Data Structures

Introduction to Data Science with Python

Python Data Structures

- **We have already explored basic data types. Now we will look at more complex types Python uses**
- Tuple
  - Immutable, fixed length sequence
  - Note that a “string” is a tuple of characters to Python
- Lists
  - Mutable sequence of objects
- Dicts (dictionary)
  - List of objects with a "key"
- Sets
  - Unique collection of objects

Tuples

- Immutable, fixed length
- Created by comma separated values, alternatively ()

```
# Tuple
tup1 = 3,4,5,6
tup2 = (3,4,5,6)
tup1 == tup2
# Immutable, raises error
tup1[1] = 10
>> TypeError: 'tuple' object does not support
item assignment
```

Type Conversion: Tuple

- We can convert any iterable object to a tuple with the tuple() function
  - If we call this on a string, what do you expect to happen?
  - Are the two calls below identical?

```
# type conversion
b = ("long string!")
print(b)
b = tuple("long string!")
print(b)
```

Tuples (cont)

- Tuples can be concatenated with "+"
- Tuples can be nested
- We index tuples as we have other objects

  Use [] Counting starts at 0

```
# Tuple functions
tup1 = 1,2
tup2 = 3,4
print(tup1 + tup2)

# Nesting
nest = (("the","first"),("the","second"))
nest[0]
nest[0][0]
nest[0][0][2]
```

Tuples Unpacking

- A "pythonic" thing

  Tuples will "unpack" themselves upon assignment if possible

```
# Unpacking
a, b = nest
print(a)
>> ('the', 'first')


(a1,a2),(b1,b2) = nest
print(a1)
>> the
```

Lists

- Mutable, variable-length collection of objects
- Use [] , or list() to create a list

```
# list
list1 = [3,4,5,6]
# mutable
list1[2] = 10
print(list1)
>> [3, 4, 10, 6]
# variable length
list1.append(36)
print(list1)
>> [3, 4, 10, 6, 36]
```

List Functions

- We can append, insert, remove list elements

```
# List functions
list2 = ["a","b","c"]
list2.append("d")
list2.insert(1,"a.5")
list2.remove("a.5")
list2.append(["d","e","f"])

# Didn't work as expected
help(list)
list2.extend(["d","e","f"])
```

Tuples Versus Lists

- Tuples () are immutable, fixed length
- Lists [] are the opposite
- This is the main functional difference.

- Use tuples when you want implicit read only protection on the data, implied constant
- Tuples are faster to iterate through
  - Q: How would you prove that statement??
  - A: Use the magic command %time or %timeit

Slicing (Indexing)

- We have seen this a few times, let's formally address it.
- We can slice (index) sequence like objects similarly
  - Use []. Python starts counting at 0, so var[0] is the first element
  - We can get a range (a slice) using ":" var[0:2]
  - Slices are start *inclusive*, finish *exclusive*

```
# slicing
seq1 = list(range(1,11))
>> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
seq1[0:2]
seq1[2:0] #Fail. Why?
seq1[2:]
seq1[:8]
seq1[7:8]
```

Slicing (cont)

- The ":" has three arguments
  - *Start* : *stop* : *increment*
  - Start/Stop can be negative, implying count from the end
  - Increment default value is 1, but can be any integer (including negative)

```
# Slice stepping
seq1[-1]

seq1[:9:2]

seq1[9:0:-2]

seq1[-2:-5]

seq1[-2:-5:-1]

# How would you Reverse the entire sequence?
seq1[2:20] # No Error

seq1[20] # Error
```

Negative Indexes

- A way to look at negative indexes is to think of the list as being recycled

```
# how you look at indexes when using negatives
list = [0, 1, 2, 3, 4, 5]

list

list[0:6]

list[-1:-7:-1]

-6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5  # the index

0    1   2   3   4   5  0  1  2  3  4  5  # the list value
```

Test Your Knowledge – Ex 4.1

- Now it's your turn to apply what you have learned.

1. Create a list of five elements from the numbers 1 through 5
2. Create a list of five elements from the letters a through e
3. Update the first three numbers of the first list to the the last three letters of the second list

Dictionaries (Dicts)

- "Mapping Object", maps values to objects
- Mutable, collection of objects with keys

  Created with curly braces {}, or with dict()

```
# Ways to Create Dicts

dict1 = dict(first="a", second="b", third="c") dict2 =
dict(zip(['first', 'second', 'last'], ["a", "b", "c"]))

dict3 = {'first': "a", 'second': "b", ‘third': "c"}

dict4 = dict([('second', "b"), ('first', "a"),
      (‘third', "c")])

dict5 = dict({‘third': "c", 'first': "a", 'second':
      "b"})

dict1 == dict2 == dict3 == dict4 == dict5
```

Dict (cont)

- Slicing refers to the ID

  Useful functions:

  del (keyword) , update() (method)

```
# Dict Functions
dict1["first"]
dict1[1] # gives error
dict1[1] = "update"
dict1[1]
# Remove element
del dict1[1]
# Update element
dict1.update({'first':'changed'})
```

Sets

- A set in Python is an unordered collection of unique objects
  - Sets are mutable (though objects within them are immutable)
  - Sets do not contain any duplicate members, so duplicates are ignored when creating a set

```
# Creating a set from a char tuple

x = set("A string in Python")

x


>> {' ', 'A', 'P', 'g', 'h', 'i', 'n', 'o', 'r', 's', 't', 'y'}
```

Reference vs Value

- Pass by value

  Simple variables are passed by value, which is a complete copy of the original variable. Any changes made to the copy do not affect the original

```
# Pass by value
var1 = 'foo'
var2 = var1
var2 = 'bar'
print(var1)
>> 'foo'
print(var2)
>> 'bar'
```

Reference vs Value

- Pass by reference

  Complex variables are references (pointers) to objects. Faster, but downside is multiple access points to same object

```
# Pass by Reference
var1 = list(range(3))
var2 = var1
var1.append(10)
print(var1)
>> [0, 1, 2, 10]
print(var2)

>> [0, 1, 2, 10]
```

The copy Method

- Use the copy method to force Python to pass a variable by value

```
# Use of copy method
var1 = list(range(3))

var2 = var1.copy()

var1.append(10)

print(var1)

print(var2)
```

Deep vs Shallow Copy

- Use the copy method to force a pass by value
- What about nested objects?
  - By default you have a "shallow" copy
  - Use the deepcopy method to create a completely independent copy

```
import copy
var1 = [['a', 'b', 'c'],[1, 2, 3]]
var2 = copy.deepcopy(var1)
var1[1].append(10)
print(var1)
print(var2)
```

Test Your Knowledge – Ex 4.2

- Input your first name into a variable
- Input your last name into a variable
- Create a list with your first name and last name
- Create a copy of this list with the first and last names reversed
- Create a copy of the original list with the characters of the first and last names reversed (can you do it without using a loop?)
- Create a dictionary using the reversed characters as the value and the original names as the key