with `append()` and `insert()`

In [ ]:
```
my_list = ['a', 'b', 'c']

# add an element to the end of the list
my_list.append('end')

# add an element to the beginning of the list
my_list.insert(0, 'beginning')

print my_list
```

Combining Two Lists

In [ ]:
```
# create a couple lists
a = range(3)
b = ['blue', 'red', 'orange']

# add all the elements of list `b` TO list `a`
a.extend(b)
print "lists a & b:", a
```

Accessing Elements

We can access elements in a list either by:

- Index (0-based)
- Iteration (in a `for` loop)

In [143]:
```
my_list = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
print "The 5th element is:", my_list[4]

# note in Python, -1 is shorthand for len(list) - 1
print "Last element, long way:", my_list[len(my_list) - 1]

# the easier way
print "Last element, easy way:", my_list[-1]
```

```
The 5th element is: e
Last element, long way: g
Last element, easy way: g
```

Accessing a Range of Elements (Slicing)

Slicing in Python works like this:

```
some_list [ <first=0> : <last=-1> : <stepsize=1> ]
```

with the defaults shown above.

In [149]:
```
# get range of elements
my_list = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
print "The 3rd through 6th characters, inclusive are:", my_list[2:
6:1]

# first five elements
print "First five elements:", my_list[:5]
print my_list[:5] == my_list[0:5] == my_list[0:5:1]  # any of these
work

# How can we get every even number between 0 and 20? HINT: think ra
nge() function!
# ...
```

```
The 3rd through 6th characters, inclusive are: ['c', 'd', 'e',
'f']
First five elements: ['a', 'b', 'c', 'd', 'e']
True
```

List Comprensions

Alternate way of constructing lists in a very compact way without using for loops:

In [ ]:
```
squares_verbose = []
n = 5

# the verbose way
for i in range(n):
    squares_verbose.append(i**2)

# the list comprehension way
squares_short = [x**2 for x in range(n)]

# verify the same
print squares_short
print squares_verbose == squares_short
```

Removing items

Options:

- **.remove(x)**
- **del[index]**

In [ ]:
```
colors = ['red', 'blue', 'orange', 'yellow']
colors.remove('blue')
del colors[0]

# now it's gone!
print colors

# but be careful...
duplicates = ['red', 'red', 'blue', 'blue']
duplicates.remove('blue')

# only removes the first instance found!
print duplicates
```

Removing Multiple Items (Filtering)

There's a better way than calling **remove()** repeatedly. List comprehensions take the form:

```
[<value-to-keep> for <temp-var> in <iterable> if <condition>]
```

Example:

In [ ]:
```
duplicates = ['red', 'red', 'blue', 'blue', 'green', 'green', 'yell
ow']

# filter the list to exclude 'blue' and 'green' - and let's format
it to make it more clear
no_blues = [
    color for color in duplicates
        if color != 'blue' and color != 'green']

# check
print no_blues
```

Searching with `index()` and `count()`

In [ ]:
```
letters = ['z','b','c','d','f', 'c', 'c']

# find index of FIRST occrence of element
print "c is at index:", letters.index('c')

# get count of number of 'c'
print "Letter 'c' occurs this many times:", letters.count('c')
```

Simple Sorting with `sort()`

We'll talk more about sorting objects or other data structures with many fields later on.

In [ ]:
```
numbers = [8, 5, 6, 2, 9, 11, 4]
print "Unsorted:", numbers

# ascending
numbers.sort()  # ascending
print "Ascending:", numbers

# reverse
numbers = [8, 5, 6, 2, 9, 11, 4]
numbers.sort(reverse=True)
print "Descending:", numbers
```

Reversing a List

In [ ]:
```
names_in_line = ['Bob', 'Amy', 'Sally', 'Jose']
print "Original:", names_in_line

# reverse the easy way
names_in_line.reverse()
print "Reversed:", names_in_line
```

Copying and Mutability

Since we started we've only been considering **immutable** types like numbers, booleans, and strings.

Lists, however are **mutable**, which means that they are passed *somewhat* like references:

In [ ]:
```
def add_apple(fruits):
    fruits.append('apple')
    return fruits

original_fruits = ['orange', 'pear', 'kiwi']
new_fruits = add_apple(original_fruits)

# we modified the original list!
print new_fruits
print new_fruits == original_fruits
```

Copying, the right way(s)

In [ ]:
```
original_letters = ['a', 'b', 'c', 'd', 'e']

# two methods
copy1 = list(original_letters)
copy2 = original_letters[:]

# the contents are the same
print "Copy1:", copy1
print "Copy2:", copy2
print "Contents are all the same:", original_letters == copy1 == co
py2

# but they aren't the same object
print original_letters is copy1
print original_letters is copy2
```

Sets:

Like Lists, except...

- Contain only distinct items
- Used to test membership (have we seen this item before?)
- Can perform a number of useful mathematical set-operations

Set Syntax

In [3]:
```
empty = set()

# seed with values
names_seen = set(['Sonny', 'Dillion', 'Wesley'])

if 'Sonny' in names_seen:
    print "Seen Sonny before!"

if 'John' in names_seen:
    print "Seen John before!"

# add a new name
names_seen.add('Will')
print 'Will' in names_seen

# remove a name
names_seen.remove('Will')
print 'Will' in names_seen
```

```
Seen Sonny before!
True
False
```

Iteration with Sets

We can't map to values, only iterate through the items contained inside.

In [81]:
```
names_seen = set(['Sonny', 'Dillion', 'Wesley'])

for name in names_seen:
    print "We've seen", name
```

```
We've seen Wesley
We've seen Sonny
We've seen Dillion
```

Why use a set?

Remove duplicates!

In [4]:
```
list_with_duplicates = ['red', 'blue', 'red', 'orange', 'yellow',
'purple']

# remove duplicates
uniques = set(list_with_duplicates)

for unique_name in uniques:
    print "Only single case of", unique_name
```

```
Only single case of blue
Only single case of orange
Only single case of purple
Only single case of yellow
Only single case of red
```

Why use a set? (Pt. 2)

Set operations:

- Union
- Sub/super set testing
- Intersection
- Difference

Set Operations

In [93]:
```
# shapes
polygons = set(['octogon', 'square', 'rectangle', 'triangle', 'rhom
bus', 'trapezoid'])
quadrilaterals = set(['square', 'rectangle', 'rhombus', 'trapezoi
d'])
rectangles = set(['square', 'rectangle'])
squares = set(['square'])
triangles = set(['triangle'])

# 1) Union: All 3 and 4 sided shapes
all_shapes = quadrilaterals.union(triangles)
print "All shapes:", all_shapes

# 2) Intersection: Quadrilaterals that are also triangles
triangles_and_quads = triangles.intersection(quadrilaterals)
print "Shapes with BOTH 3 and 4 sides only:", triangles_and_quads

# 3) Subset: Are quadrilaterals a subset of polygons?
print "Quads are subset of polygons?", quadrilaterals.issubset(poly
gons)

# 4) Difference: Polygons that are NOT four-sided
non_four_sided_polys = polygons.difference(quadrilaterals)
print "Polygons, not 4-sided:", non_four_sided_polys
```

```
All shapes: set(['trapezoid', 'square', 'triangle', 'rectangle',
'rhombus'])
Shapes with BOTH 3 and 4 sides only: set([])
Quads are subset of polygons? True
Polygons, not 4-sided: set(['octogon', 'triangle'])
```

Tuples

- Immutable
- Sequence of any fixed length
- Used for packaging (perhaps) heterogenous items together
- Are cheaper than objects (performance-wise)

In [ ]:
```
bob_vehicles = ('car', 'bike', 'truck')
alice_vehicles = ('car', 'boat')

# indexing / iteration
```

Tuples are Immutable

In [105]:
```
bob_vehicles = ('car', 'bike', 'truck')

# does this work?
# del bob_vehicles[1]

# what about this?
# bob_vehicles[1] = 'yacht'
```

```
-------------------------------------------------------------------
---------
TypeError                                 Traceback (most recent c
all last)
<ipython-input-105-c7cb6d7985a1> in <module>()
      5
      6 # what about this?
----> 7 bob_vehicles[1] = 'yacht'

TypeError: 'tuple' object does not support item assignment
```

Tuples are Iterable

In [107]:
```
bob_vehicles = ('car', 'bike', 'truck')

for vehicle in bob_vehicles:
    print vehicle
```

```
car
bike
truck
```

Tuple Unpacking (Same Length Tuples)

In [118]:
```
boston = ('Boston', 'MA')
lexington = ('Lexington', 'MA')

# single unpacking
city, state = boston
print city, "is in", state

# let's unpack in a for loop
city_to_state = [boston, lexington]

for city, state in city_to_state:
    print city, "is in", state
```

```
Boston is in MA
Boston is in MA
Lexington is in MA
```

Dictionaries

If Lists are the most used data structure, dictionaries are probably the most *useful* data structure.

Dictionaries allow you to map any immutable key to any value.

In [122]:
```
# create a dictionary
city_to_state = {
    'Boston' : 'MA',
    'Lexington' : 'MA',
    'Los Angeles' : 'CA',
    'London' : None,
    'Kansas City' : ('MI', 'KS'),
}
print city_to_state

# access value at key 'Boston'
print "Boston is in:", city_to_state['Boston']

# delete mapping
del city_to_state['Boston']
```

```
{'Boston': 'MA', 'London': None, 'Lexington': 'MA', 'Los Angeles':
'CA', 'Kansas City': ('MI', 'KS')}
Boston is in: MA
```

Deleting Values

In [137]:
```
city_to_state = {
    'Boston' : 'MA',
    'Lexington' : 'MA',
    'Los Angeles' : 'CA',
    'London' : None,
    'Kansas City' : ('MI', 'KS'),}
print city_to_state

# delete mapping
del city_to_state['Boston']
print city_to_state

# try to access it (Exception!)
print city_to_state['Boston']
```

```
{'Boston': 'MA', 'London': None, 'Lexington': 'MA', 'Los Angeles':
'CA', 'Kansas City': ('MI', 'KS')}
{'London': None, 'Lexington': 'MA', 'Los Angeles': 'CA', 'Kansas C
ity': ('MI', 'KS')}

------------------------------------------------------------------
---------
KeyError                                  Traceback (most recent c
all last)
<ipython-input-137-6c34730912cc> in <module>()
     12
     13 # try to access it (Exception!)
---> 14 print city_to_state['Boston']
     15
     16 # a safer way!

KeyError: 'Boston'
```

Using `get()` for safer access

What happens when you're not sure if the key will be mapped already?

In [140]:
```
city_to_state = {
    'Boston' : 'MA',
    'Lexington' : 'MA',
    'Los Angeles' : 'CA',
    'London' : None,
    'Kansas City' : ('MI', 'KS'),}

# this will throw a KeyError (more on Exceptions later)
print city_to_state['Boston']
del city_to_state['Boston']

# a safer way!
print city_to_state.get('Boston', None)
```

```
MA
None
```

Tuples -> Dictionary

In [121]:
```
mappings = [
    ('Boston', 'MA'),
    ('Lexington', 'MA'),
    ('Los Angeles', 'CA'),
    ('London', None),
    ('Kansas City', ('MI', 'KS')),
]

city_to_state = dict(mappings)  # alternate / copy constructor
print city_to_state
```

```
{'Boston': 'MA', 'London': None, 'Lexington': 'MA', 'Los Angeles':
'CA', 'Kansas City': ('MI', 'KS')}
```

Iterating through a Dictionary

In [102]:

```python
# create a dictionary
city_to_state = {
    'Boston' : 'MA',
    'Lexington' : 'MA',
    'Los Angeles' : 'CA',
    'London' : None,
    'Kansas City' : ['MI', 'KS'],
}

# iterate through key, value pairs
for city, state in city_to_state.iteritems():
    if state != None:
        print city, "is in", state
```

```
Boston is in MA
Lexington is in MA
Los Angeles is in CA
Kansas City is in ['MI', 'KS']
```

Updating a Dictionary

In [125]:

```python
existing_dict = {
    'red' : 'fish',
    'blue' : 'fish',
    'green' : 'water',}
print existing_dict

# another dict
new_dict = {'purple' : 'elephant'}

# update
existing_dict.update(new_dict)
print existing_dict
```

```
{'blue': 'fish', 'green': 'water', 'red': 'fish'}
{'blue': 'fish', 'purple': 'elephant', 'green': 'water', 'red': 'f
ish'}
```

Convience Functions

In [129]:
```
city_to_state = {
    'Boston' : 'MA',
    'Lexington' : 'MA',
    'Los Angeles' : 'CA',
    'London' : None,
    'Kansas City' : ('MI', 'KS'),
}

# keys
print city_to_state.keys()

# values
print city_to_state.values()

# number of key-value pairs
print len(city_to_state)
```

```
['Boston', 'London', 'Lexington', 'Los Angeles', 'Kansas City']
['MA', None, 'MA', 'CA', ('MI', 'KS')]
5
```

Lab: Coding Excercises

Fill in the method definitions in the file `excercises/data_structures.py`.

Make sure you can pass tests with:

```
$ py.test tests/test_syntax.py::SyntaxExcercises::<function_name>  # te
st single function
$ py.test tests/test_syntax.py::SyntaxExcercises                   # te
st all at once
```

Conclusion

- **Lists**
  - `append`, `extend`, `len()`
  - Iteration
  - Slicing `[start:end:step]`
- **Sets**
  - Adding, removing elements
  - Iteration
  - Membership testing
  - Set operations
- **Tuples**
  - Immutable
  - Heterogenous types
  - Unpacking
- **Dictionaries**
  - Adding, removing key/values
  - Iteration
  - Updating
  - Convience functions