Lesson D1 – Data Analysis Introduction

In this module, we will talk about some important data types and libraries for doing data analysis using Python.

- Review Python built-in data types:
  - lists
  - tuples
  - dictionaries
- Collections library:
  - defaultdict and counter
- Data exchange formats:
  - JSON

Outline for this module:

- Data analysis introduction
  - Readings
    - PDA Ch 1
    - PDA Ch 3
  - Topics
    - data structures
      - built-in Python data types: lists, tuples, dictionaries
      - Collections library: defaultdict
  - Learning goals
    - Be able to explain the differences between how lists, tuples, dictionaries, and sets work in Python and in what situations to use each
    - Gain understanding and experience with data analysis operations in Python.
    - Be able to read and process data in JSON format.
    - Become familiar with defaultdict from the Collections library.
  - Exercises
    - Exercise D1.1: Use defaultdict and collections.Counter to count data
    - Exercise D1.2: Read, manipulate, and extract data from a JSON data set

---

Lists

In Python, a list is a sequential collection of data

- Lists are a built-in data type in Python

- Each value is identified by an index
- Values are called the elements of the list
- Elements can be any data type
- Elements in a list can be of different data types

- Items in a list can be accessed using the bracket notation

In [1]:
```
a = [10, 20, 30]
b = ["unc", "duke", "ncstate"]
c = ["luke", 3.14, [10, 20], 50]
print(a)
print(b)
print(b[0])
print(b[1])
print(c[2][1])
```

```
[10, 20, 30]
['unc', 'duke', 'ncstate']
unc
duke
20
```

List slices

- For many ordered collection objects (including lists), Python supports 'slicing':

In [2]:
```
acc_costal = ["duke", "gatech", "miami", "unc", "pitt", "uva", "vatech"]
print (acc_costal[2:4])
print (acc_costal[1:])
print (acc_costal[:3])
```

```
['miami', 'unc']
['gatech', 'miami', 'unc', 'pitt', 'uva', 'vatech']
['duke', 'gatech', 'miami']
```

Lists are mutable

- Can change a list as part of an LHS assignment:

In [3]:
```
acc_costal = ["duke", "gatech", "miami", "unc", "pitt", "uva", "vatech"]
print (acc_costal)
acc_costal[6] = "vt"
print(acc_costal)
```

```
['duke', 'gatech', 'miami', 'unc', 'pitt', 'uva', 'vatech']
['duke', 'gatech', 'miami', 'unc', 'pitt', 'uva', 'vt']
```

Lists methods

- Lists are Python objects with many built-in methods:

Method	Parameters	Result	Description

append | item |mutator |Adds a new item to the end of a list insert |position, item |mutator |Inserts a new item at the position given pop |none |hybrid |Removes and returns the last item pop |position |hybrid |Removes and returns the item at position sort |none |mutator |Modifies a list to be sorted reverse |none |mutator |Modifies a list to be in reverse order index |item |return idx |Returns the position of first occurrence of item count |item |return ct |Returns the number of occurrences of item remove |item |mutator |Removes the first occurrence of item

In [4]:
```
mylist = []
mylist.append("duke")
mylist.append("unc")
mylist.append("ncstate")
print(mylist)

mylist.insert(1, "vt")
print(mylist)
print(mylist.count(12))

print(mylist.index("unc"))
print(mylist.count(5))

mylist.reverse()
print(mylist)

mylist.sort()
print(mylist)

mylist.remove("duke")
print(mylist)

lastitem = mylist.pop()
print(lastitem)
print(mylist)
```

```
['duke', 'unc', 'ncstate']
['duke', 'vt', 'unc', 'ncstate']
0
2
0
['ncstate', 'unc', 'vt', 'duke']
['duke', 'ncstate', 'unc', 'vt']
['ncstate', 'unc', 'vt']
vt
['ncstate', 'unc']
```

Tuples

- Tuples are similar to lists, but are immutable

In [5]:
```
t = ('a', 'b', 'c') #typical creation using ()
print (t)
print (type(t))

a = ['unc', 'vt']
print (a)
print (type(a))

t = tuple(a)  # can also create by passing a list to tuple()
print (t)
print (type(t))
```

```
('a', 'b', 'c')
<class 'tuple'>
['unc', 'vt']
<class 'list'>
('unc', 'vt')
<class 'tuple'>
```

Indexing works like lists

- Accessing elements of a tuple works similar to lists:

In [6]:
```
t = ('a', 'b', 'c', 'd')
print(t[1])
print(t[1:3])
```

```
b
('b', 'c')
```

Tuples are immutable

- Meaning that the elements in a tuple cannot be changed
- The following code will produce a TypeError:

In [7]:
```
t = ('a', 'b', 'c', 'd')
t[1] = 'x'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_17272/3669148927.py in <module>
      1 t = ('a', 'b', 'c', 'd')
----> 2 t[1] = 'x'

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

Dictionaries

- So far, we have talked about sequential collections: lists, tuples
  - Have an order from left to right
  - Use integer indices to access values (e.g. a[0], t[2])
- Dictionaries are a *mapping* type

- Unordered, associative collection
- Mapping from *keys* to *values*
  - Keys can be any immutable type
  - Values can be any Python data object (including other collections)
- Dictionaries are mutable

In [8]:
```
# in this example, we will map keys that are strings to values that are different strin

e2s = {}   # create an empty dictionary using {}
e2s['one'] = 'uno'   # associate the key 'one' with the value 'uno'
e2s['two'] = 'dos'
e2s['three'] = 'tres'
print (e2s)
print (e2s['two'])
```

```
{'one': 'uno', 'two': 'dos', 'three': 'tres'}
dos
```

Dictionary operations

- We can access items in a dictionary and perform operations to
  - change the values associated with a key
  - delete elements in the dictionary
- Keep in mind that the key-value pairs in a dictionary are *unordered*
  - Even though when we print the dictionary, they are output in an order
  - There is no order to the key-value pairs in the dictionary

In [9]:
```
inv = {'apples': 430, 'bananas':312,
       'oranges': 523, 'pears':217}
print (inv)
inv['pears'] = 0
inv['bananas'] += 200
del inv['oranges']
print (inv)
print (len(inv))
```

```
{'apples': 430, 'bananas': 312, 'oranges': 523, 'pears': 217}
{'apples': 430, 'bananas': 512, 'pears': 0}
3
```

Dictionary methods

- There are a number of different methods we can use to perform operations with dictionaries:

Method	Parameters	Description
keys	none	Returns a view of the keys in the dict
values	none	Returns a view of the values in the dict

Method	Parameters	Description
items	none	Returns a view of the key-value pairs in the dict
get	key	Returns the value associated with the key; if the key does not exist, returns None
get	key,alt	Returns the value associated with the key; if the key does not exist, returns alt

In [10]:
```
inv = {'apples': 430, 'bananas':312, 'oranges': 523, 'pears':217}

# keys() returns a list of the keys in the dict
print (inv.keys())

for akey in inv.keys():
    print ("The key", akey, "maps to value", inv[akey])
```

```
dict_keys(['apples', 'bananas', 'oranges', 'pears'])
The key apples maps to value 430
The key bananas maps to value 312
The key oranges maps to value 523
The key pears maps to value 217
```

In [11]:
```
inv = {'apples': 430, 'bananas':312, 'oranges': 523, 'pears':217}

# items() returns key-value pairs in the dict as a list of tuples
print(inv.items())

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

```
dict_items([('apples', 430), ('bananas', 312), ('oranges', 523), ('pears', 217)])
apples 430
bananas 312
oranges 523
pears 217
```

In [12]:
```
inv = {'apples': 430, 'bananas':312, 'oranges': 523, 'pears':217}

# for..in can be used to iterate over the keys in a dict
for k in inv:
    print (k, inv[k])
```

```
apples 430
bananas 312
oranges 523
pears 217
```

A very big issue to know about dictionaries

- Using the bracket notation to try to access a key that does not exist in a dict will generate a KeyError

In [13]:
```
inv = {'apples': 430, 'bananas':312, 'oranges': 523, 'pears':217}
print (inv['apples'])
```

```
print (inv['kiwi'])      # error!
```

```
430
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_17272/1774483177.py in <module>
      1 inv = {'apples': 430, 'bananas':312, 'oranges': 523, 'pears':217}
      2 print (inv['apples'])
----> 3 print (inv['kiwi'])      # error!

KeyError: 'kiwi'
```

Instead, use .get() or .setdefault()

- .get(key,[default])
  - Returns the value for the key if the key is in the dict. Otherwise, returns default (or None if default is not specified).

Method	Parameters	Description
get	key	Returns the value associated with the key; if the key does not exist in the dict, returns None
get	key, default	Returns the value associated with the key; if the key does not exist in the dict, returns default
setdefault	key, default	Returns the value associated with the key; if not, inserts the key with value of default and returns default

In [14]:
```
inv = {'apples': 430, 'bananas':312, 'oranges': 523, 'pears':217}
print (inv.get('kiwi'))
print (inv.get('kiwi',0))
print (inv.get('kiwi',9999))
print (inv)
```

```
None
0
9999
{'apples': 430, 'bananas': 312, 'oranges': 523, 'pears': 217}
```

.get() versus .setdefault()

In [15]:
```
inv = {'apples': 430, 'bananas':312, 'oranges': 523, 'pears':217}
print (inv.setdefault('kiwi',0))
print (inv)
```

```
0
{'apples': 430, 'bananas': 312, 'oranges': 523, 'pears': 217, 'kiwi': 0}
```

Defaultdict (part of collections in the Python standard library)

- defaultdict is like a dict, expect it will automatically initialize new keys the first time they are seen

- defaultdict uses a *default_factory* that must be specified (e.g., often a datatype like int or list)

To use defaultdict, we must first import it from the collections library:

In [16]:
```
from collections import defaultdict
data = [('apples', 430), ('bananas',312), ('oranges', 523), ('pears', 217)]
inv = defaultdict(int)   # int is the default_factory; this initializes new values to b
for k,v in data:
    inv[k] = v
print (inv)
inv['grape']
inv['kiwi'] += 1
print (inv)
```

```
defaultdict(<class 'int'>, {'apples': 430, 'bananas': 312, 'oranges': 523, 'pears': 21
7})
defaultdict(<class 'int'>, {'apples': 430, 'bananas': 312, 'oranges': 523, 'pears': 217,
'grape': 0, 'kiwi': 1})
```

Counter (part of collections in the Python standard library)

- Counter is a dict subclass for counting objects.
- Elements are stored as keys and their counts are stored as values

To use counter, we must first import it from the collections library:

In [17]:
```
from collections import Counter
t = ['a','a','b','a','c','b','a','d']
c = Counter(t)
print(c)
```

```
Counter({'a': 4, 'b': 2, 'c': 1, 'd': 1})
```

Counter has useful functions such as .most_common()

- Remember that Counters are not lists
- To access elements, we must use Counter methods:

In [18]:
```
from collections import Counter
t = ['a','a','b','a','c','b','a','d']
c = Counter(t)
top2 = c.most_common(2)  # will return two most common key-value pairs as a list of tup
print (top2)
```

```
[('a', 4), ('b', 2)]
```

Exercise D1.1 – Counting

- Assume you have the following data about course numbers (e.g., 760, 509) and instructor names (e.g., Capra, Arguello) in a text file called course_data.txt:

```
760 Capra
509 Arguello
512 Haas
523 Capra
884 Kelly
509 Kelly
523 Haas
523 Mostafa
509 Losee
```

Write a Python program to read the data and do the following:

1. Create a dict in which the keys are the course numbers and the value associated with each key is a list of the names of the instructors who have taught the class (duplicate names in the list are okay).
2. Create a Collection Counter object and use it to print out which two courses have been taught the most times.

You can assume that you start with the following code:

```
fp = open("course_data.txt","r")

mydict = {}

for line in fp:
    line = line.strip()
    (cnum, instr) = line.split()

    # add your code here

print (mydict)
```

JSON -- JavaScript Object Notation

- JSON is a textual representation of a JavaScript data object
- JSON is a very common lightweigh data interchange format
- Even though it originated with JavaScript, it is used across many different languages
- There are a *lot* of datasets available in JSON format
- Python has support for JSON using the standard library

Below is an example JSON data:

- Notice that JSON uses {} for key-value pairs (like Python dicts)
- And [] for sequences of items (like Python lists)

In [19]:

```
# store JSON data into a Python string s using a triple quoted string
s = """{
```

```
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": 10021
    },
    "phoneNumbers": [
        {
            "type": "home",
            "number": "212 555-1234"
        },
        {
            "type": "fax",
            "number": "646 555-4567"
        }
    ]
}"""
```

In [20]:

```
print(s)
```

```
{
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": 10021
    },
    "phoneNumbers": [
        {
            "type": "home",
            "number": "212 555-1234"
        },
        {
            "type": "fax",
            "number": "646 555-4567"
        }
    ]
}
```

- In the case above, j is a dictionary with key-value pairs.
- The value for the key 'address' is another dictionary with keys streetAddress, city, state, and postalCode
- The value for the key 'phoneNumbers' is a list with two items
  - Each of the two items is a dictionary with two keys: type and number

Reading JSON data from a file

- Often, JSON data will be stored in a text file and we will read it into a string using open() and read():

In [ ]:
```
fp = open("ex1.json","r")
s = fp.read()
```

Load a JSON text string into a Python data object

- Once we have JSON data in a text string, we can use the json library to load it into a Python data object
- To do this, we must first import json:

In [21]:
```
import json

#fp = open("ex1.json", "r")
#s = fp.read()

j = json.loads(s)

print (j)
print ()

print ("first = ", j['firstName'])
print ("last = ", j['lastName'])
print ("age = ", j['age'])
print ("city = ", j['address']['city'])
for phnum in j['phoneNumbers']:
    print (phnum['type'], " = ",
           phnum['number'])
```

```
{'firstName': 'John', 'lastName': 'Smith', 'age': 25, 'address': {'streetAddress': '21 2
nd Street', 'city': 'New York', 'state': 'NY', 'postalCode': 10021}, 'phoneNumbers':
[{'type': 'home', 'number': '212 555-1234'}, {'type': 'fax', 'number': '646 555-4567'}]}

first =  John
last =  Smith
age =  25
city =  New York
home  =  212 555-1234
fax  =  646 555-4567
```

- After we have used .loads(), the Python data object can be used like any other complex data structure.

Exercise D1.2 – JSON

- Assume you have the following JSON data in a text file called json.txt:

```
{
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address": {
        "streetAddress": "21 2nd Street",
```

```
            "city": "New York",
            "state": "NY",
            "postalCode": 10021
        },
        "phoneNumbers": [
            {
                "type": "home",
                "number": "212 555-1234"
            },
            {
                "type": "fax",
                "number": "646 555-4567"
            }
        ]
    }
```

Write a Python program to read the data into a Python data object. Then access and print the following values from the data object:

1. 'Smith'
2. 'NY'
3. '212 555-1234'
4. 'fax'

In [ ]: