COLUMBIA | ENGINEERING
EXECUTIVE EDUCATION

Week 2

Intermediate Python – Data Structures for your Analysis

Applied Data Science

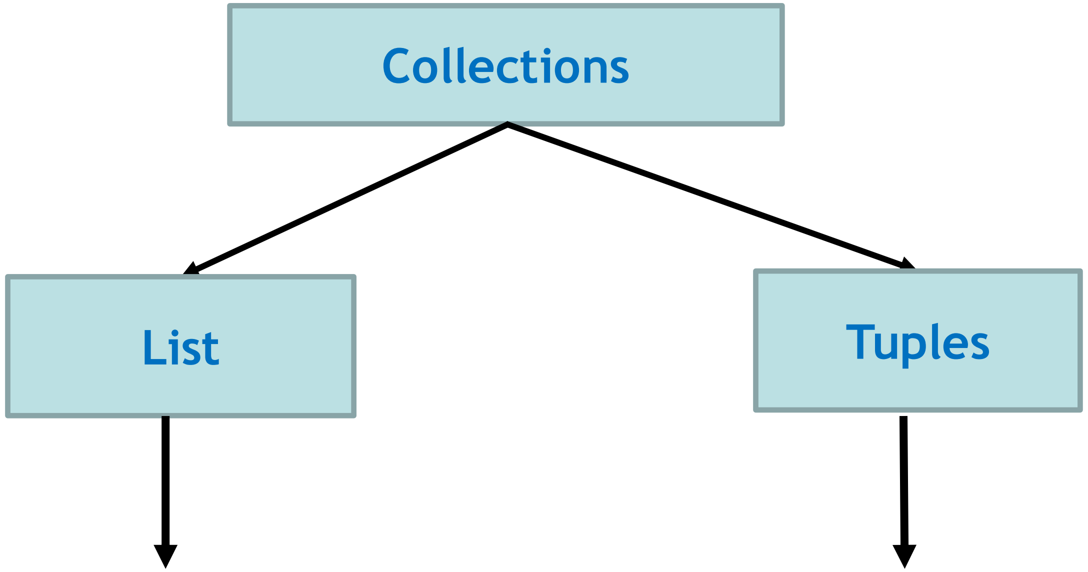
Columbia University - Columbia Engineering

Course Agenda

- Week 1: Python Basics: How to translate procedures into codes
- **Week 2: Intermediate Python – Data Structures for Your Analysis**
- Week 3: Relational Databases – Where Big Data is Typically Stored
- Week 4: SQL – Ubiquitous Database Format/Language
- Week 5: Statistical Distributions – The Shape of Data
- Week 6: Sampling – When You Can't or Won't Have ALL the Data
- Week 7: Hypothesis Testing – Answering Questions about Your Data
- Week 8: Data Analysis and Visualization – Using Python's NumPy for Analysis
- Week 9: Data Analysis and Visualization – Using Python's Pandas for Data Wrangling
- Week 10: Text Mining – Automatic Understanding of Text
- Week 11: Machine Learning – Basic Regression and Classification
- Week 12: Machine learning – Decision Trees and Clustering

Python Data Structures- Collections

In addition to basic data types like strings and floating integers, there are collections.



Collections

List

Tuples

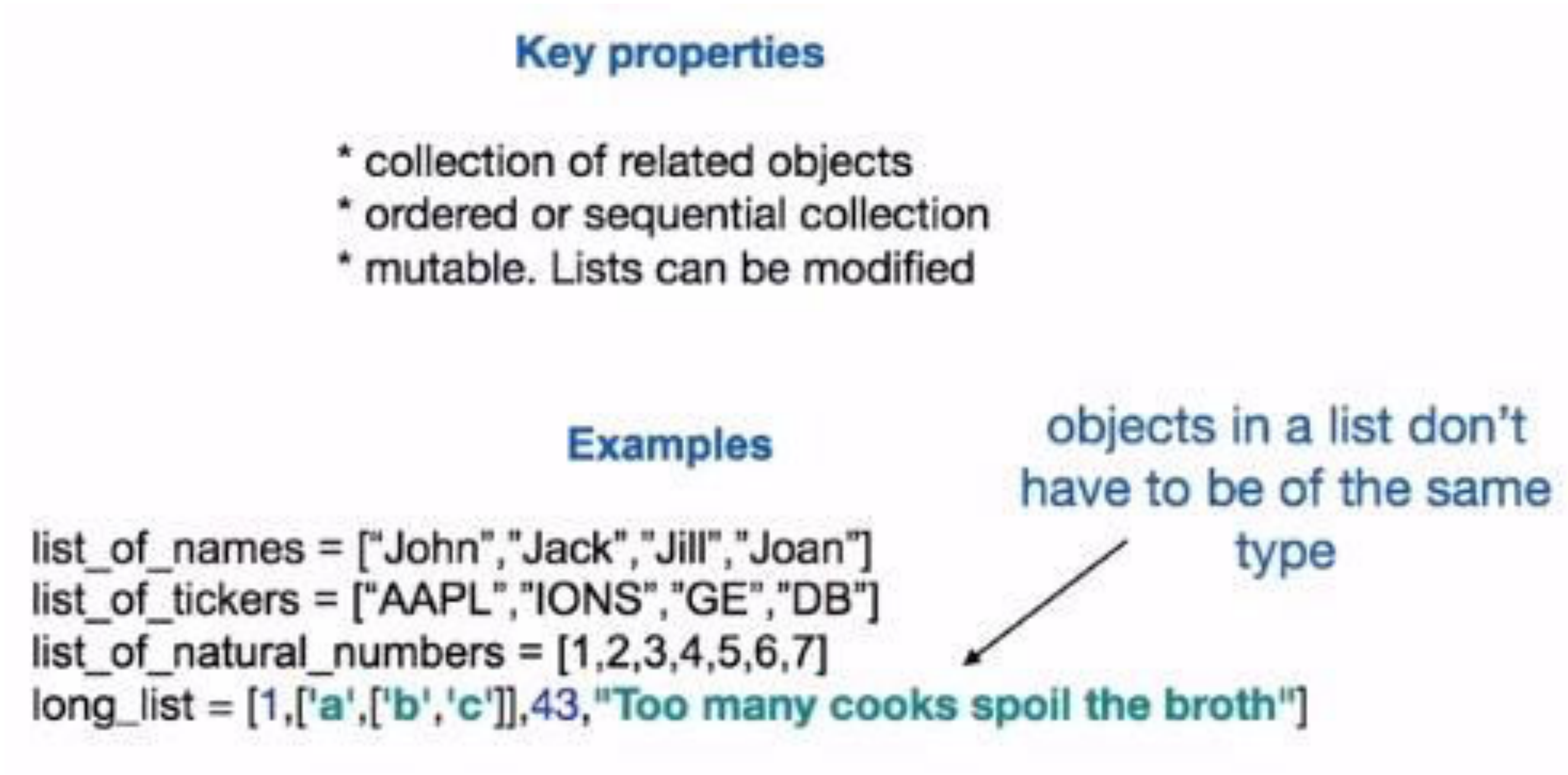
Lists are collections, that are;

- Sequential
- Ordered
- Mutable

Tuples are collections, that are;

- Sequential
- Ordered
- Immutable

Lists: Sequential Ordered Mutable Collections



Key properties

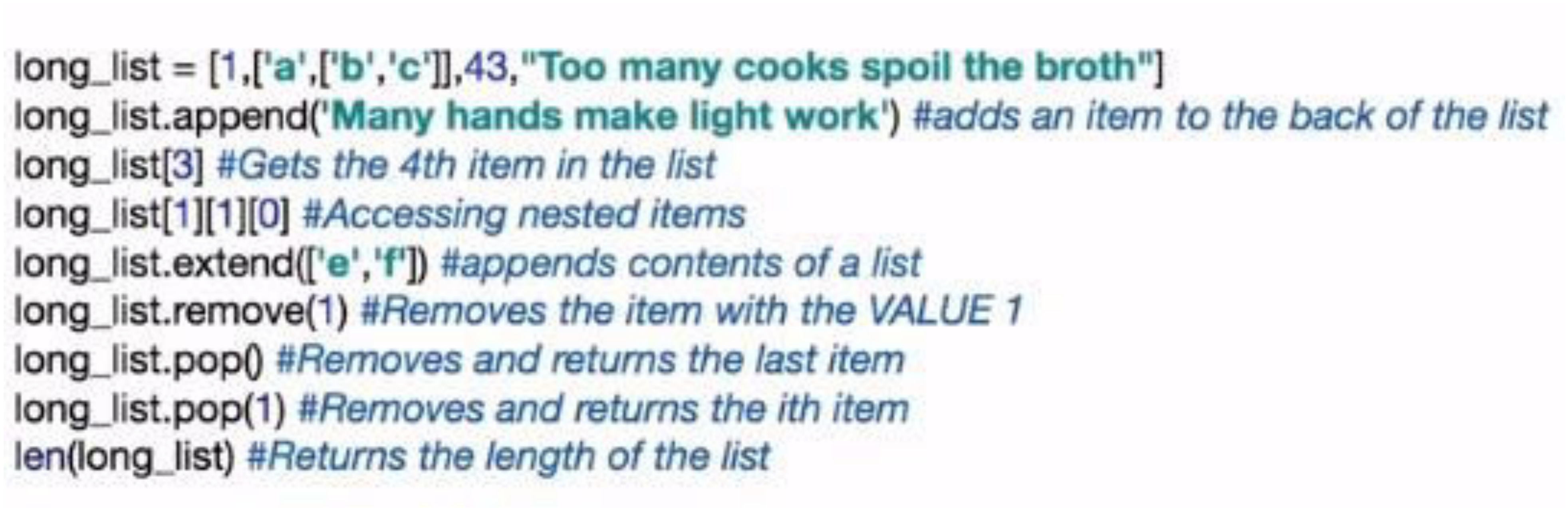
* collection of related objects
* ordered or sequential collection
* mutable. Lists can be modified

Examples

objects in a list don't have to be of the same type

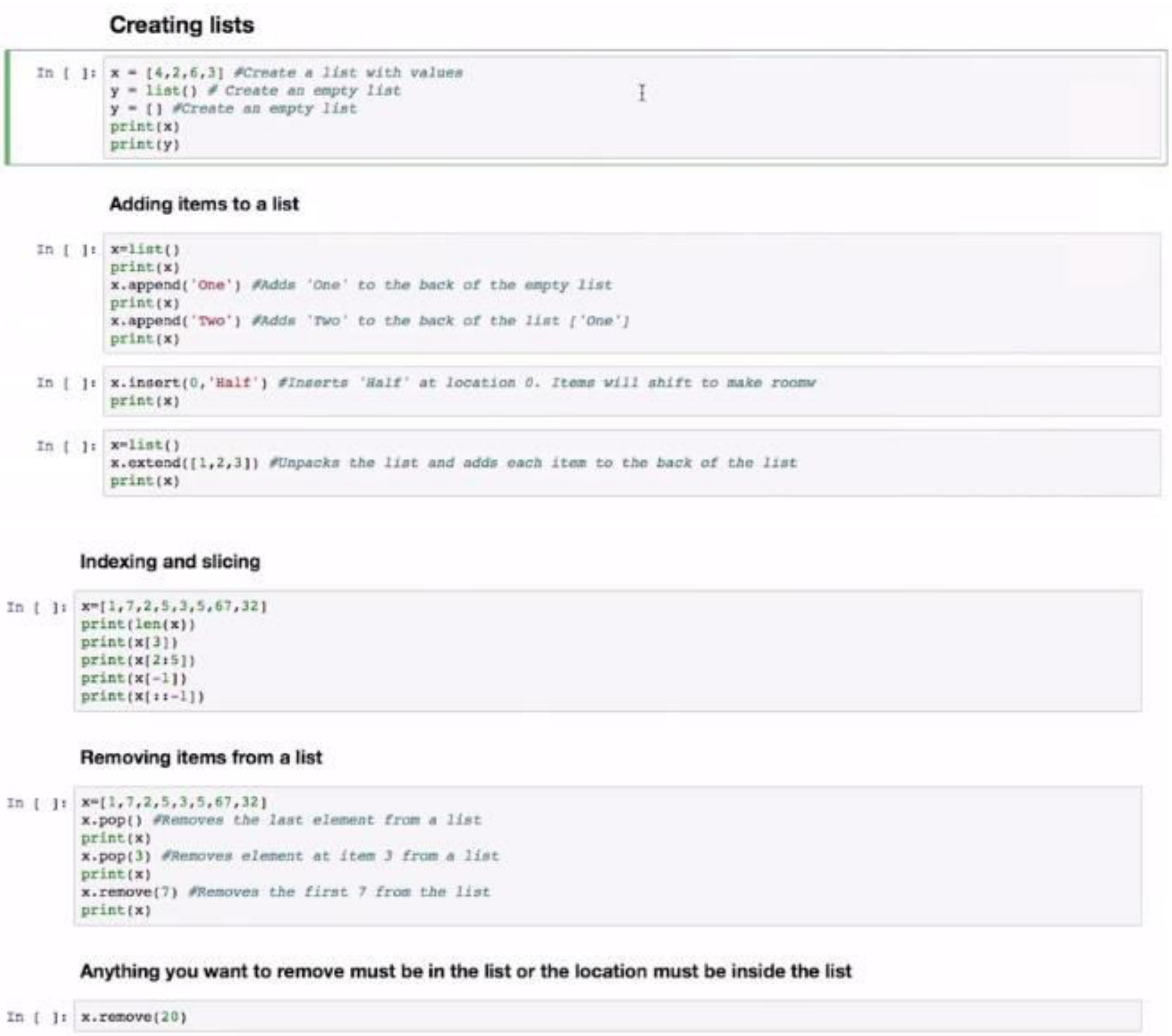
```
list_of_names = ["John","Jack","Jill","Joan"]
list_of_tickers = ["AAPL","IONS","GE","DB"]
list_of_natural_numbers = [1,2,3,4,5,6,7]
long_list = [1,['a',['b','c']],43,"Too many cooks spoil the broth"]
```

Operations on Lists



```
long_list = [1,['a',['b','c']],43,"Too many cooks spoil the broth"]
long_list.append('Many hands make light work') #adds an item to the back of the list
long_list[3] #Gets the 4th item in the list
long_list[1][1][0] #Accessing nested items
long_list.extend(['e','f']) #appends contents of a list
long_list.remove(1) #Removes the item with the VALUE 1
long_list.pop() #Removes and returns the last item
long_list.pop(1) #Removes and returns the ith item
len(long_list) #Returns the length of the list
```

Lists: Sequential, Ordered Collection



Creating lists

```
In [ ]: x = [4,2,6,3] #Create a list with values
        y = list() # Create an empty list
        y = [] #Create an empty list
        print(x)
        print(y)
```

Adding items to a list

```
In [ ]: x=list()
        print(x)
        x.append('One') #Adds 'One' to the back of the empty list
        print(x)
        x.append('Two') #Adds 'Two' to the back of the list ['One']
        print(x)
```

```
In [ ]: x.insert(0,'Half') #Inserts 'Half' at location 0. Items will shift to make roomw
        print(x)
```

```
In [ ]: x=list()
        x.extend([1,2,3]) #Unpacks the list and adds each item to the back of the list
        print(x)
```

Indexing and slicing

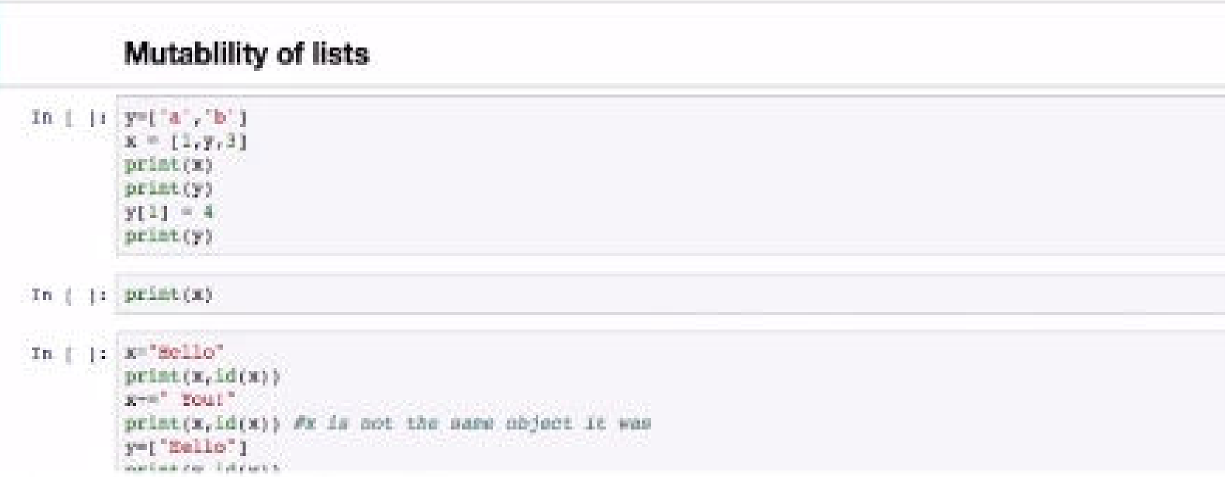
```
In [ ]: x=[1,7,2,5,3,5,67,32]
        print(len(x))
        print(x[3])
        print(x[2:5])
        print(x[-1])
        print(x[::-1])
```

Removing items from a list

```
In [ ]: x=[1,7,2,5,3,5,67,32]
        x.pop() #Removes the last element from a list
        print(x)
        x.pop(3) #Removes element at item 3 from a list
        print(x)
        x.remove(7) #Removes the first 7 from the list
        print(x)
```

Anything you want to remove must be in the list or the location must be inside the list

```
In [ ]: x.remove(20)
```



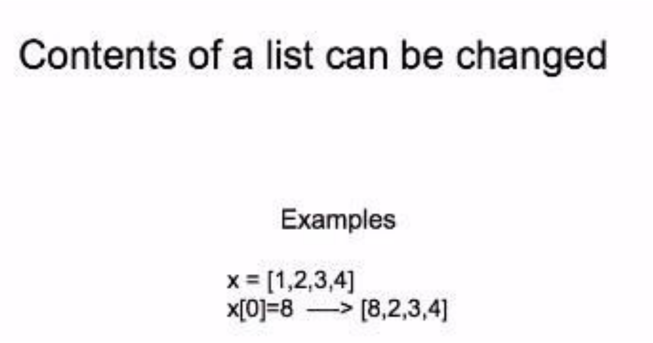
Mutablility of lists

```
In [ ]: y=['a','b']
        x = [1,y,3]
        print(x)
        print(y)
        y[1] = 4
        print(y)
```

```
In [ ]: print(x)
```

```
In [ ]: x="Hello"
        print(x,id(x))
        x+=" You!"
        print(x,id(x)) #x is not the same object it was
        y=["Hello"]
```

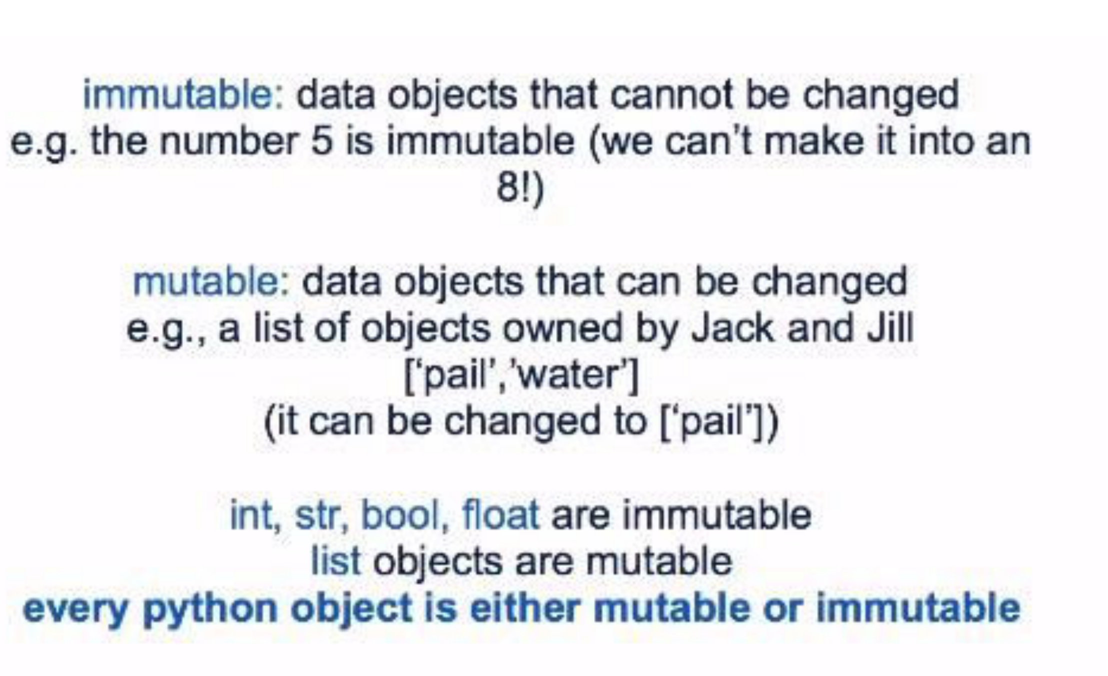
Mutability: Lists are mutable



Contents of a list can be changed

Examples

```
x = [1,2,3,4]
x[0]=8  —–> [8,2,3,4]
```

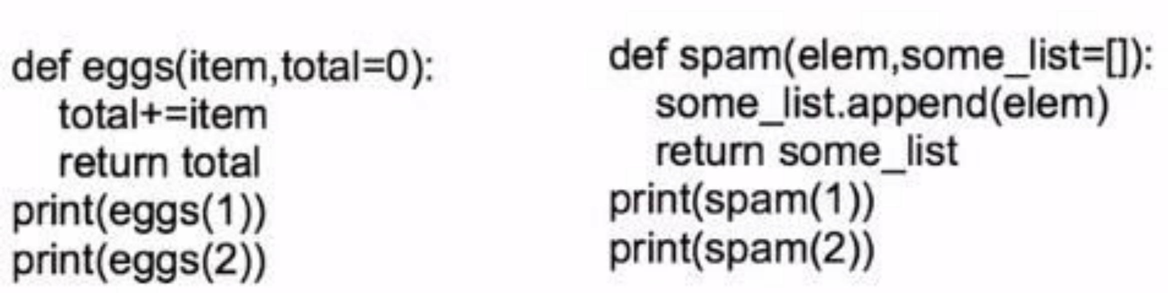


immutable: data objects that cannot be changed
e.g. the number 5 is immutable (we can't make it into an 8!)

mutable: data objects that can be changed
e.g., a list of objects owned by Jack and Jill
['pail','water']
(it can be changed to ['pail'])

int, str, bool, float are immutable
list objects are mutable
every python object is either mutable or immutable

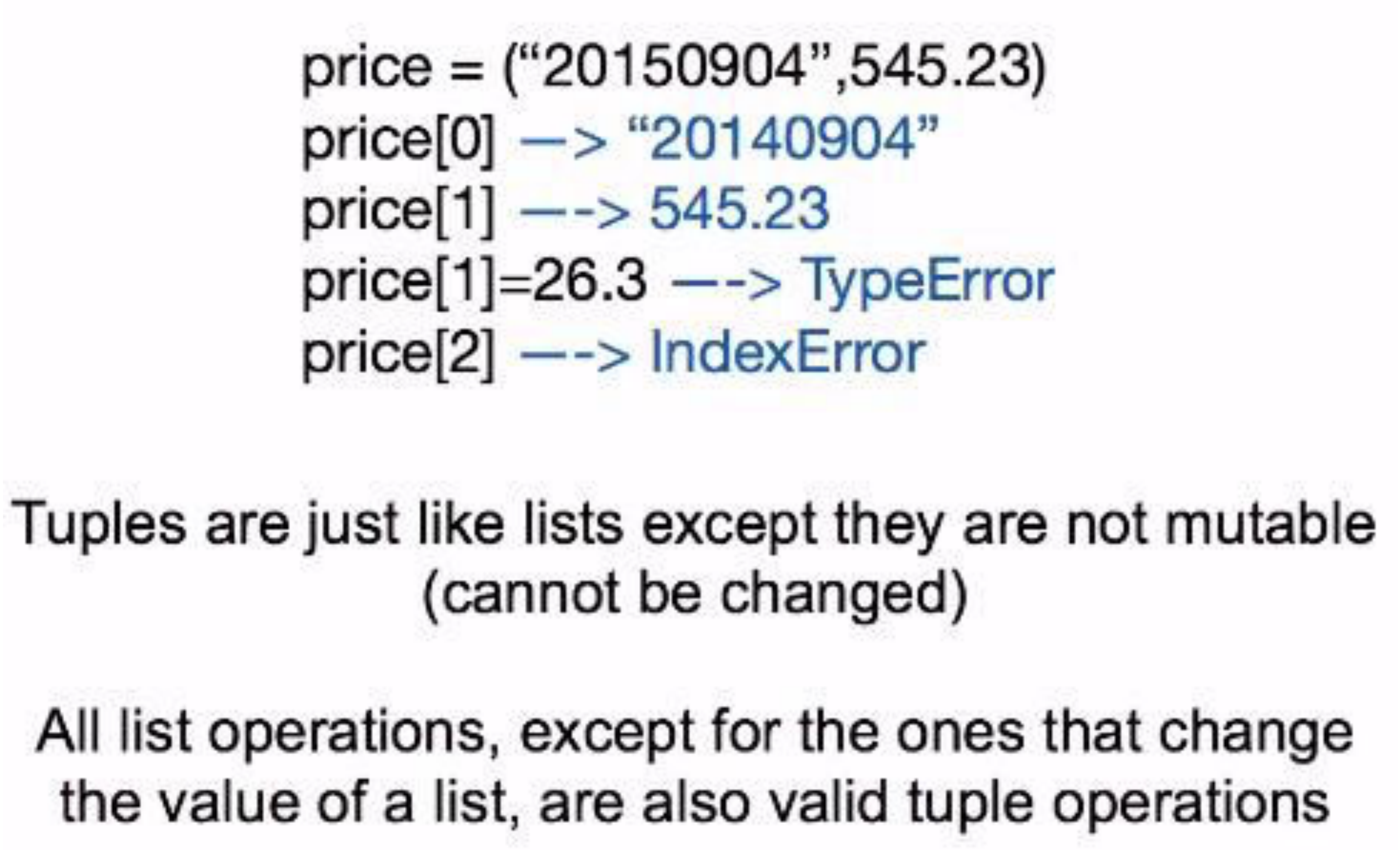
Mutable vs Immutable: What's the difference?



```
def eggs(item,total=0):
    total+=item
    return total
print(eggs(1))
print(eggs(2))
```

```
def spam(elem,some_list=[]):
    some_list.append(elem)
    return some_list
print(spam(1))
print(spam(2))
```

Tuples: Sequential, Immutable Collections

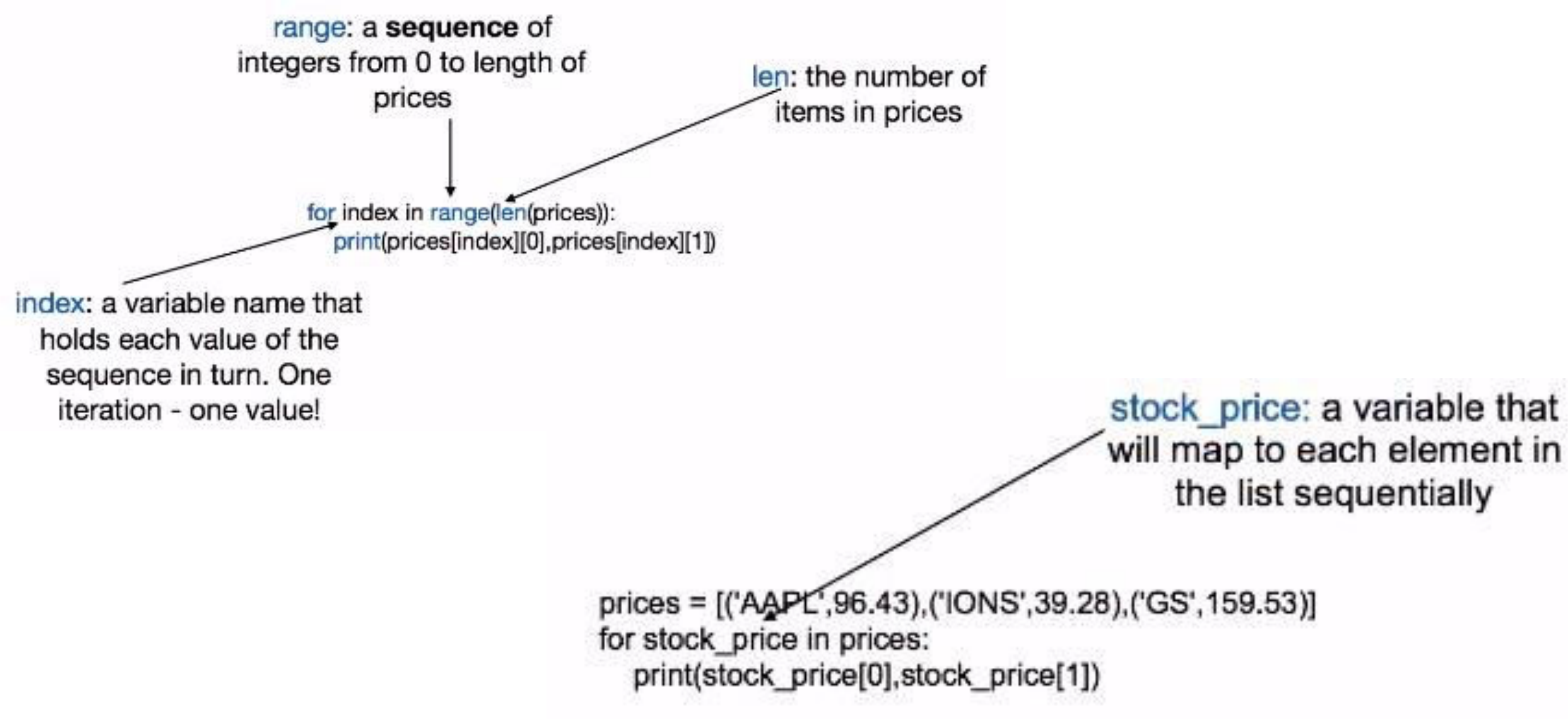


```
price = (“20150904”,545.23)
price[0] —> “20140904”
price[1] —-> 545.23
price[1]=26.3 —-> TypeError
price[2] —-> IndexError
```

Tuples are just like lists except they are not mutable (cannot be changed)

All list operations, except for the ones that change the value of a list, are also valid tuple operations

Iterating Using Location Indices and Accessing Items Sequentially



range: a **sequence** of integers from 0 to length of prices

len: the number of items in prices

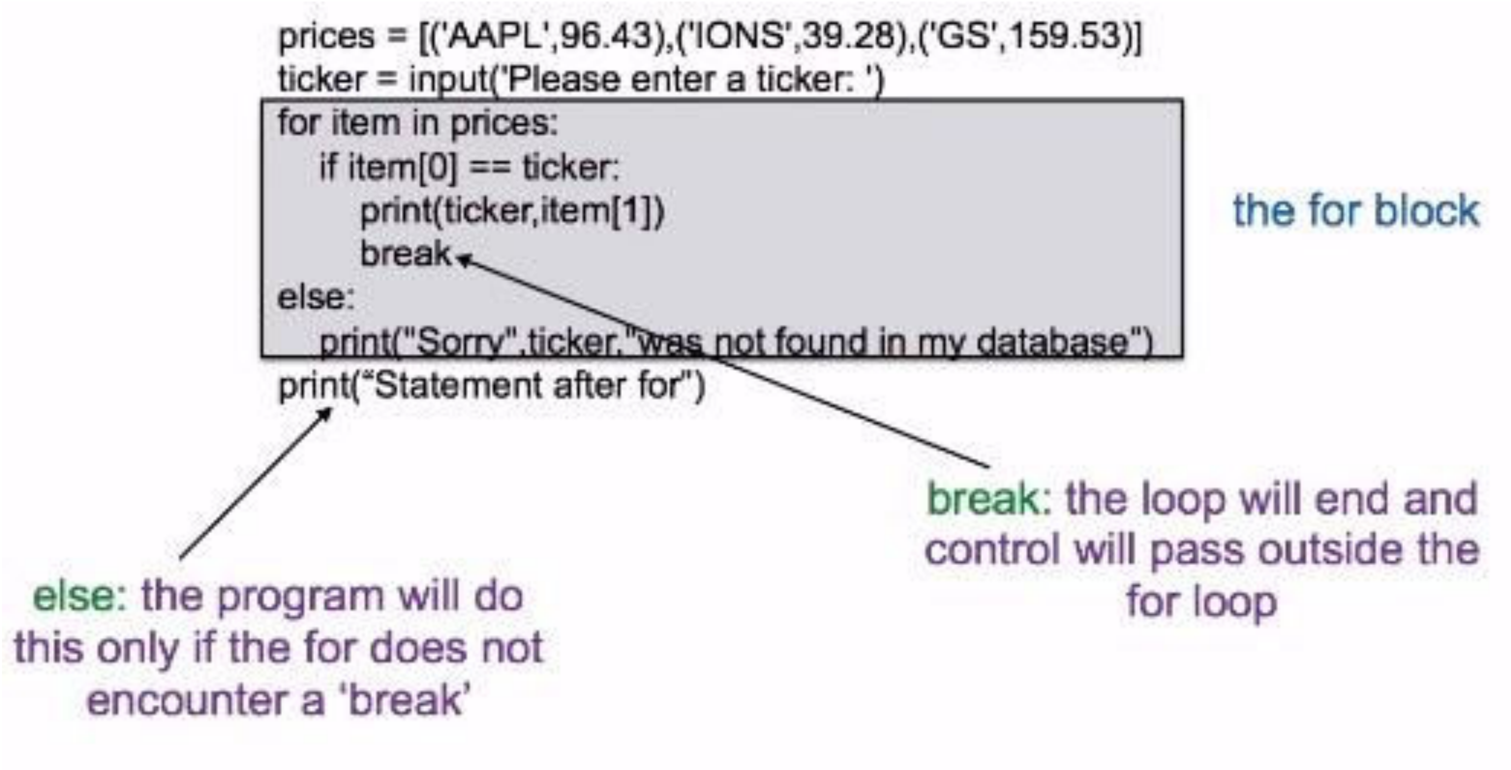
```
for index in range(len(prices)):
    print(prices[index][0],prices[index][1])
```

index: a variable name that holds each value of the sequence in turn. One iteration - one value!

stock_price: a variable that will map to each element in the list sequentially

```
prices = [('AAPL',96.43),('IONS',39.28),('GS',159.53)]
for stock_price in prices:
    print(stock_price[0],stock_price[1])
```

Controlling Iteration: Break and Else



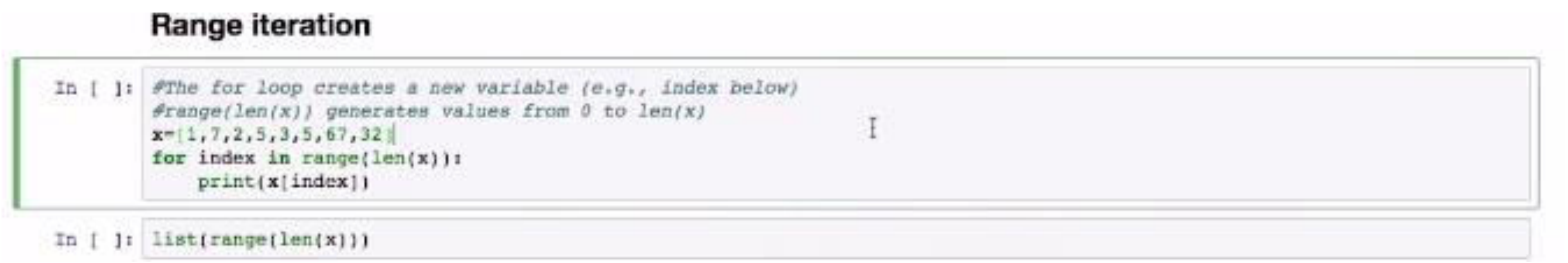
```
prices = [('AAPL',96.43),('IONS',39.28),('GS',159.53)]
ticker = input('Please enter a ticker: ')
for item in prices:
    if item[0] == ticker:
        print(ticker,item[1])
        break
else:
    print("Sorry",ticker,"was not found in my database")
print(“Statement after for")
```

the for block

break: the loop will end and control will pass outside the for loop

else: the program will do this only if the for does not encounter a 'break'

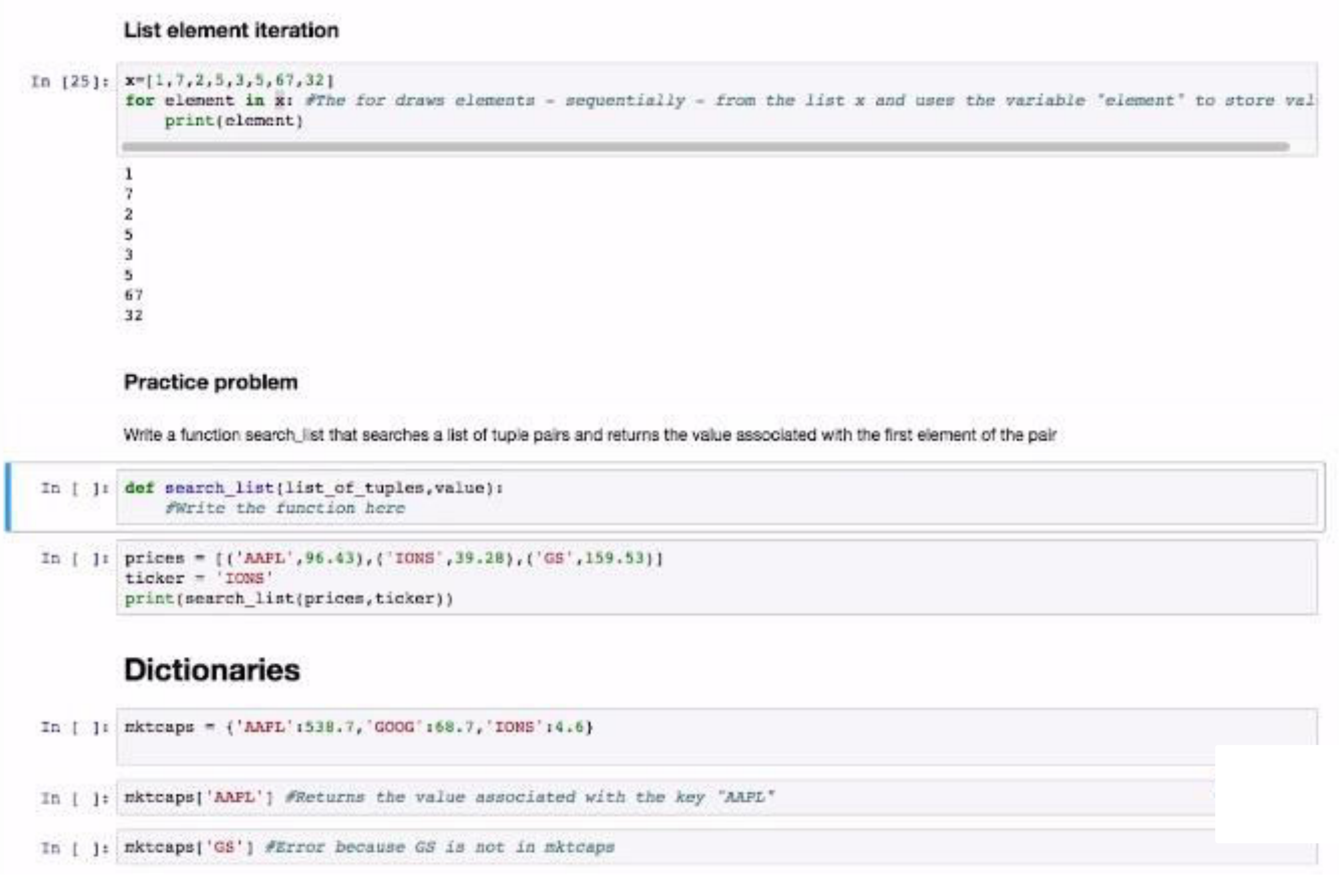
Iteration



Range iteration

```
In [ ]: #The for loop creates a new variable (e.g., index below)
        #range(len(x)) generates values from 0 to len(x)
        x=[1,7,2,5,3,5,67,32]
        for index in range(len(x)):
            print(x[index])
```

```
In [ ]: list(range(len(x)))
```



List element iteration

```
In [25]: x=[1,7,2,5,3,5,67,32]
         for element in x: #The for draws elements - sequentially - from the list x and uses the variable "element" to store val
             print(element)
```

```
1
7
2
5
3
5
67
32
```

Practice problem

Write a function search_list that searches a list of tuple pairs and returns the value associated with the first element of the pair

```
In [ ]: def search_list(list_of_tuples,value):
            #Write the function here
```

```
In [ ]: prices = [('AAPL',96.43),('IONS',39.28),('GS',159.53)]
        ticker = 'IONS'
        print(search_list(prices,ticker))
```

Dictionaries

```
In [ ]: mktcaps = {'AAPL':538.7,'GOOG':68.7,'IONS':4.6}
```

```
In [ ]: mktcaps['AAPL'] #Returns the value associated with the key "AAPL"
```

```
In [ ]: mktcaps['GS'] #Error because GS is not in mktcaps
```

Dictionaries and Sets

Dictionaries are collections that are;

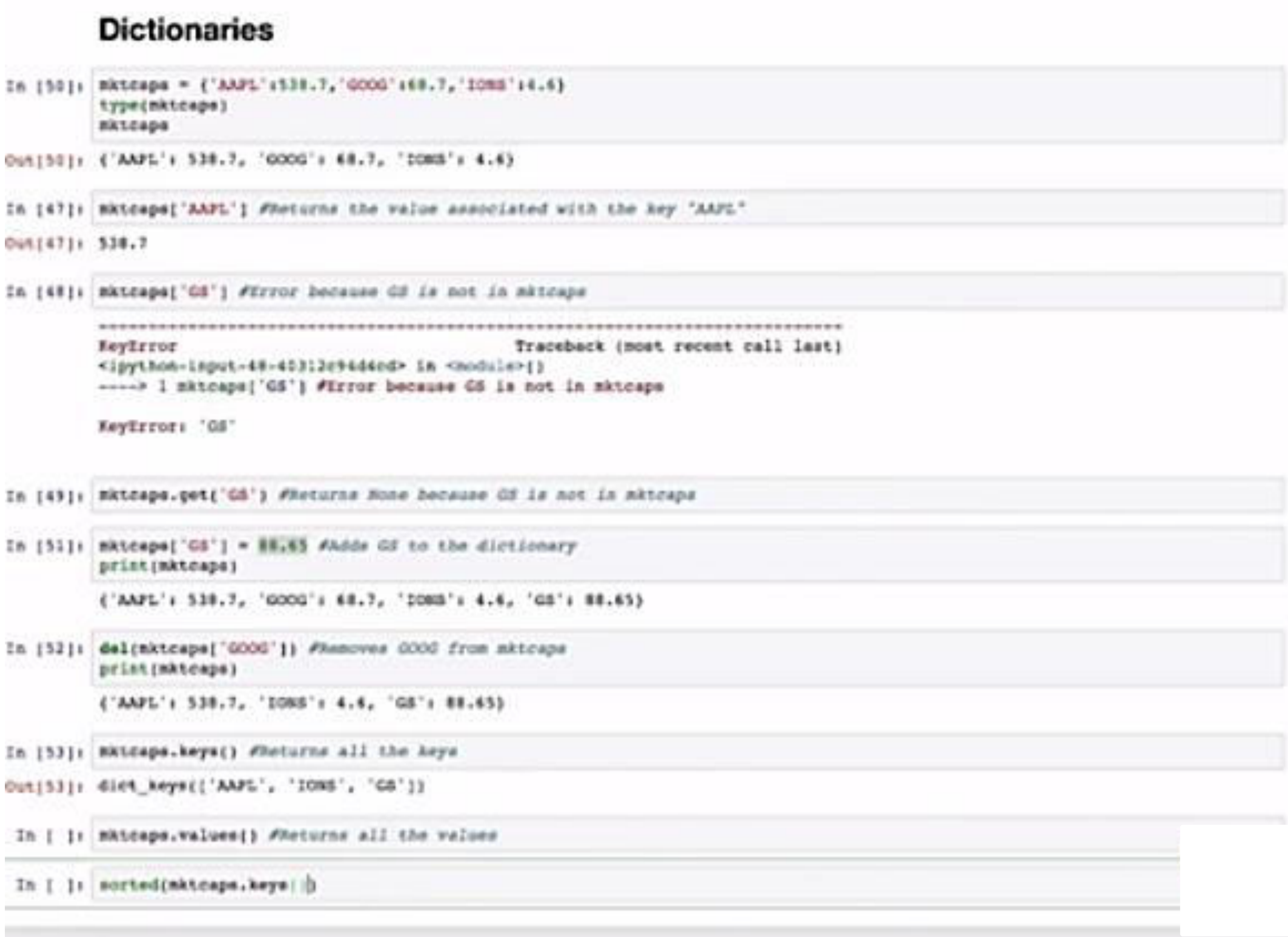
- Unordered
- Pair of elements with a key and a value
- Access values through keys
- Keys are immutable

Sets are collections that are;

- Unordered
- Collection of unique elements
- Does not contain key-value pairs
- Values are immutable

Dictionaries: Key-value Pairs

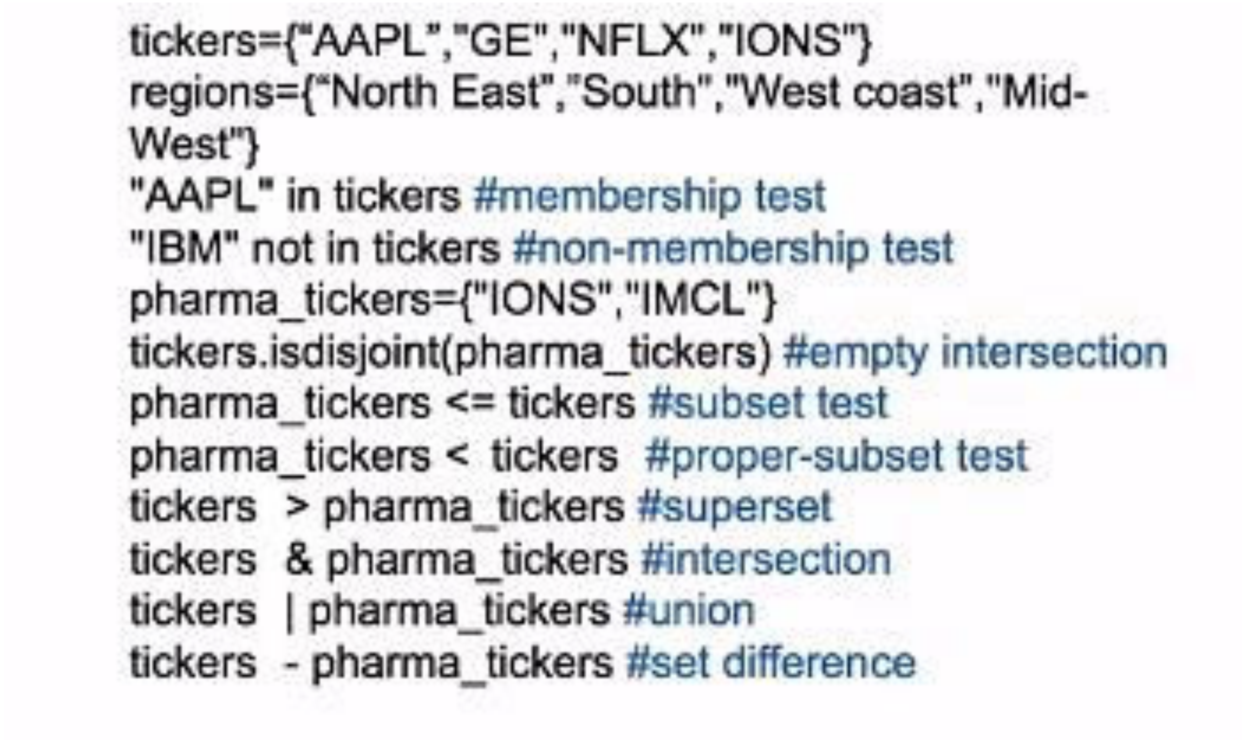
```
mktcaps = {‘AAPL':538.7,'GOOG':68.7,'IONS':4.6}
mktcaps['AAPL']  #key-based retrieval
print(mktcaps[‘AAPL'])
mktcaps['GE'] #error (no "GE")
'GE' in  mktcaps
mktcaps.keys() #returns a list of keys
sorted(mktcaps.keys()) #returns a sorted list of keys
```



Dictionaries

```
In [50]: mktcaps = {'AAPL':538.7,'GOOG':68.7,'IONS':4.6}
         type(mktcaps)
         mktcaps
Out[50]: {'AAPL': 538.7, 'GOOG': 68.7, 'IONS': 4.6}

In [47]: mktcaps['AAPL'] #Returns the value associated with the key "AAPL"
Out[47]: 538.7

In [48]: mktcaps['GS'] #Error because GS is not in mktcaps

         ---------------------------------------------------------------------------
         KeyError                                  Traceback (most recent call last)
         <ipython-input-48-403132c94d4ed> in <module>()
         ----> 1 mktcaps['GS'] #Error because GS is not in mktcaps

         KeyError: 'GS'

In [49]: mktcaps.get('GS') #Returns None because GS is not in mktcaps

In [51]: mktcaps['GS'] = 88.65 #Adds GS to the dictionary
         print(mktcaps)

         {'AAPL': 538.7, 'GOOG': 68.7, 'IONS': 4.6, 'GS': 88.65}

In [52]: del(mktcaps['GOOG']) #Removes GOOG from mktcaps
         print(mktcaps)

         {'AAPL': 538.7, 'IONS': 4.6, 'GS': 88.65}

In [53]: mktcaps.keys() #Returns all the keys
Out[53]: dict_keys(['AAPL', 'IONS', 'GS'])

In [ ]: mktcaps.values() #Returns all the values

In [ ]: sorted(mktcaps.keys())
```

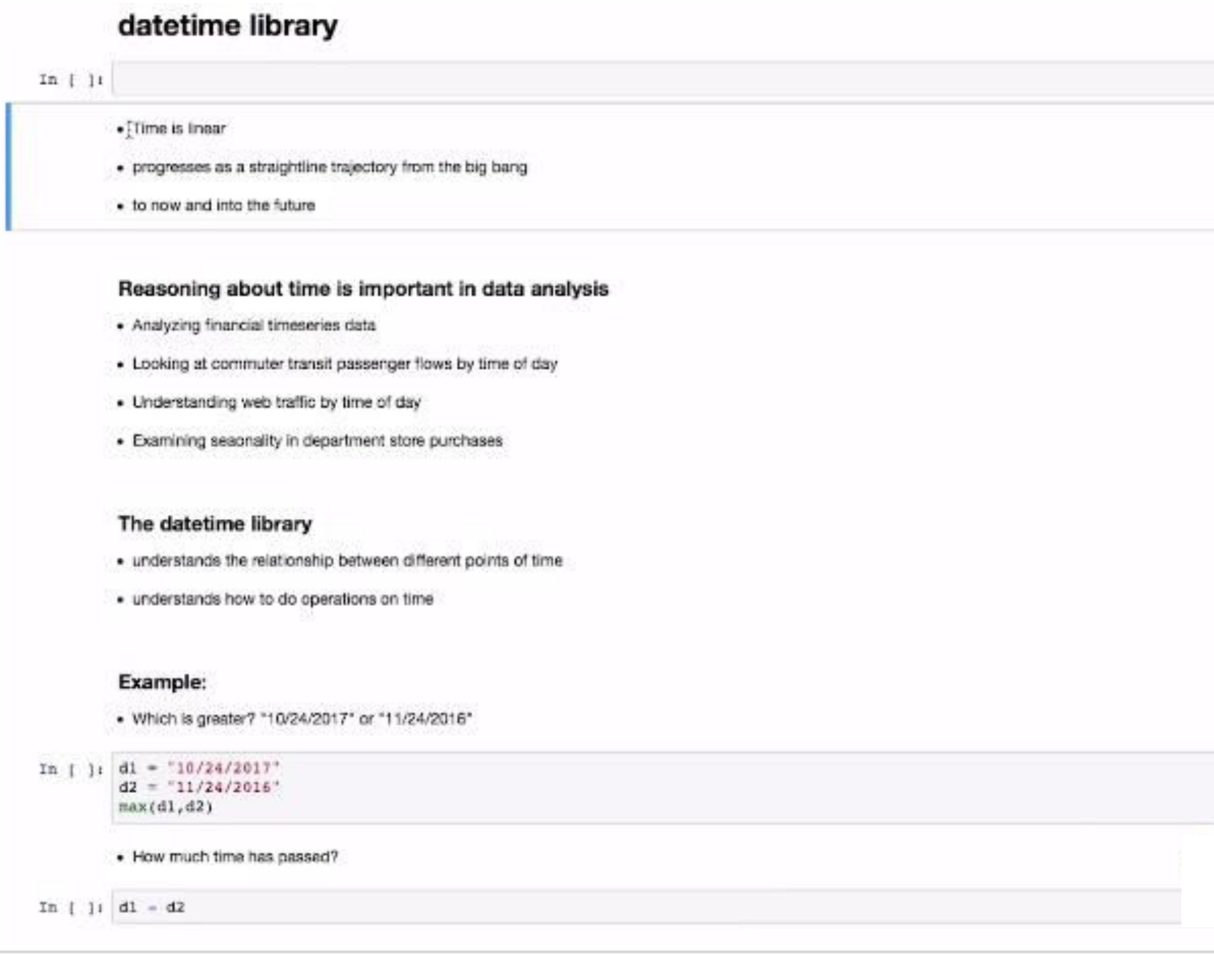
Sets: Unordered Collections of Unique Objects



```
tickers={“AAPL","GE","NFLX","IONS"}
regions={“North East","South","West coast","Mid-
West"}
"AAPL" in tickers #membership test
"IBM" not in tickers #non-membership test
pharma_tickers={"IONS","IMCL"}
tickers.isdisjoint(pharma_tickers) #empty intersection
pharma_tickers <= tickers #subset test
pharma_tickers <  tickers  #proper-subset test
tickers  > pharma_tickers #superset
tickers  & pharma_tickers #intersection
tickers  | pharma_tickers #union
tickers  - pharma_tickers #set difference
```

The Datetime Library

In Python, the ‘datetime' library is an extremely useful library for data analysis because time is a critical data element.



datetime library

```
In [ ]:
```

- Time is linear
- progresses as a straightline trajectory from the big bang
- to now and into the future

Reasoning about time is important in data analysis

- Analyzing financial timeseries data
- Looking at commuter transit passenger flows by time of day
- Understanding web traffic by time of day
- Examining seasonality in department store purchases

The datetime library

- understands the relationship between different points of time
- understands how to do operations on time

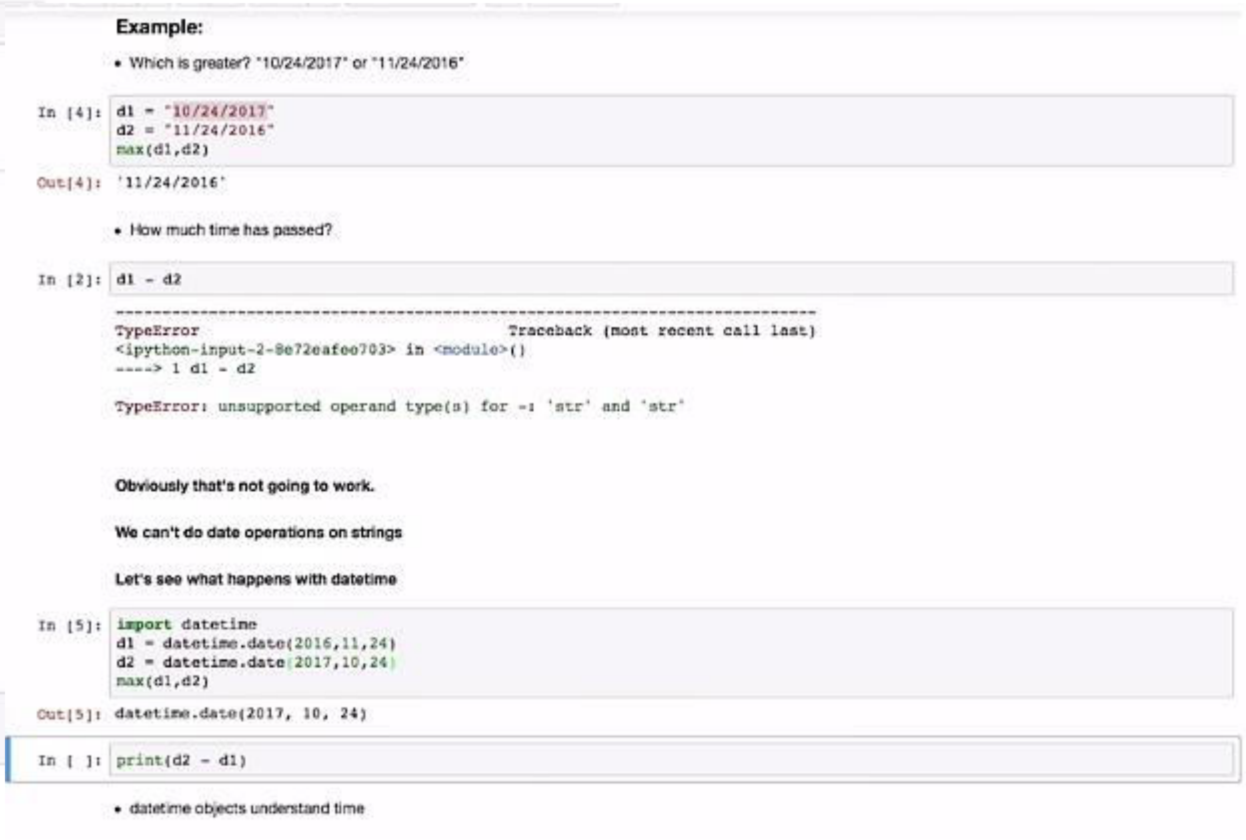
Example:

- Which is greater? "10/24/2017" or "11/24/2016"

```
In [ ]: d1 = "10/24/2017"
        d2 = "11/24/2016"
        max(d1,d2)
```

- How much time has passed?

```
In [ ]: d1 - d2
```



Example:

- Which is greater? "10/24/2017" or "11/24/2016"

```
In [4]: d1 = "10/24/2017"
        d2 = "11/24/2016"
        max(d1,d2)

Out[4]: '11/24/2016'
```

- How much time has passed?

```
In [2]: d1 - d2

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-8e72eafee703> in <module>()
----> 1 d1 - d2

TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

Obviously that's not going to work.

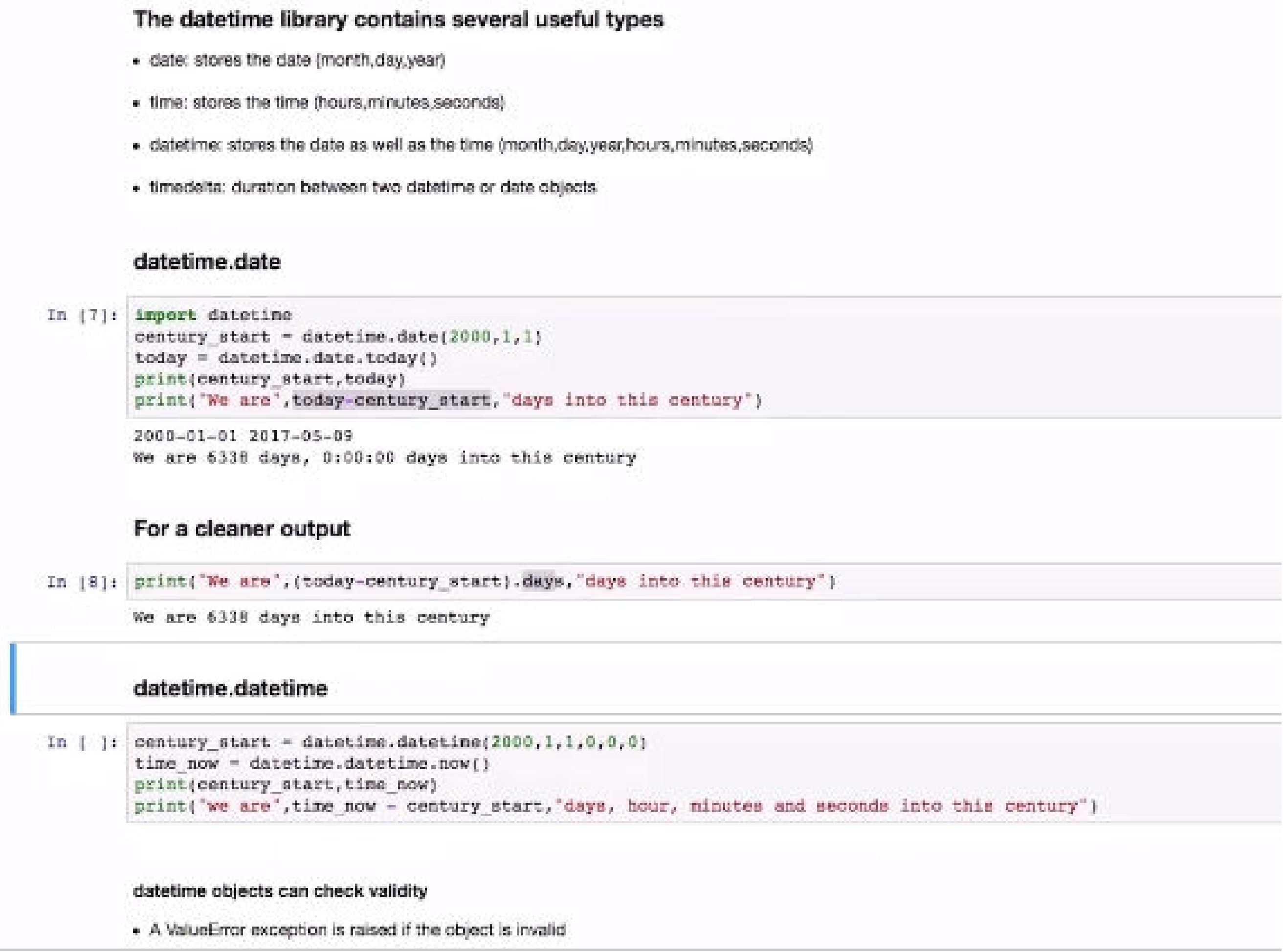
We can't do date operations on strings

Let's see what happens with datetime

```
In [5]: import datetime
        d1 = datetime.date(2016,11,24)
        d2 = datetime.date(2017,10,24)
        max(d1,d2)

Out[5]: datetime.date(2017, 10, 24)

In [ ]: print(d2 - d1)
```

- datetime objects understand time

The Datetime Library: Types 1



The datetime library contains several useful types

- date: stores the date (month,day,year)
- time: stores the time (hours,minutes,seconds)
- datetime: stores the date as well as the time (month,day,year,hours,minutes,seconds)
- timedelta: duration between two datetime or date objects

datetime.date

```
In [7]: import datetime
        century_start = datetime.date(2000,1,1)
        today = datetime.date.today()
        print(century_start,today)
        print("We are",today-century_start,"days into this century")

        2000-01-01 2017-05-09
        We are 6338 days, 0:00:00 days into this century
```

For a cleaner output

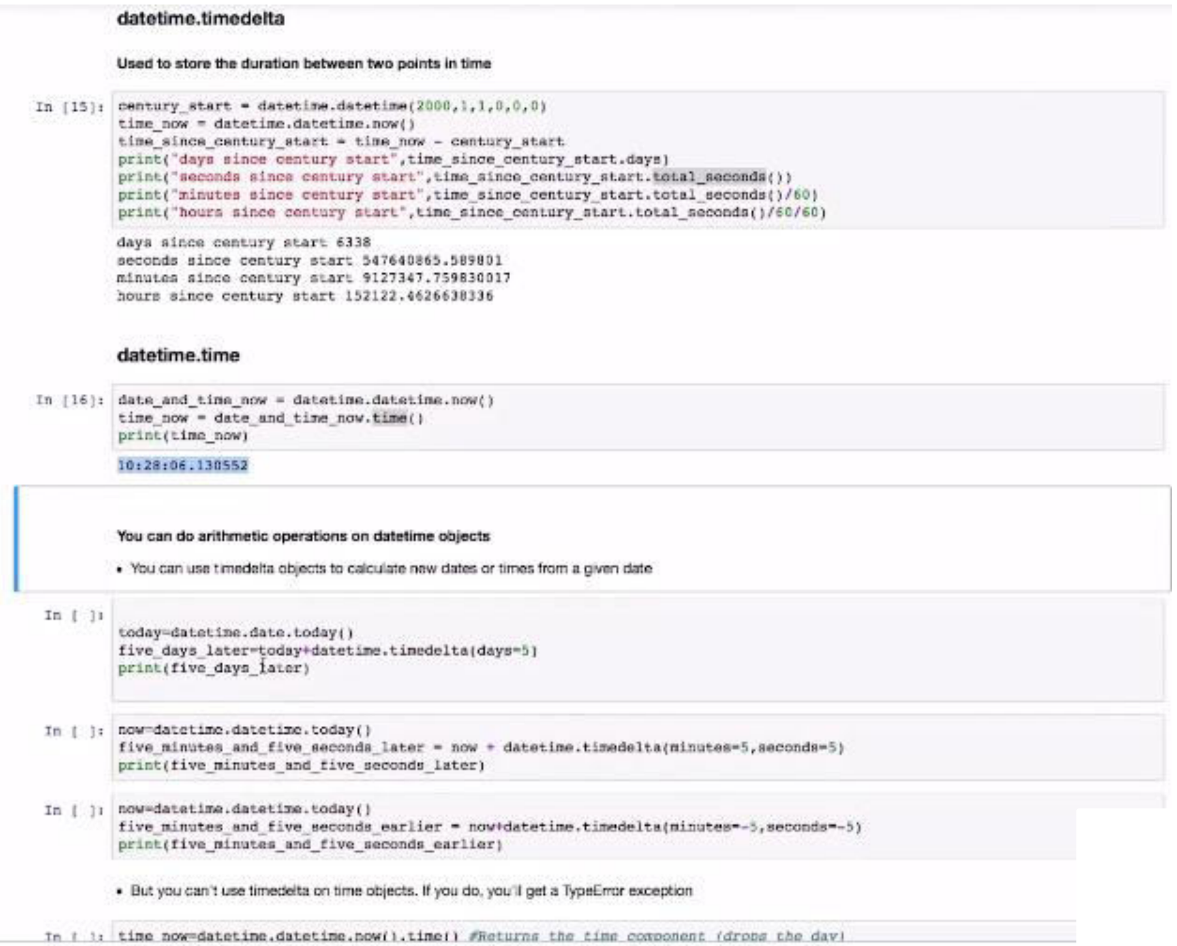
```
In [8]: print("We are",(today-century_start).days,"days into this century")

        We are 6338 days into this century
```

datetime.datetime

```
In [ ]: century_start = datetime.datetime(2000,1,1,0,0,0)
        time_now = datetime.datetime.now()
        print(century_start,time_now)
        print("we are",time_now - century_start,"days, hour, minutes and seconds into this century")
```

datetime objects can check validity

- A ValueError exception is raised if the object is invalid



datetime.timedelta

Used to store the duration between two points in time

```
In [15]: century_start = datetime.datetime(2000,1,1,0,0,0)
         time_now = datetime.datetime.now()
         time_since_century_start = time_now - century_start
         print("days since century start",time_since_century_start.days)
         print("seconds since century start",time_since_century_start.total_seconds())
         print("minutes since century start",time_since_century_start.total_seconds()/60)
         print("hours since century start",time_since_century_start.total_seconds()/60/60)

         days since century start 6338
         seconds since century start 547640865.589801
         minutes since century start 9127347.759830017
         hours since century start 152122.4626638336
```

datetime.time

```
In [16]: date_and_time_now = datetime.datetime.now()
         time_now = date_and_time_now.time()
         print(time_now)

         10:28:06.130552
```

You can do arithmetic operations on datetime objects

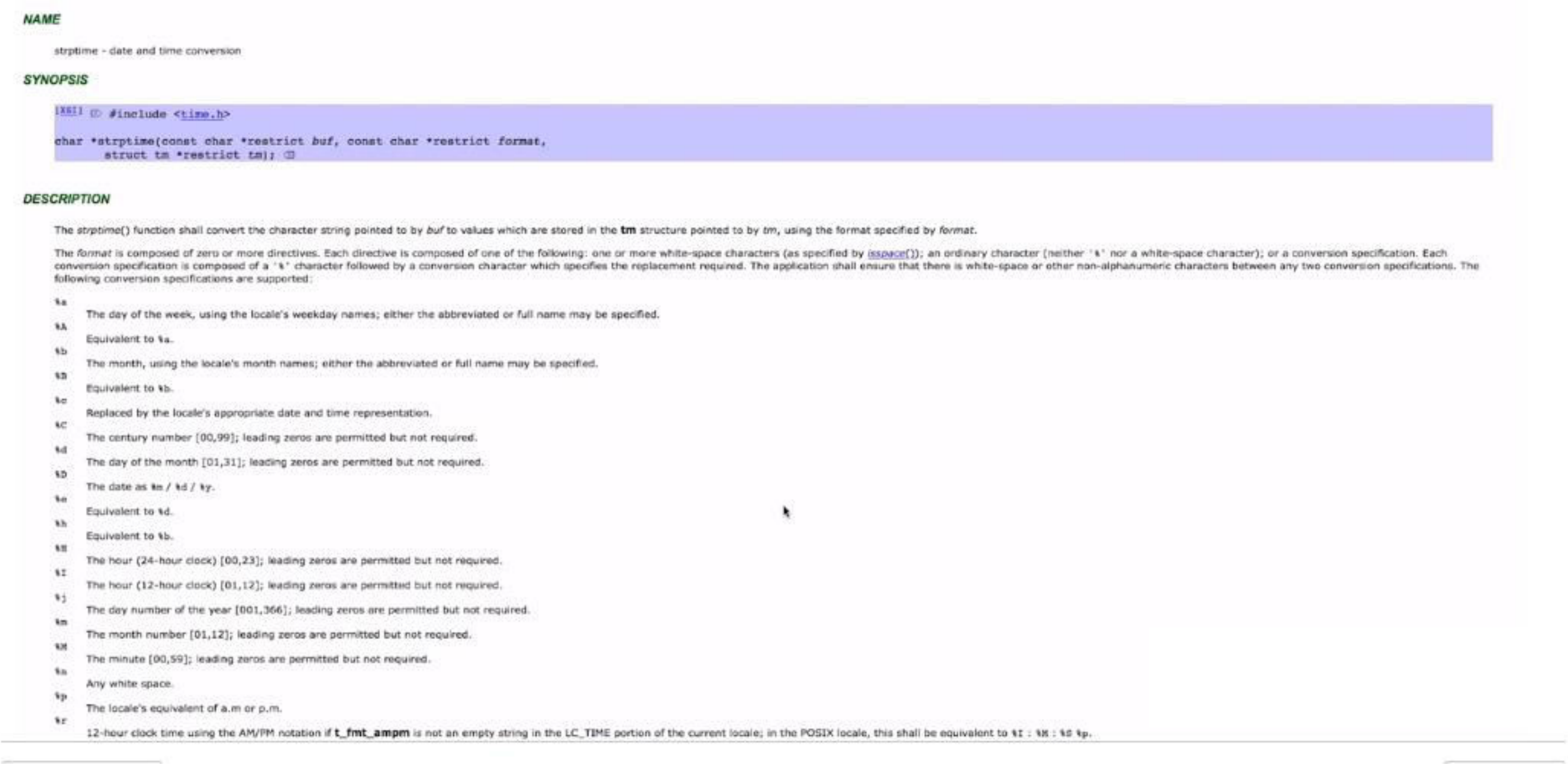
- You can use timedelta objects to calculate new dates or times from a given date

```
In [ ]: 
        today=datetime.date.today()
        five_days_later=today+datetime.timedelta(days=5)
        print(five_days_later)

In [ ]: now=datetime.datetime.today()
        five_minutes_and_five_seconds_later = now + datetime.timedelta(minutes=5,seconds=5)
        print(five_minutes_and_five_seconds_later)

In [ ]: now=datetime.datetime.today()
        five_minutes_and_five_seconds_earlier = now+datetime.timedelta(minutes=-5,seconds=-5)
        print(five_minutes_and_five_seconds_earlier)
```

- But you can't use timedelta on time objects. If you do, you'll get a TypeError exception

```
In [ ]: time_now=datetime.datetime.now().time() #Returns the time component (drops the day)
```

The Datetime Library: Types 2



NAME

strptime - date and time conversion

SYNOPSIS

```
[XSI] ⊡ #include <time.h>

char *strptime(const char *restrict buf, const char *restrict format,
       struct tm *restrict tm); ⊡
```

DESCRIPTION

The *strptime*() function shall convert the character string pointed to by *buf* to values which are stored in the **tm** structure pointed to by *tm*, using the format specified by *format*.

The *format* is composed of zero or more directives. Each directive is composed of one of the following: one or more white-space characters (as specified by isspace()); an ordinary character (neither '%' nor a white-space character); or a conversion specification. Each conversion specification is composed of a '%' character followed by a conversion character which specifies the replacement required. The application shall ensure that there is white-space or other non-alphanumeric characters between any two conversion specifications. The following conversion specifications are supported:

%a
: The day of the week, using the locale's weekday names; either the abbreviated or full name may be specified.

%A
: Equivalent to %a.

%b
: The month, using the locale's month names; either the abbreviated or full name may be specified.

%B
: Equivalent to %b.

%c
: Replaced by the locale's appropriate date and time representation.

%C
: The century number [00,99]; leading zeros are permitted but not required.

%d
: The day of the month [01,31]; leading zeros are permitted but not required.

%D
: The date as %m / %d / %y.

%e
: Equivalent to %d.

%h
: Equivalent to %b.

%H
: The hour (24-hour clock) [00,23]; leading zeros are permitted but not required.

%I
: The hour (12-hour clock) [01,12]; leading zeros are permitted but not required.

%j
: The day number of the year [001,366]; leading zeros are permitted but not required.

%m
: The month number [01,12]; leading zeros are permitted but not required.

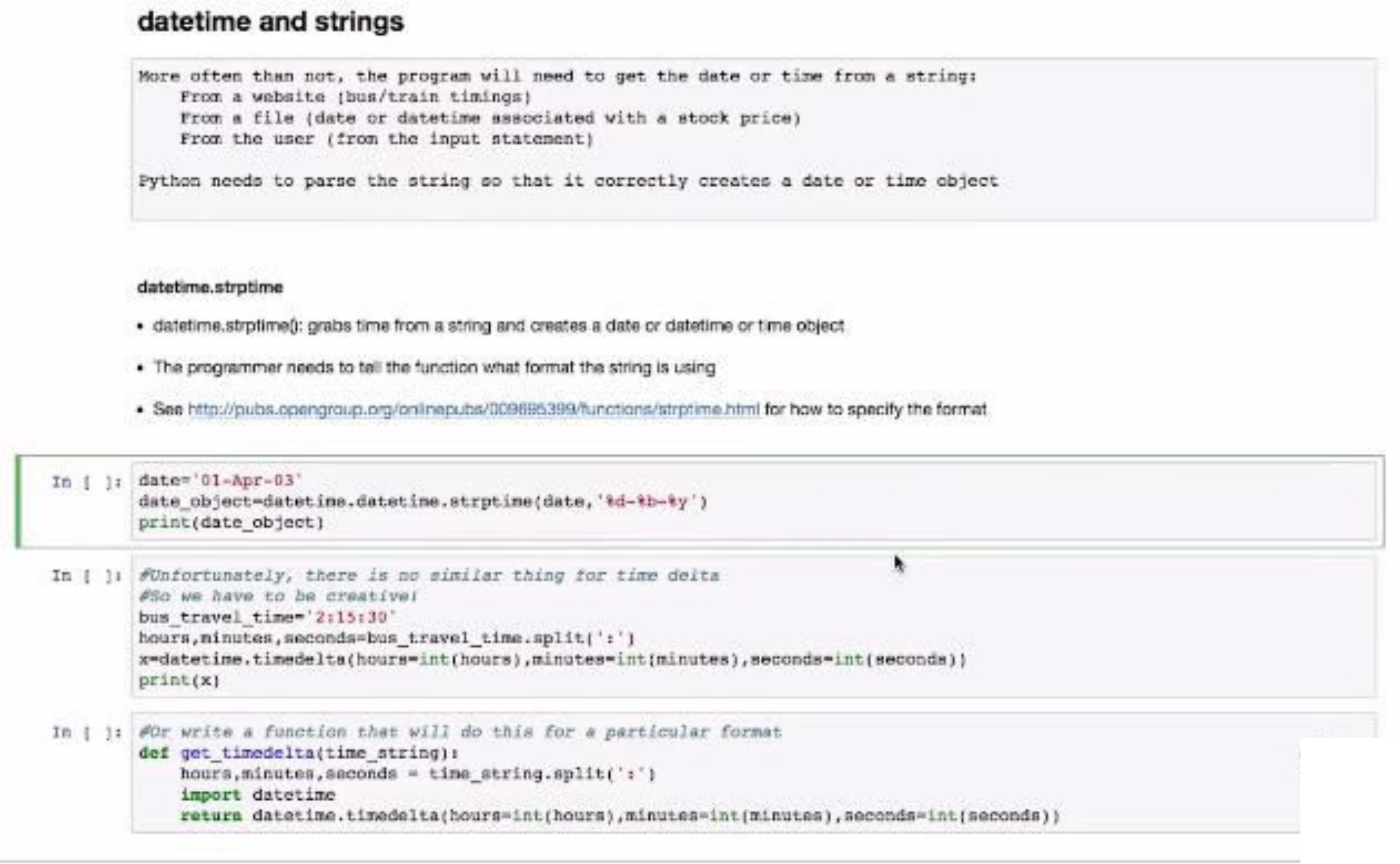
%M
: The minute [00,59]; leading zeros are permitted but not required.

%n
: Any white space.

%p
: The locale's equivalent of a.m or p.m.

%r
: 12-hour clock time using the AM/PM notation if **t_fmt_ampm** is not an empty string in the LC_TIME portion of the current locale; in the POSIX locale, this shall be equivalent to %I : %M : %S %p.

Datetime and Strings



datetime and strings

```
More often than not, the program will need to get the date or time from a string:
    From a website (bus/train timings)
    From a file (date or datetime associated with a stock price)
    From the user (from the input statement)

Python needs to parse the string so that it correctly creates a date or time object
```

datetime.strptime

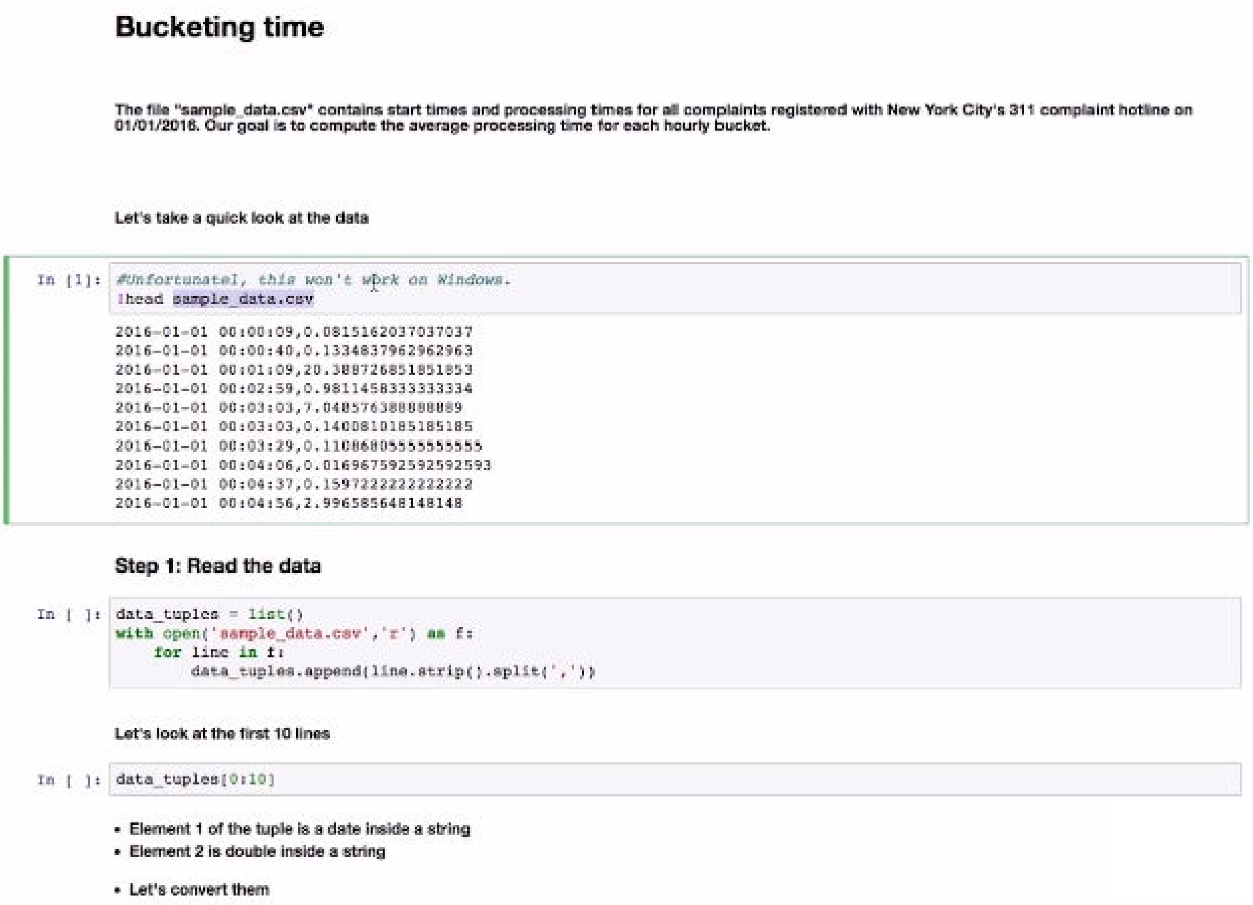
- datetime.strptime(): grabs time from a string and creates a date or datetime or time object
- The programmer needs to tell the function what format the string is using
- See http://pubs.opengroup.org/onlinepubs/009695399/functions/strptime.html for how to specify the format

```
In [ ]: date='01-Apr-03'
        date_object=datetime.datetime.strptime(date,'%d-%b-%y')
        print(date_object)
```

```
In [ ]: #Unfortunately, there is no similar thing for time delta
        #So we have to be creative!
        bus_travel_time='2:15:30'
        hours,minutes,seconds=bus_travel_time.split(':')
        x=datetime.timedelta(hours=int(hours),minutes=int(minutes),seconds=int(seconds))
        print(x)
```

```
In [ ]: #Or write a function that will do this for a particular format
        def get_timedelta(time_string):
            hours,minutes,seconds = time_string.split(':')
            import datetime
            return datetime.timedelta(hours=int(hours),minutes=int(minutes),seconds=int(seconds))
```

Bucketing Time Part 1



Bucketing time

The file "sample_data.csv" contains start times and processing times for all complaints registered with New York City's 311 complaint hotline on 01/01/2016. Our goal is to compute the average processing time for each hourly bucket.

Let's take a quick look at the data

```
In [1]: #Unfortunatel, this won't work on Windows.
        !head sample_data.csv

2016-01-01 00:00:09,0.0815162037037037
2016-01-01 00:00:40,0.1334837962962963
2016-01-01 00:01:09,20.388726851851853
2016-01-01 00:02:59,0.9811458333333334
2016-01-01 00:03:03,7.048576388888889
2016-01-01 00:03:03,0.1400810185185185
2016-01-01 00:03:29,0.1108680555555555
2016-01-01 00:04:06,0.016967592592592593
2016-01-01 00:04:37,0.1597222222222222
2016-01-01 00:04:56,2.996585648148148
```

Step 1: Read the data

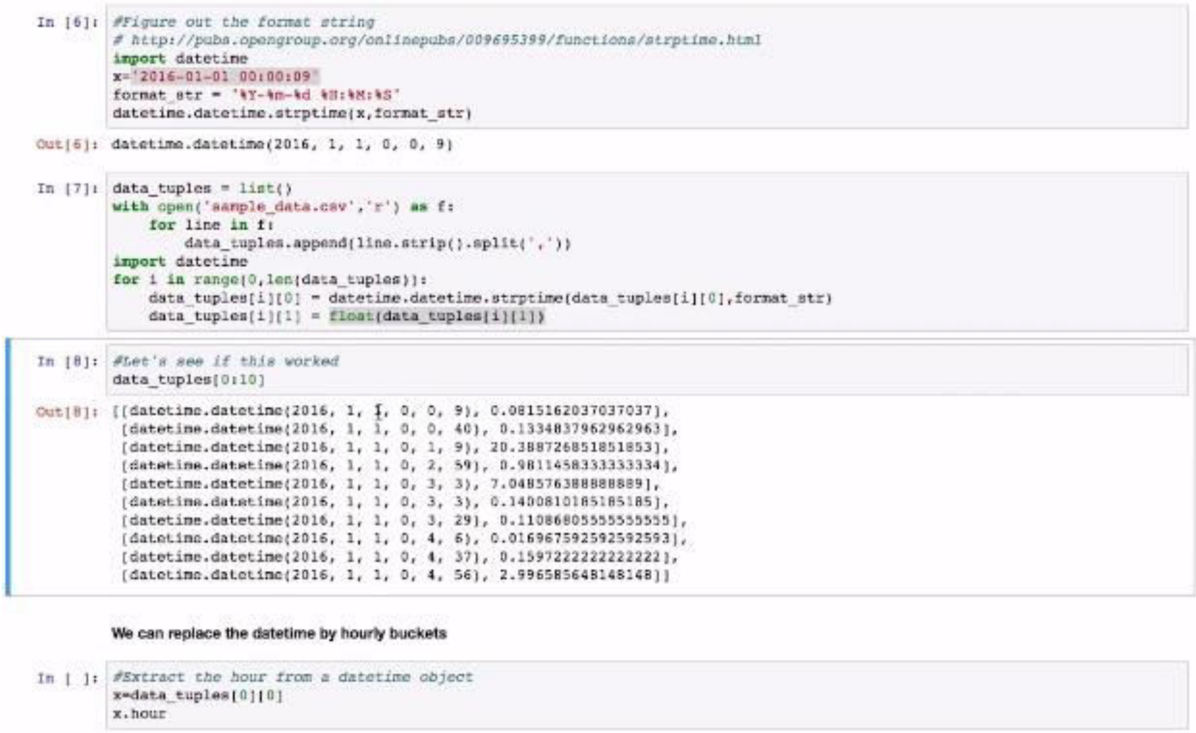
```
In [ ]: data_tuples = list()
        with open('sample_data.csv','r') as f:
            for line in f:
                data_tuples.append(line.strip().split(','))
```

Let's look at the first 10 lines

```
In [ ]: data_tuples[0:10]
```

- Element 1 of the tuple is a date inside a string
- Element 2 is double inside a string

- Let's convert them



```
In [6]: #Figure out the format string
        # http://pubs.opengroup.org/onlinepubs/009695399/functions/strptime.html
        import datetime
        x='2016-01-01 00:00:09'
        format_str = '%Y-%m-%d %H:%M:%S'
        datetime.datetime.strptime(x,format_str)

Out[6]: datetime.datetime(2016, 1, 1, 0, 0, 9)
```

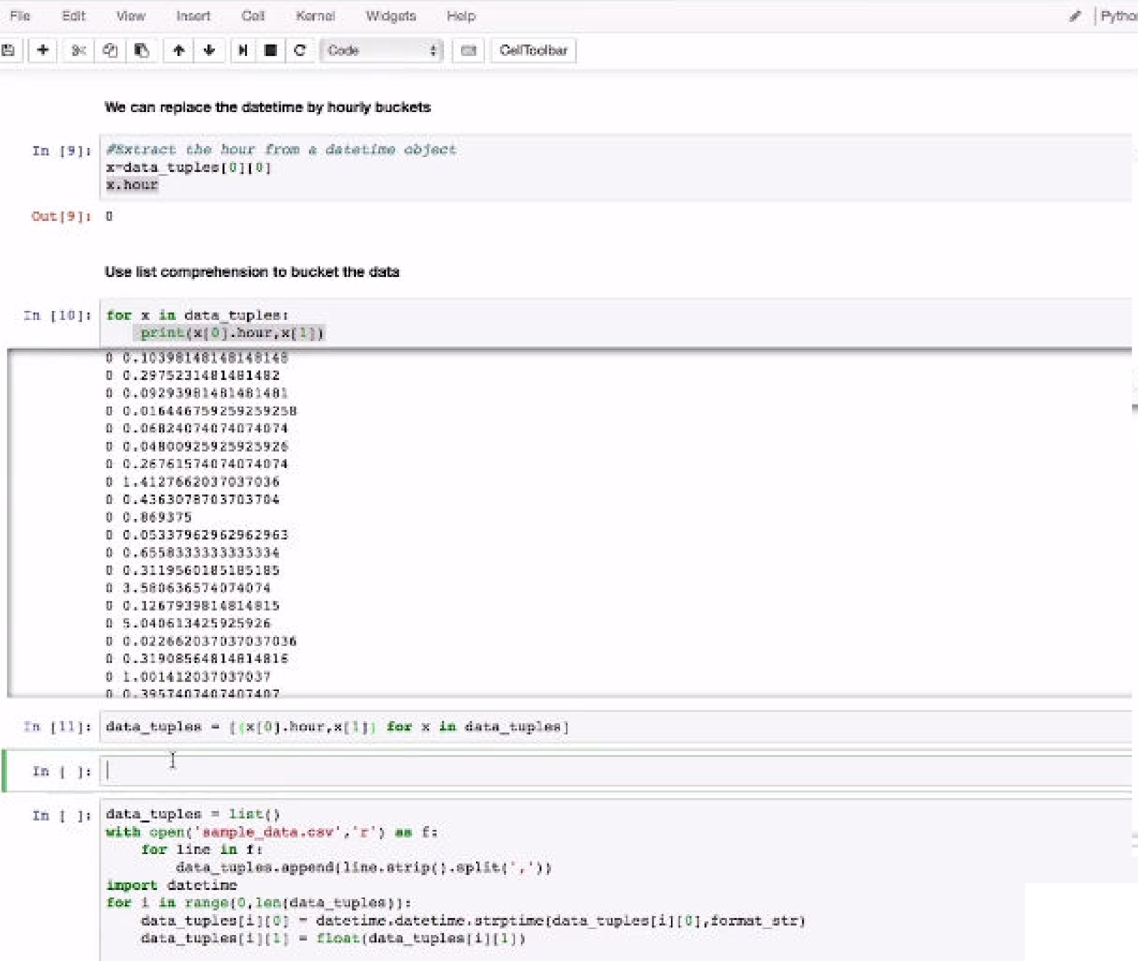
```
In [7]: data_tuples = list()
        with open('sample_data.csv','r') as f:
            for line in f:
                data_tuples.append(line.strip().split(','))
        import datetime
        for i in range(0,len(data_tuples)):
            data_tuples[i][0] = datetime.datetime.strptime(data_tuples[i][0],format_str)
            data_tuples[i][1] = float(data_tuples[i][1])
```

```
In [8]: #Let's see if this worked
        data_tuples[0:10]

Out[8]: [[datetime.datetime(2016, 1, 1, 0, 0, 9), 0.0815162037037037],
         [datetime.datetime(2016, 1, 1, 0, 0, 40), 0.1334837962962963],
         [datetime.datetime(2016, 1, 1, 0, 1, 9), 20.388726851851853],
         [datetime.datetime(2016, 1, 1, 0, 2, 59), 0.9811458333333334],
         [datetime.datetime(2016, 1, 1, 0, 3, 3), 7.048576388888889],
         [datetime.datetime(2016, 1, 1, 0, 3, 3), 0.1400810185185185],
         [datetime.datetime(2016, 1, 1, 0, 3, 29), 0.1108680555555555],
         [datetime.datetime(2016, 1, 1, 0, 4, 6), 0.016967592592592593],
         [datetime.datetime(2016, 1, 1, 0, 4, 37), 0.1597222222222222],
         [datetime.datetime(2016, 1, 1, 0, 4, 56), 2.996585648148148]]
```

We can replace the datetime by hourly buckets

```
In [ ]: #Extract the hour from a datetime object
        x=data_tuples[0][0]
        x.hour
```

Bucketing Time Part 2

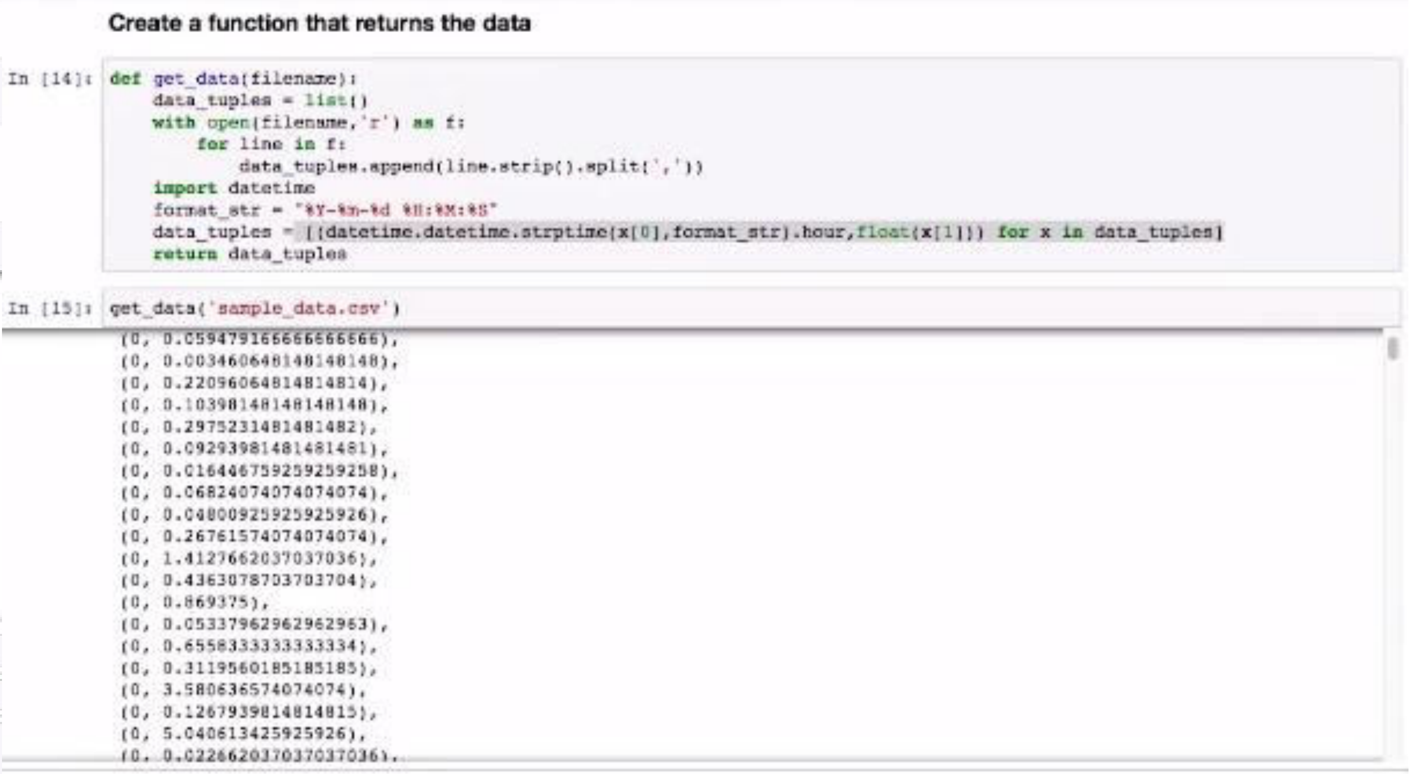


We can replace the datetime by hourly buckets

```
In [9]: #Extract the hour from a datetime object
        x=data_tuples[0][0]
        x.hour

Out[9]: 0
```

Use list comprehension to bucket the data

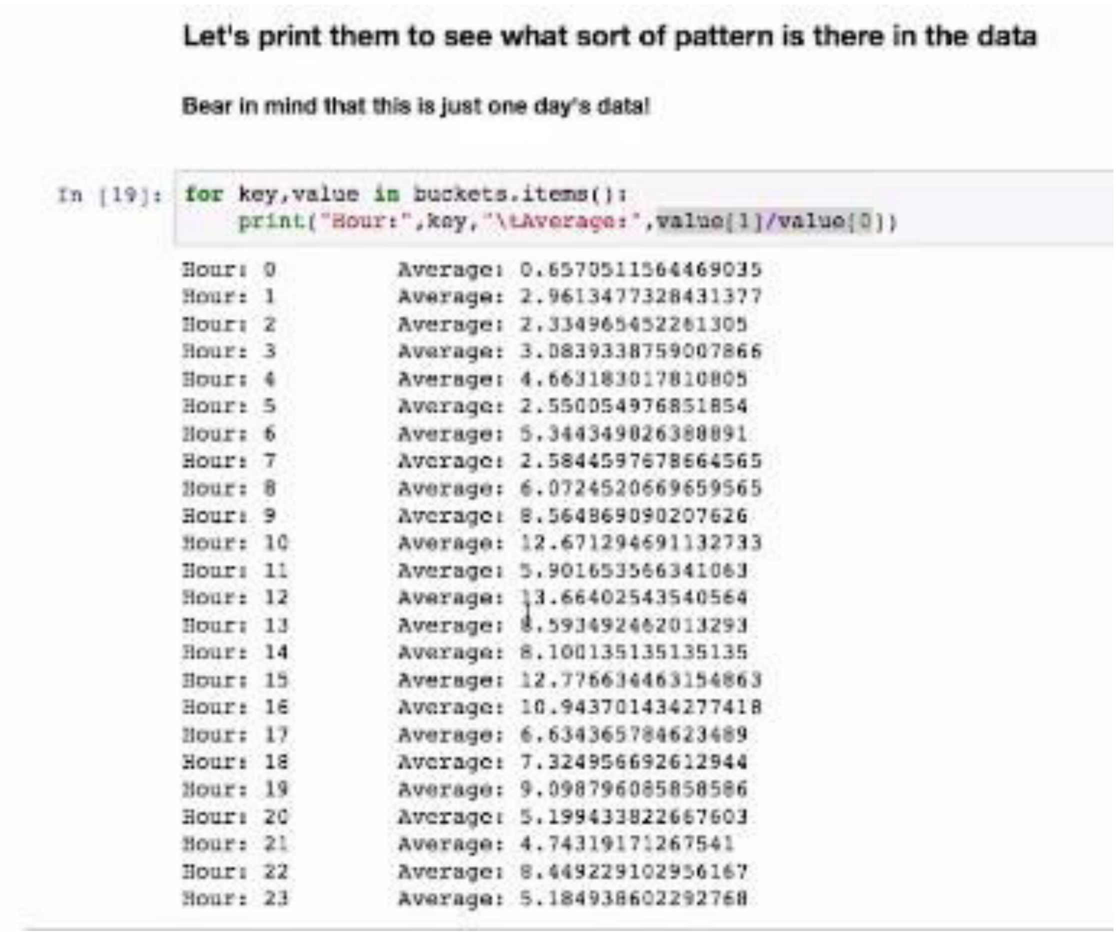
```
In [10]: for x in data_tuples:
             print(x[0].hour,x[1])

0 0.10398148148148148
0 0.2975231481481482
0 0.09293981481481481
0 0.016446759259259258
0 0.06824074074074074
0 0.04800925925925926
0 0.26761574074074074
0 1.4127662037037036
0 0.4363078703703704
0 0.869375
0 0.05337962962962963
0 0.6558333333333334
0 0.3119560185185185
0 3.580636574074074
0 0.1267939814814815
0 5.040613425925926
0 0.022662037037037036
0 0.3190856481481816
0 1.001412037037037
0 0.3957407407407407

In [11]: data_tuples = [(x[0].hour,x[1]) for x in data_tuples]

In [ ]: 

In [ ]: data_tuples = list()
        with open('sample_data.csv','r') as f:
            for line in f:
                data_tuples.append(line.strip().split(','))
        import datetime
        for i in range(0,len(data_tuples)):
            data_tuples[i][0] = datetime.datetime.strptime(data_tuples[i][0],format_str)
            data_tuples[i][1] = float(data_tuples[i][1])
```



Create a function that returns the data

```
In [14]: def get_data(filename):
             data_tuples = list()
             with open(filename,'r') as f:
                 for line in f:
                     data_tuples.append(line.strip().split(','))
             import datetime
             format_str = "%Y-%m-%d %H:%M:%S"
             data_tuples = [(datetime.datetime.strptime(x[0],format_str).hour,float(x[1])) for x in data_tuples]
             return data_tuples

In [15]: get_data('sample_data.csv')

 (0, 0.059479166666666666),
 (0, 0.0034606481481481482),
 (0, 0.22096064814814814),
 (0, 0.10398148148148148),
 (0, 0.2975231481481482),
 (0, 0.09293981481481481),
 (0, 0.016446759259259258),
 (0, 0.06824074074074074),
 (0, 0.04800925925925926),
 (0, 0.26761574074074074),
 (0, 1.4127662037037036),
 (0, 0.4363078703703704),
 (0, 0.869375),
 (0, 0.05337962962962963),
 (0, 0.6558333333333334),
 (0, 0.3119560185185185),
 (0, 3.580636574074074),
 (0, 0.1267939814814815),
 (0, 5.040613425925926),
 (0, 0.022662037037037036),
```

Bucketing Time Part 3

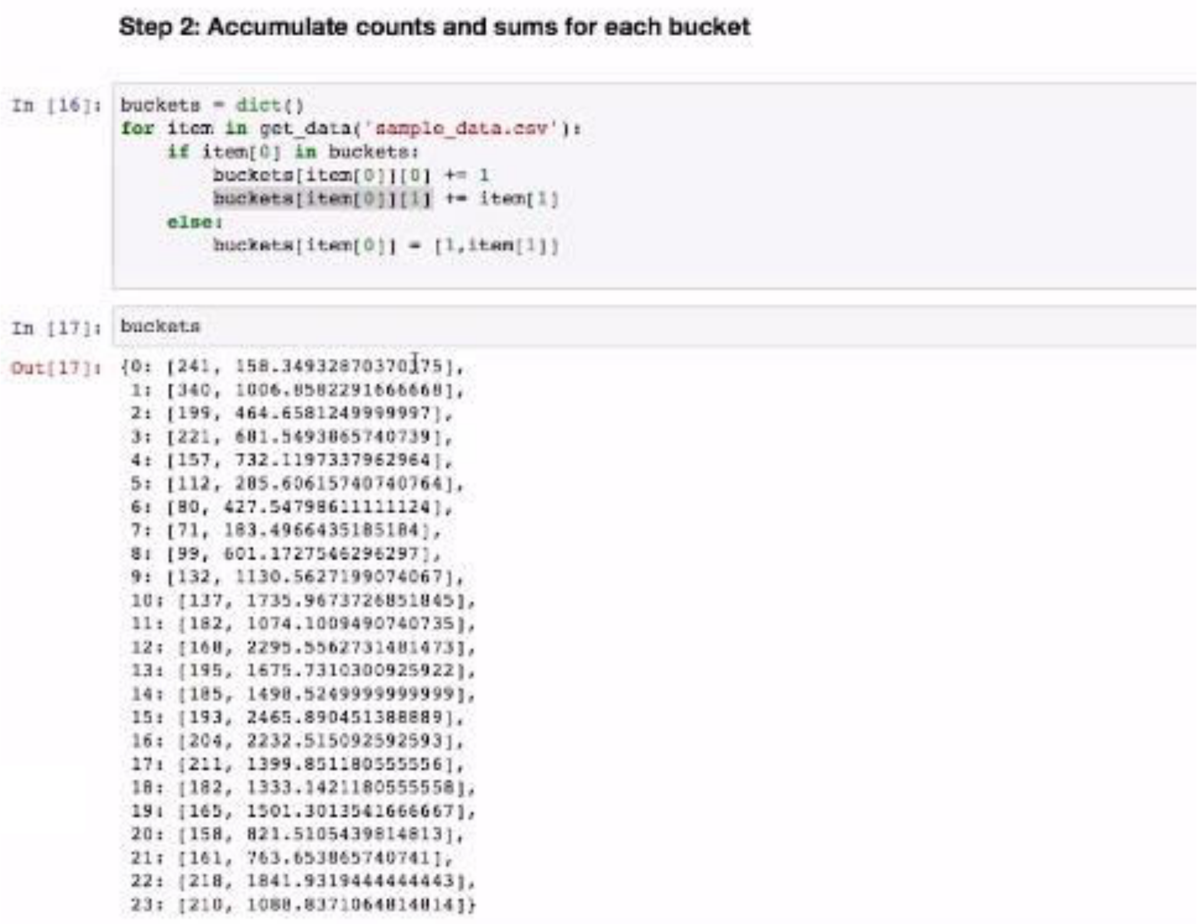


Let's print them to see what sort of pattern is there in the data

Bear in mind that this is just one day's data!

```
In [19]: for key,value in buckets.items():
             print("Hour:",key,"\tAverage:",value[1]/value[0])
```

```
Hour: 0         Average: 0.6570511564469035
Hour: 1         Average: 2.9613477328431377
Hour: 2         Average: 2.334965452261305
Hour: 3         Average: 3.0839338759007866
Hour: 4         Average: 4.663183017810805
Hour: 5         Average: 2.550054976851854
Hour: 6         Average: 5.344349026388891
Hour: 7         Average: 2.5844597678664565
Hour: 8         Average: 6.0724520669659565
Hour: 9         Average: 8.564869090207626
Hour: 10        Average: 12.671294691132733
Hour: 11        Average: 5.901653566341063
Hour: 12        Average: 13.66402543540564
Hour: 13        Average: 8.593492462013293
Hour: 14        Average: 8.100135135135135
Hour: 15        Average: 12.776634463154863
Hour: 16        Average: 10.943701434277418
Hour: 17        Average: 6.634365784623489
Hour: 18        Average: 7.324956692612944
Hour: 19        Average: 9.098796085858586
Hour: 20        Average: 5.199433822667603
Hour: 21        Average: 4.74319171267541
Hour: 22        Average: 8.449229102956167
Hour: 23        Average: 5.184938602292768
```



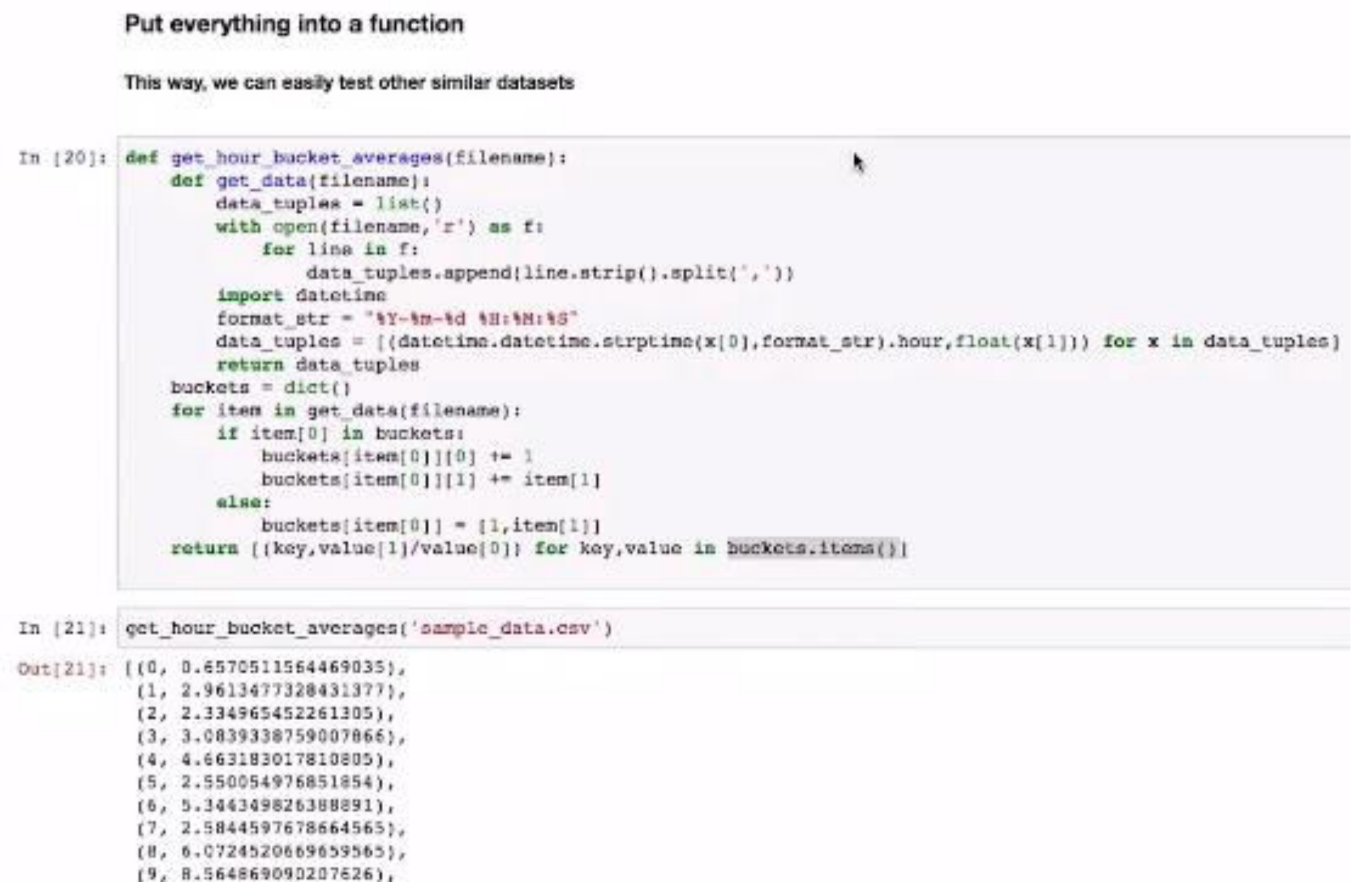
Step 2: Accumulate counts and sums for each bucket

```
In [16]: buckets = dict()
         for item in get_data('sample_data.csv'):
             if item[0] in buckets:
                 buckets[item[0]][0] += 1
                 buckets[item[0]][1] += item[1]
             else:
                 buckets[item[0]] = [1,item[1]]
```

```
In [17]: buckets
```

```
Out[17]: {0: [241, 158.34932870370375],
          1: [340, 1006.8582291666668],
          2: [199, 464.6581249999997],
          3: [221, 681.5493865740739],
          4: [157, 732.1197337962964],
          5: [112, 285.6061574074076 4],
          6: [80, 427.5479861111124],
          7: [71, 183.4966435185184],
          8: [99, 601.1727546296297],
          9: [132, 1130.5627199074067],
          10: [137, 1735.9673726851845],
          11: [182, 1074.1009490740735],
          12: [168, 2295.5562731481473],
          13: [195, 1675.7310300925922],
          14: [185, 1498.5249999999999],
          15: [193, 2465.890451388889],
          16: [204, 2232.515092592593],
          17: [211, 1399.851180555556],
          18: [182, 1333.1421180555558],
          19: [165, 1501.3013541666667],
          20: [158, 821.5105439814813],
          21: [161, 763.653865740741],
          22: [218, 1841.9319444444443],
          23: [210, 1088.8371064814814]}
```

Bucketing Time Part 3

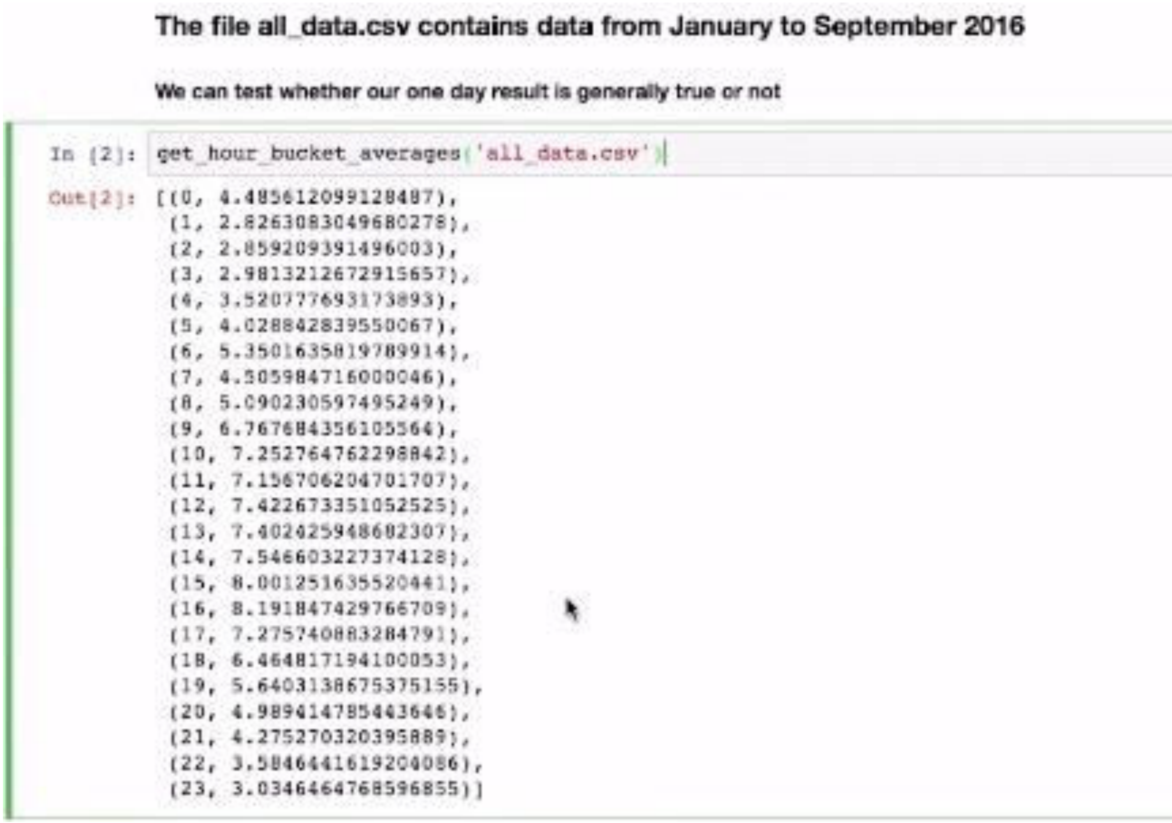


Put everything into a function

This way, we can easily test other similar datasets

```
In [20]: def get_hour_bucket_averages(filename):
             def get_data(filename):
                 data_tuples = list()
                 with open(filename,'r') as f:
                     for line in f:
                         data_tuples.append(line.strip().split(','))
                 import datetime
                 format_str = "%Y-%m-%d %H:%M:%S"
                 data_tuples = [(datetime.datetime.strptime(x[0],format_str).hour,float(x[1])) for x in data_tuples]
                 return data_tuples
             buckets = dict()
             for item in get_data(filename):
                 if item[0] in buckets:
                     buckets[item[0]][0] += 1
                     buckets[item[0]][1] += item[1]
                 else:
                     buckets[item[0]] = [1,item[1]]
             return [(key,value[1]/value[0]) for key,value in buckets.items()]
```

```
In [21]: get_hour_bucket_averages('sample_data.csv')

Out[21]: [(0, 0.6570511564469035),
          (1, 2.9613477328431377),
          (2, 2.334965452261305),
          (3, 3.0839338759007866),
          (4, 4.663183017810805),
          (5, 2.550054976851854),
          (6, 5.344349826388891),
          (7, 2.5844597678664565),
          (8, 6.0724520669659565),
          (9, 8.564869090207626),
```



The file all_data.csv contains data from January to September 2016

We can test whether our one day result is generally true or not

```
In [2]: get_hour_bucket_averages('all_data.csv')

Out[2]: [(0, 4.485612099128487),
         (1, 2.826308304968027B),
         (2, 2.859209391496003),
         (3, 2.9813212672915657),
         (4, 3.520777693173893),
         (5, 4.028842839550067),
         (6, 5.350163581978991 4),
         (7, 4.505984716000046),
         (8, 5.090230597495249),
         (9, 6.767684356105564),
         (10, 7.252764762298842),
         (11, 7.156706204701707),
         (12, 7.422673351052525),
         (13, 7.402425948682307),
         (14, 7.546603227374128),
         (15, 8.001251635520441),
         (16, 8.191847429766709),
         (17, 7.275740883284791),
         (18, 6.464817194100053),
         (19, 5.640313867537515 5),
         (20, 4.989414785443646),
         (21, 4.275270320395889),
         (22, 3.584644161920408 6),
         (23, 3.0346464768596855)]
```

GLOBAL IVY EDUCATION

www.emeritus.org