DS1 Lecture 01 supplement

Jim Bagrow

This notebook acts as an additional supplement on programming in Python. You are expected to learn this material during the first week of class, and the first quiz of the semester will assess this.

This document complements the course's **"Whirlwind Tour" reading**. You are also expected to complete this reading assignment. I recommend you review this supplement after reading the whirlwind tour.

About this document

- This is a **Jupyter Notebook**. I'll be using these for code-heavy lectures and lecture supplements. Notebooks contain IPython-style input and output cells in a document, interleaved with narrative (non-code) text.
- Your **projects and assignments** will be submitted as Python scripts (`.py` files) with associated write-ups (`.docx` files).

Contents:

Tuples vs. Lists

Lists and tuples are two array data structures built into Python.

[1]:
```
L = [0,1,2]
T = (0,1,2)
```

Notice that lists are delineated with square brackets while tuples uses parentheses.

Why have both array types? The main difference between the two is that a list is **mutable** while a tuple is **immutable**.

Once a list is made you can update it in place:

[2]:
```
L = [0,1,2,3,4]
print(L)
print("Hi Bob")
L.append('a')
print(L)
```

```
[0, 1, 2, 3, 4]
Hi Bob
[0, 1, 2, 3, 4, 'a']
```

- Recall using IPython to explore the methods associated with lists.

But something that can change can't be used as a key in a dictionary:

[3]:
```
social_network = {}
link = ['John','Paul']
social_network[link] = 'are friends'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-9dd97218a62a> in <module>
      1 social_network = {}
      2 link = ['John','Paul']
----> 3 social_network[link] = 'are friends'

TypeError: unhashable type: 'list'
```

This error message (see the Whirlwind Tour for details on how to read Python error messages) tells us that lists cannot be used as a dictionary key because the computation needed to determine where the key will sit in memory (called hashing) will not be done on a list.

Because a list can change, once you have put it into the dictionary, you could later change the contents of the list (for example, using `append`), which would then cause the dictionary's internal data structure to lose track of the key within your memory. To avoid the complexities of tracking mutable objects, Python dictionaries simply forbid this operation. A benefit you get from this however, is that Python dictionaries are incredibly fast at key lookups.

But what if I want to use an array of data as a key? Am I stuck?

Instead of a list, just use a **tuple**:

[4]:
```
L = ['John','Paul']
T = ('John','Paul') # looks almost the same!
```

`T` works almost exactly like `L` except you can't change it after it's been created, it's **immutable**:

[5]:
```
print("The first element of T is", T[0])
L[0] = "Ringo" # change L
T[0] = "Ringo" # can't change T
```

```
The first element of T is John
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-fe2de7bd000d> in <module>
      1 print("The first element of T is", T[0])
      2 L[0] = "Ringo" # change L
----> 3 T[0] = "Ringo" # can't change T

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

So we can use tuples for our dictionary keys!

[6]:
```
social_network[ ('John', 'Paul') ] = "are friends"
print(social_network)
```

```
{('John', 'Paul'): 'are friends'}
```

Note in this case that the order of the tuple matters:

[7]:
```
print(("Paul", "John") in social_network)
print(("John", "Paul") in social_network)
```

```
False
True
```

- Aside: look how easy it is to test key membership: `K in D` is a simple boolean statement

[8]:
```
social_network["Paul", "John"] = social_network["John", "Paul"]
print(social_network)
```

```
{('John', 'Paul'): 'are friends', ('Paul', 'John'): 'are friends'}
```

There are some other small differences between tuples and lists, so *be careful*. For example:

[9]:
```
print(type( [5]  )) # <type 'list'>
print(type( 5    )) # <type 'int'>
print(type( (5)  )) # <type 'int'>
print(type( (5,) )) # <type 'tuple'>
```

```
<class 'list'>
<class 'int'>
<class 'int'>
<class 'tuple'>
```

Multi-assignment

Functions that return multiple values always return tuples:

[10]:
```
def swap(a,b):
    return b,a

output = swap("George","Ringo")
print(type(output))
```

```
<class 'tuple'>
```

Python's "=" understands tuples and lists in a very convenient way:

[11]:
```
coordinate = (0.9,0.1,0.5)
x,y,z = coordinate # same as (x,y,z) = coordinate

# much shorter than:
x = coordinate[0]
y = coordinate[1]
z = coordinate[2]
```

Combine multi-assignment with for-loops for short, **readable** code:

[13]:
```
colors = [ (255,0,128), (192,64,8), (128,128,128) ]

for r,g,b in colors:
    print("The red and blue channels are {} and {}" .format(r,b))
```

```
The red and blue channels are 255 and 128
The red and blue channels are 192 and 8
The red and blue channels are 128 and 128
```

- This example works exactly the same using a **list-of-lists** instead of a **list-of-tuples**

Compare that with using numeric indexing:

[14]:
```
N = len(colors) # length
for i in range(N): # range(N) = [0, 1, ..., N-1]
    r = colors[i][0]
    g = colors[i][1]
    b = colors[i][2]
    print("The red and blue channels are {} and {}".format(r,b))

print("*** too much bookkeeping :( ***")
```

```
The red and blue channels are 255 and 128
The red and blue channels are 192 and 8
The red and blue channels are 128 and 128
*** too much bookkeeping :( ***
```

Notice how much more compact and readable the code is when using multi-assignment.

Sorting

Something we will be doing at times is building up collections of data and then ordering them in certain ways.

Here's a tiny example:

[15]:
```
values = [52,66,41,28]
print(min(values), max(values))
```

```
28 66
```

Suppose we need to sort `values`. Googling brings up two options:

```
values.sort()
sorted(values)
```

What's the difference?

[16]:
```
values2 = [52,66,41,28].sort()
values3 = sorted( [52,66,41,28] )
print("Are they the same?", values2 == values3)
print("The .sort() result is", values2)
```

```
Are they the same? False
The .sort() result is None
```

What happened? Why is `values2` None?

Answer:

- `.sort()` modifies a list **in place**
- `sorted()` returns a sorted **copy**

[17]:
```
print(values)
values.sort()
```

```
print(values)
```

```
[52, 66, 41, 28]
[28, 41, 52, 66]
```

More sorting

Here's something more realistic.

We have social network data, and we've computed the **number of friends** each user has:

[18]:
```
name_numFriends = [ ('Paul',64), ('Ringo',16),
                    ('George',32),('John',128) ]
```

Now let's find out who is *most popular*. We can build a sorted copy of this list:

[19]:
```
L = sorted(name_numFriends) # L is a bad variable name!
print(L)
```

```
[('George', 32), ('John', 128), ('Paul', 64), ('Ringo', 16)]
```

Oops! We weren't careful with the phrase `sorted` and the computer sorted the list *alphabetically*

We want to sort by number of friends, so let's try this:

[20]:
```
numFriends_name = []
for name,num in name_numFriends:
    numFriends_name.append( (num,name) ) # swapped!

# now sort:
nF_name_sorted = sorted(numFriends_name)
nF_name_sorted_reversed = sorted(numFriends_name,reverse=True) # !

print(numFriends_name)
print(nF_name_sorted)
print(nF_name_sorted_reversed)
```

```
[(64, 'Paul'), (16, 'Ringo'), (32, 'George'), (128, 'John')]
[(16, 'Ringo'), (32, 'George'), (64, 'Paul'), (128, 'John')]
[(128, 'John'), (64, 'Paul'), (32, 'George'), (16, 'Ringo')]
```

We've got it, let's just unswap the elements of the list:

[21]:
```
name_numFriends_sorted = []
for num,name in nF_name_sorted_reversed:
    name_numFriends_sorted.append( (name,num) ) # swapped back!
```

This process is called **Decorate-Sort-Undecorate**. (Python has other goodies to do this as well.)

Zipping

We're seeing a lot of lists of tuples:

`L = [('a',7.60),('b',8.10), ... ]`

But often we'll be stuck with **separate lists** we want to combine:

[22]:
```
names = ['Paul', 'Ringo', 'George', 'John']
counts = [64, 16, 32,128]
```

```
print(list(zip(names,counts))) # bingo!
```

```
[('Paul', 64), ('Ringo', 16), ('George', 32), ('John', 128)]
```

Note that we needed to convert zip to a list for printing because by default it return an **iterator**, a Python function that builds the list while you loop over it (see the Whirlwind Tour for details).

[23]:
```
L = [ ('Paul',64), ('Ringo',16), ('George',32),('John',128) ]

X,Y = zip(*L) # bizarre!

print(X, type(X))
print(Y)
```

```
('Paul', 'Ringo', 'George', 'John') <class 'tuple'>
(64, 16, 32, 128)
```

Comprehensions

Previously you saw we had to build modified lists:

```
new_list = []
for x in old_list:
    new_list.append( f(x) )
```

There's a shorter (and faster!) way to do this:

```
new_list = [f(x) for x in old_list]
```

May look a little weird at first. (And why not use `map`?)

Let's do the decorate-sort-undecorate again, using these **list comprehensions**:

[24]:
```
name_numFriends = [ ('Paul',64), ('Ringo',16),
                    ('George',32),('John',128) ]

# decorate (swap):
numFriends_name = [ (num,name) for name,num in name_numFriends ]
# sort:
numFriends_name.sort(reverse=True)
# undecorate:
name_numFriends = [ (name,num) for num,name in numFriends_name ]

print(name_numFriends)
```

```
[('John', 128), ('Paul', 64), ('George', 32), ('Ringo', 16)]
```

All those loops are replaced. It may take some practice to read this style, but you're likely to see these beasts in practice...

Lists, dicts, sets, oh my! There are also dict and set comprehensions. Set comprehensions match our normal mathematical set notation very well.

[25]:
```
S = { t for t in range(100) if 10 <= t <= 50 } # curly braces!

print(S, "&", type(S))
```

```
{10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
50} & <class 'set'>
```

Working with files

Reading and writing data is one of the most important, and most common things we will do. For now let's start off with plaintext files.

Reading `data.txt` Suppose I have a *DAQ card* and I'm collecting `voltage` and `current` as functions of `time`. A simple way to store this information is a text file, where each line of the file records one data point:

```
# time(sec) voltage current
0.0 3.58066793237 0.269466805146
0.0942477 2.35761842277 -0.0343609063213
0.1884954 2.54291875337 0.402181644387
0.2827431 3.01547024563 -0.00663739540361
. . .
. . .
. . .
```

A nice, simple $N \times 3$ matrix, stored as **space-separated** text with a one-line header row.

Let's read this into Python and print to the screen. The for-loop works nicely with files, using the `open` command:

[26]:
```
# python for loops can read a file line-by-line automatically!
i = 0
for line in open("data.txt"): # this file does not have a header row
    print(line)

    i += 1
    if i > 8: # suppress long output
        print("...")
        break
```

```
0.0 3.58066793237 0.269466805146

0.0942477 2.35761842277 -0.0343609063213

0.1884954 2.54291875337 0.402181644387

0.2827431 3.01547024563 -0.00663739540361

0.3769908 1.97720698289 0.214729259821

0.4712385 3.08988264269 0.450300612049

0.5654862 2.55471487263 0.217870513189

0.6597339 2.78877808088 0.792510286325

0.7539816 2.63650949635 0.872919361726

...
```

Ok, are there supposed to be **blank lines**? Let's try again:

[27]:
```
i = 0
for line in open("data.txt"):
    print(repr(line)) # repr!!!

    i += 1 # suppress long output
    if i > 8:
        print("...")
        break
```

```
'0.0 3.58066793237 0.269466805146\n'
'0.0942477 2.35761842277 -0.0343609063213\n'
'0.1884954 2.54291875337 0.402181644387\n'
'0.2827431 3.01547024563 -0.00663739540361\n'
'0.3769908 1.97720698289 0.214729259821\n'
'0.4712385 3.08988264269 0.450300612049\n'
'0.5654862 2.55471487263 0.217870513189\n'
'0.6597339 2.78877808088 0.792510286325\n'
'0.7539816 2.63650949635 0.872919361726\n'
...
```

Ah! Each time through the loop the variable `line` becomes a string representing each **line** of the text file. This string includes the **newline** character at the end.

- We can strip away *whitespace* at the ends of a string easily with `.strip()`
- Let's also break each line into a list, using `.split()`

[28]:
```
i = 0
for line in open("data.txt"):
    # make new string without newlines, then split that
    # string into a list:
    print(line.strip().split())

    i += 1
    if i > 8:
        print("...")
        break
```

```
['0.0', '3.58066793237', '0.269466805146']
['0.0942477', '2.35761842277', '-0.0343609063213']
['0.1884954', '2.54291875337', '0.402181644387']
['0.2827431', '3.01547024563', '-0.00663739540361']
['0.3769908', '1.97720698289', '0.214729259821']
['0.4712385', '3.08988264269', '0.450300612049']
['0.5654862', '2.55471487263', '0.217870513189']
['0.6597339', '2.78877808088', '0.792510286325']
['0.7539816', '2.63650949635', '0.872919361726']
...
```

Aside: Method chaining:

Just now I used `line.strip().split()`. That may look a little weird if you haven't see it before.

- Recall: `line` is a string. It has a method called `strip` which returns a new string that is a copy of the old string with any leading/trailing whitespace removed. Strings also have a method called `split` which turns them into lists.
- This can be written as:

```
a_string = line.strip()
a_list = a_string.split()
```

And all I've done above make this more compact: I've eliminated `a_string` on the second line by exactly replacing it with `line.strip()`, its definition on the first line: `line.strip().split()` ***

Combine with multi-assignment:

[29]:
```
i = 0
for line in open("data.txt"):
    time, volt, curr = line.strip().split()

    print("t = {}, c = {}".format(time,curr))

    i += 1
    if i > 8:
        print("...")
        break
```

```
t = 0.0, c = 0.269466805146
t = 0.0942477, c = -0.0343609063213
t = 0.1884954, c = 0.402181644387
t = 0.2827431, c = -0.00663739540361
t = 0.3769908, c = 0.214729259821
t = 0.4712385, c = 0.450300612049
t = 0.5654862, c = 0.217870513189
t = 0.6597339, c = 0.792510286325
t = 0.7539816, c = 0.872919361726
...
```

So we can quickly make variables for the line, but the variables are **strings**. We can't do:

[30]:
```
for line in open("data.txt"):
    time,volt,curr = line.strip().split()

    R = volt / curr
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-30-d5ffb645418f> in <module>
      2     time,volt,curr = line.strip().split()
      3
----> 4     R = volt / curr

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

No problem, just **convert** the strings to floats:

[31]:
```
list_times = []
list_resistances = []
```

```
for line in open("data.txt"):
    time,volt,curr = line.strip().split()
    time = float(time)
    volt = float(volt)
    curr = float(curr)

    list_times.append(time)
    list_resistances.append(volt/curr)
```

Writing `data.txt` Suppose our big project was just to compute these resistances from our experiment and then record them to another file.

To do this we :

1. Open a file for writing
2. Write **strings** to it using `.write()`
3. Close the file when finished using `.close()` [*important!*]

[32]:
```
# open a new file to write to:
fout = open("data_res.txt", 'w') # w = write

for t,r in zip(list_times,list_resistances): # zip() lets us loop over two things at␣
 ↪once!
    # t and r are floats, need to make them strings first
    # and we need to take care of newlines ourselves:

    line_out = str(t) + " " + str(r) + "\n"
    #                   ^^^ space-separating *delimiter*
    line_out = "{} {}\n".format(t,r) # same, but shorter

    fout.write(line_out)

# close the file to be safe:
fout.close()
```

OK, so dealing with files is a bit of a pain because: * we are working directly with strings * managing the variable types ourselves.

The above was a teaching example. Later I'll show you how to read and write whole files of regular, **matrix** data with a **single command**!

But what about reading a "ragged" text file? For example, a file where each line records (1) a person and (2) a list of that person's friends:

```
Alice Bob Jack Kerry
Bob Alice Jack Peter Frank Megan Nancy
Jack Alice Bob Andrea
...
```

This is a compact data format, and it's very convenient in python:

```
for line in open(filename):
    L = line.strip().split()
    user = L[0]
    friends = L[1:]
```

Try doing **that** in C or MATLAB!

By the way, if we had commas separating the values instead of spaces, we could just use `line.strip().split(",")`. (Python has a CSV module as well, batteries included!)

Exploring the standard library

Python's philosophy of "batteries included" means that lots of useful functionality is built into Python by default. Most of this is organized in the *standard library*, a large collection of modules that can be imported from any Python code.

For example, there are a number of math functions we can use:

[33]:
```
import math

print(2*math.pi)

print(math.log(2.7182))

print(math.ceil(2.4))
```

```
6.283185307179586
0.9999698965391098
3
```

Question: Why would we want to write `math.log`? Isn't that way more work and less readable than just writing `log` or `ln`?

Answer: Using import statements allows scripts to manage what code they require and avoids **namespace collisions**. A log file `log = open("f.txt", 'w')` can be distinguished from the logarithm function (`math.log`) by keeping the log function inside the math module.

A word about importing OK, `import math` creates a new module called `math` containing all the math goods. But there are some useful *variations* for import statements:

[34]:
```
 # rename the math module:
import math as m

print(m.sin(3.14))


# bring in functions but not the module:
from math import log, ceil

print(log(10,10))

log = "system.log" # gotcha, we just killed the math function


# bring in everything:
from math import *
```

```
0.0015926529164868282
1.0
```

There are many other libraries and packages we'll be using, even built-in. They'll be showing up in homeworks soon!

Third-Party packages

Beyond the standard library that all Python code can use, there is also a **rich ecosystem** of incredibly useful, popular libraries that are not made by Python.org, but are easy to install (and often come by default in distributions of Python such as Anaconda Python). We'll be seeing these again.

Numpy

Numpy (Numeric Python) provides a very useful data structure specialized for numeric data, as well as a number of other great functions.

Suppose we want 11 floats evenly spaced between zero and one. We need to do something like:

[35]:
```
numbers_py = []
for r in range(0,10+1):
    numbers_py.append(r/10.0)

print(numbers_py, type(numbers_py))
```

```
[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0] <class 'list'>
```

Numpy lets us skip the bookkeeping:

[36]:
```
import numpy as np # standard practice to rename it to `np`

numbers_np = np.linspace(0,1,11)
print(numbers_np, type(numbers_np))
```

```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ] <class 'numpy.ndarray'>
```

This variable created by numpy does not behave like a python list. It behaves like a MATLAB vector:

[37]:
```
print(numbers_py*2) # concatenate numbers_py to itself
print()
print(numbers_np*2) # multiplied each element by 2
```

```
[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 0.0, 0.1, 0.2, 0.3, 0.4,
0.5, 0.6, 0.7, 0.8, 0.9, 1.0]

[0.  0.2 0.4 0.6 0.8 1.  1.2 1.4 1.6 1.8 2. ]
```

Both lists and ndarrays are very handy, for different things. Choosing the right one can make your life much easier:

- Remember the voltage/current data file we read. That was **purely numeric**, so it's a good candidate for numpy:

[38]:
```
M = np.loadtxt("data.txt")
print(M[:12,:]) # print first 12 rows...
```

```
[[ 0.          3.58066793  0.26946681]
 [ 0.0942477   2.35761842 -0.03436091]
 [ 0.1884954   2.54291875  0.40218164]
 [ 0.2827431   3.01547025 -0.0066374 ]
 [ 0.3769908   1.97720698  0.21472926]
 [ 0.4712385   3.08988264  0.45030061]
 [ 0.5654862   2.55471487  0.21787051]
 [ 0.6597339   2.78877808  0.79251029]
 [ 0.7539816   2.6365095   0.87291936]
```

```
 [ 0.8482293   2.04928684  0.50242701]
 [ 0.942477    2.01675315  0.77564766]
 [ 1.0367247   0.59550182  0.72930271]]
```

`loadtxt` is a great function for helping us read numeric data into a **matrix**. We can take row- and column-slices easily:

[39]:
```
print(M.shape) # numRows, numCols

time = M[:,0] # all rows, column zero
volt = M[:,1] # all rows, column one
curr = M[:,2]
last_datapoint = M[-1,:]

print(last_datapoint)
```

```
(201, 3)
[18.84954    3.9366731 -0.1701687]
```

[40]:
```
time.shape
```

[40]:
```
(201,)
```

Scipy

A very handy package for **scientific computing**. Lots of statistical tools, special functions (`scipy.special.gamma`, `scipy.special.erf`, etc.), signal processing, image analysis, function search and optimization, and more.

We'll be seeing scipy in the future.

Matplotlib

Don't forget, **visualization**. Includes making figures.

- Matplotlib is a powerful (but a bit complicated) library for generating figures. There's a nice gallery worth checking out.

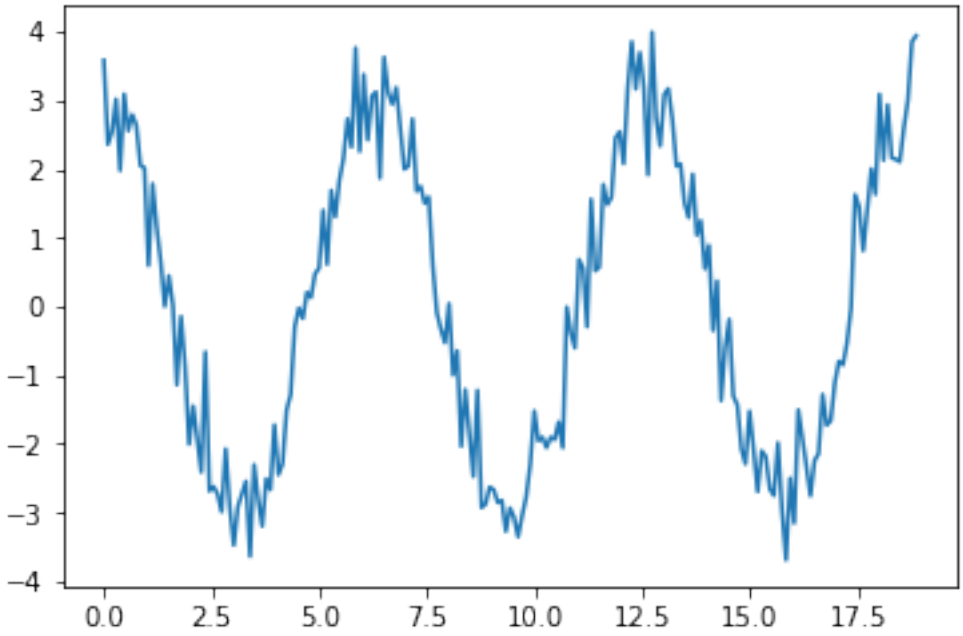
Let's quickly plot our old time, voltage, current data:

[41]:
```
%matplotlib inline
# ^^^ ipython-specific command for putting plots inside the web browser

import matplotlib.pyplot as plt # see, complicated

plt.plot(time,volt) # works with numpy
```

[41]:
```
[<matplotlib.lines.Line2D at 0x7fd3a2ab1d30>]
```

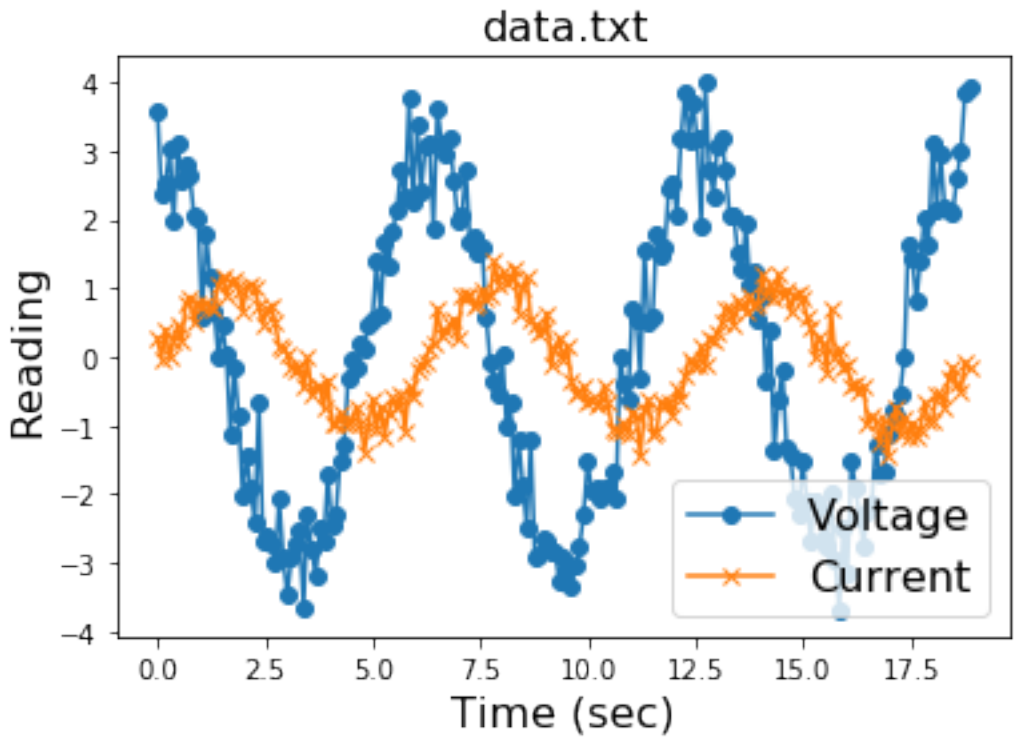


- Notice how easy it is to plot one numpy array against another? Note that you need to make sure the elements of the arrays are in corresponding order, otherwise you will be mixing up the x- and y-coordinates of your points/curve.

[42]:
```
plt.plot(time,volt,'o-')
plt.plot(time,curr,'x-')

plt.xlabel("Time (sec)", fontsize=16)
plt.ylabel("Reading",    fontsize=16)
plt.legend(["Voltage", "Current"], fontsize=16)
plt.title("data.txt", fontsize=16)
```

[42]:
```
Text(0.5, 1.0, 'data.txt')
```



Random stuff

Python has a great random package within the standard library. This lets us do stochastic simulations by giving us a pseudorandom number generator.

The minimum a programming language needs for this is a function that gives you access to random floats between zero and one. Python does this:

[43]:
```
import random

for _ in range(4): # underscore makes a handy "dummy variable"
    print(random.random())
```

```
0.9698541802267313
0.06240885519838868
0.9855331143558425
0.4450682795184209
```

Now, suppose I have a list that I want to make random samples from. Normally you'd code up something like:

[44]:
```
suit = ["hearts", "diamonds", "clubs","spades"]

# sample w/ replacement four times:
for _ in range(4):
    r = random.random()*len(suit) # r in [0,4)
    print( suit[int(r)] )
```

```
spades
clubs
clubs
diamonds
```

But this does not look very pretty. Can we do better? Yep:

[45]:
```
for _ in range(4):
    print(random.choice(suit)) # goodies!
```

```
clubs
hearts
clubs
hearts
```

And don't forget our fancy list comprehensions:

[46]:
```
draws = []
for _ in range(4):
    draws.append(random.choice(suit))
print(draws)
```

```
['hearts', 'hearts', 'hearts', 'diamonds']
```

There are also functions for random integers, normally-distributed random variables, and more.

- Taken together, these additional functions (`choice`, `sample`, etc.) provide a lot of convenience and lead to short, simple and readable code.

Let's talk about two more, `sample` **and** `shuffle`**:** If we run `random.choice` on a list repeatedly we are sampling uniformly from that list **with replacement**. If we want to sample without replacement, meaning the same element of that list cannot be selected more than once, we can use `random.sample`:

[47]:
```
numbers = range(1000)
print(random.sample( numbers, 5)) # make five draws (input must
                                  # have len >= 5)
```

```
[844, 537, 0, 387, 546]
```

[48]:
```
speech = """We choose to go to the moon. We choose to go to
the moon in this decade, not because it easy, but because
it is hard."""

list_words = speech.split() # split on any whitespace and
                            # remove empty strings

print(list_words)
```

```
['We', 'choose', 'to', 'go', 'to', 'the', 'moon.', 'We', 'choose', 'to', 'go',
'to', 'the', 'moon', 'in', 'this', 'decade,', 'not', 'because', 'it', 'easy,',
'but', 'because', 'it', 'is', 'hard.']
```

[49]:
```
random.shuffle(list_words) # shuffle works in place!!!
print(list_words)
```

```
['in', 'this', 'go', 'We', 'to', 'to', 'choose', 'to', 'but', 'to', 'because',
'hard.', 'the', 'moon.', 'moon', 'the', 'go', 'it', 'decade,', 'it', 'We',
'easy,', 'not', 'choose', 'is', 'because']
```

Random sampling and shuffling becomes incredibly useful when getting into monte carlo methods, permutation tests, and many other statistical techniques!

Conclusion:

Don't forget to study the Whirlwind Tour reading assignment. This short supplement here is helpful after completing the Whirlwind Tour.

Expect to be **quizzed** on these and other topics from lectures and to need these topics for the rest of the semester.