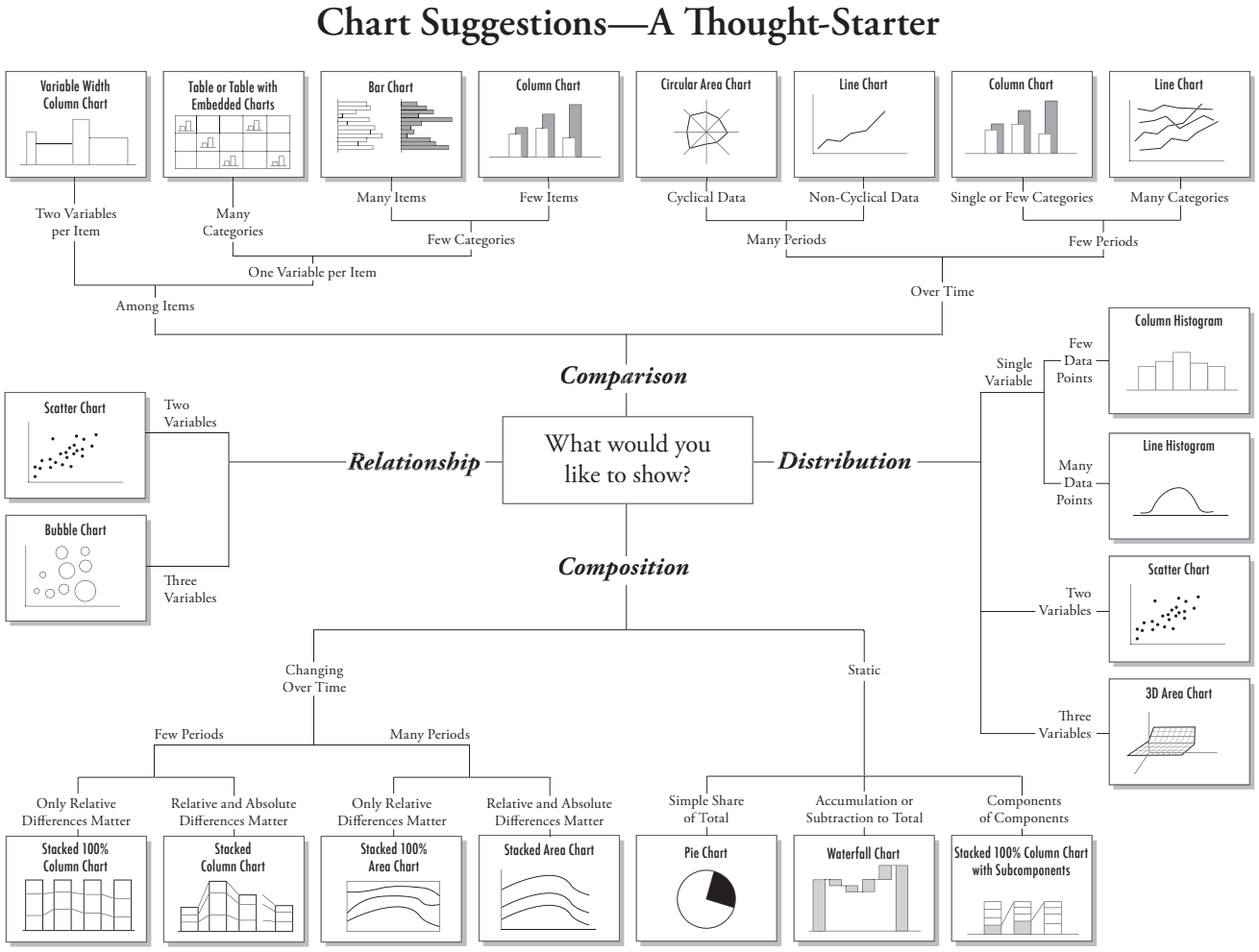
Computational Social Science Lab 2
Data Visualisation with Python

According to an excellent article by Hilary Mason [13] there are 5 aspects to Data Science; Obtain, Scrub, Explore, Model and Interpret. Having pointed you in the right direction in finding data sources and using command line tools to clean data, we move onto exploring data by visualisation.

1 Choosing a Plot Type

Choosing the right way to present your data is extremely important to convince your audience of the strength of your results. There are a number of distinct ways to present your data as this now famous graphic shows [1]. It is worth spending time of developing good, clear plots; as the saying goes 'A picture is worth a thousand words'!



We will cover several different kinds of plots, and their strengths and demonstrate how to use Python, specifically the flexible matplotlib [9] library. You may be used to using MATLAB/Octave or R, so you might be tempted to continue using these languages for plotting. Also, despite the fact that Python code is very simple for most plots, you will see some extra complexity as you try to do more complicated things. Don't be put off by this! Matplotlib gives you a lot of control over all aspects of your plot and gives figures of research-quality, which cannot always be

said of other languages. Finally once you have your plots from Python, you may need to make further small changes. Other packages which are well worth looking at are GIMP [4] (≈open source Photoshop), InkScape [7] (≈ open source Publisher) and the ImageMagick API [5].

2 Python FAQ

Why Python? Python was designed to be simple and readable. Python is extremely flexible and is suitable to most tasks, from simple scripting to interactive web applications. It is completely open-source and portable between Windows, Mac and Linux and there is a huge, active development community. Finally, MIT has decided to use Python as the official language for undergraduate instruction [15] and they know what they are talking about.

In practise Python works best as a 'glue' language. If you want to download some files via FTP, unzip them, rename them, create some directories, move the files into a directory based on their name, any of these tasks could be done with Python. Of course if you wanted to then parse and filter the files, perform some complex signal processing algorithms and then create a compelling plot of the data, you could use Python for this also.

Why not Python? Python is a great all-round language, but like all languages, it has some drawbacks. First of all the official documentation is not good, it is best to learn from other people, teaching yourself or from blogs, Stack Overflow etc. Secondly, you must use care to write code that will deal with large data structures; some Python structures are inefficient (although there are modules which allow Python to access low level structures in C/FORTRAN)

How is Python used? Python can be run in interactive mode à la MATLAB or in batch mode. We will focus on writing a script which is executed with `python script.py` from the command line.

What else should you know about Python? Python is dynamically typed and interpreted. Python 3.x is still a work in progress, so almost everyone still uses 2.x.

2.1 Modules

Modules contain extra functions, in the same way as packages in R or libraries in C. There is an astonishing number of Python modules written for all kinds of applications. The most important ones we will use are matplotlib for plotting and NumPy for numerical calculations (these modules do not come in the standard distribution but can be easily installed using `apt-get`). Other core Python modules which are useful are csv (for parsing .csv files) and os/shutil (for performing operating system commands).

The following example demonstrates how to import and use the sys module

```
import sys # Preserves namespace
print sys.version # Prints information on the Python version
```

An alternative (not advised!) way to import modules is as follows

```
from sys import * # Now all functions from sys moved into global namespace
print version # Same as sys.version above
```

In this case we can access sys functions *without specifying the sys module*. However this may conflict with a function called version from a different module later on. More info here [12]

2.2 Lists

The most common Python structure is a list, basically a mutable array. Lists can contain a mixture of anything from integers and floats to strings and user defined classes. Lists are extremely useful when you don't know in advance how much space you will need, but they are not memory efficient or fast. For large amounts of data it is better to use NumPy arrays (covered later).

```
testList=[] # Make an empty list
testList.append(1) # Add an entry
```

```
testList.append(2) # Add another entry to the end
testList.append(3) # Keep adding as many as you like

print testList[0] # Returns the first entry of the list (1 in this case)
print testList.pop() # Removes the last entry of the list and returns it

testList2=[3.5,7,4.2] # Make a list initialised with 3 numbers

testList3=[i for i in range(10)] # Make a list with integers 0-9

testList4=[i*i for i in testList3] # New list with square of entries of testList3

print testList4 # Prints the contents of the list

print len(testList4) # Returns number of elements in list

testList5=['a',43,27.2] # This list contains a string a float and an integer

for entry in testList5: # Loop over entries of list
        print entry     # Loop contents must be indented

testList6=testList5+1
# We cannot do vectorised operations with lists
```

1

More info here [8]

2.3 NumPy Arrays

NumPy is an excellent numerical computing package for Python including array operations and linear algebra among others. The central data structure is a NumPy array, which is an alternative to a Python list. The main differences between the two are summarised below. However in general lists are useful for holding small amounts of data whereas NumPy arrays are better for large amounts of data especially multi-dimensional data and performing vectorised operations. (If you are familiar with R syntax)

NumPy Array	Python List
Fast	Slow
Homogeneous	Fixed Type
Fixed Size	Mutable
Vectorised	Element-wise

Numpy arrays can be created in 3 different ways;

1. Converting a list into a NumPy array
2. The return value of a built-in function
3. Reading from a file directly into a NumPy array

```
import numpy as np

exampleList=[1,2,3,4]
exampleArray=np.array(exampleList)
# Convert list to numpy array

exampleArray2=np.zeros(shape=(2,2),dtype=float)
# Creates a 2x2 array of zeros
# with type float
```

[1] Latex highlighting for Python adapted from here http://blog.miliauskas.lt/2008/09/python-syntax-highlighting-in-latex.html

```
exampleArray3=np.loadtxt('input_file.txt',delimiter='\t')
# Array size is calculated from number of
# rows and collumns in the file.
```

NumPy arrays can be sliced and indexed.

```
import numpy as np

testArray=np.arange(15)
# Creates an array with elements 0-14

testArray[3]
# Access the 4th element of array

np.shape(testArray)
# Returns shape of array (=(15,))

testArray2=np.linspace(0,0.9,10)
# Creates an array with elements 0,0.1,0.2...0.9

testArray3=testArray2.reshape(2,5)
# Reshape into a 2x5 array

subArray=testArray3[:,1]
# Returns all elements from 2nd column

subArray2=testArray3[1,:]
# Returns first row of array
```

2.4 List vs NumPy Array Speed Test

Lists and NumPy arrays are fundamentally different. As elements are added to lists Python allocates the required memory in chunks, whereas the memory is allocated in one go when NumPy arrays are initialised. As a result NumPy arrays cannot be extended once initialised. As this test shows, it is much quicker to create a NumPy array than a list.

```
import numpy as np
import time

testSize=100000

startTime=time.time()
timedList=[0 for i in range(testSize)]
print time.time()-startTime
# 0.0.8s

startTime=time.time()
timedArray=np.zeros(shape=testSize)
print time.time()-startTime
# 0.0003s
```

NumPy arrays are essentially Python wrappers for C style arrays, these low-level structures are much faster than pure Python structures yet the syntax is simpler than the corresponsing C code. This is one of the reasons Python is so popular; data intensive yet scalable algorithms can be developed relatively easily. In this test we compare the time to create a list and a NumPy array and add 1 to each element of both.

```
import numpy as np
import time

testSize=100000

startTime=time.time()
```

```
timedList=[0 for i in range(testSize)]

for i in range(len(timedList)):
        timedList[i]=timedList[i]+1
# Must operate on each element individually
# timedList=timedList+1 will not work

print time.time()-startTime
# 0.02s
startTime=time.time()

timedArray=np.zeros(shape=testSize)

timedArray=timedArray+1
# No need to loop here

print time.time()-startTime
# 0.0009s
```

Practically speaking lists can be used when the size is around 1000 entries. Any larger than this, it is probably worth converting to a NumPy array especially if you can replace lots of loops with vectorised operations.

It is also worth explaining Python loops. Python is unique as the *indentation and whitespace* matters. Whenever you define a function or class, or open any kind of loop using a colon;':', the body of the loop or definition must be indented. You can choose any number of spaces or a tab to indent, but you must use the same thing consistently. In practise it is best to use a tab to indent each line.

```
for i in range(10):
# Each line of the body of the loop begins with a tab
        print i

        if i>5:
        # The body of the if statement must begin with 2 tabs
                print 'Greater than 5'
```

2.5 Strings

Strings are created with single or double quotations and are mutable

```
testString='Hello'
print testString

print testString[0]
# Strings act like lists of characters
# Prints 'H'

testString=testString+' world'
# We can add to strings

testNumber=42

print testNumber
# Returns integer representation of number, 42

print str(testnumber)
# Returns string representation of number, '42'

answerString='The answer is '+str(testNumber)
print answerString
# 'The answer is 42'
```

2.6 Dictionaries

Dictionaries are effectively hash tables which store values of any arbitrary type which can be accessed using keys of arbitrary type.

```
directory={'Alice':4001,'Bob':1993}
# Creates a dictionary with two entries

directory['Alice']
# Returns the value for key 'Alice'

directory.keys()
# Returns a list of keys
# ['Alice','Bob']

directory.values()
# Returns a list of values for keys
#[4001,1993]

for k,v in directory.items():
        print k,v
# Prints out each key,value pair
# 'Alice',4001
# 'Bob',1993
```

2.7 Reading Files

We considered `np.loadtxt(...)` when discussing NumPy array. This is useful when we have a 'clean' input file such as `data_clean.txt`. Here there are 2 columns of numbers, there are no missing entries and no entries which are not numbers i.e. strings. This kind of file is likely to have been cleaned or pre-processed to arrive in this state. However there may be cases when we need to filter the lines of a text file, for example if there are text strings in the file between numbers which we need to ignore such as in `data.txt`. For cases like this, the csv module is preferable. It allows you to loop through each line one by one and, filter if neccesary and load each one into a list or NumPy array element by element. In other words *the cleaning and the parsing are done in one step*.

```
import numpy as np

array=np.loadtxt('data_clean.txt',delimiter="\t")
# Read from file straight into array
# We can use single or double quotes '' or ""

np.shape(array)
# Read in 2 collumns of length 100
```

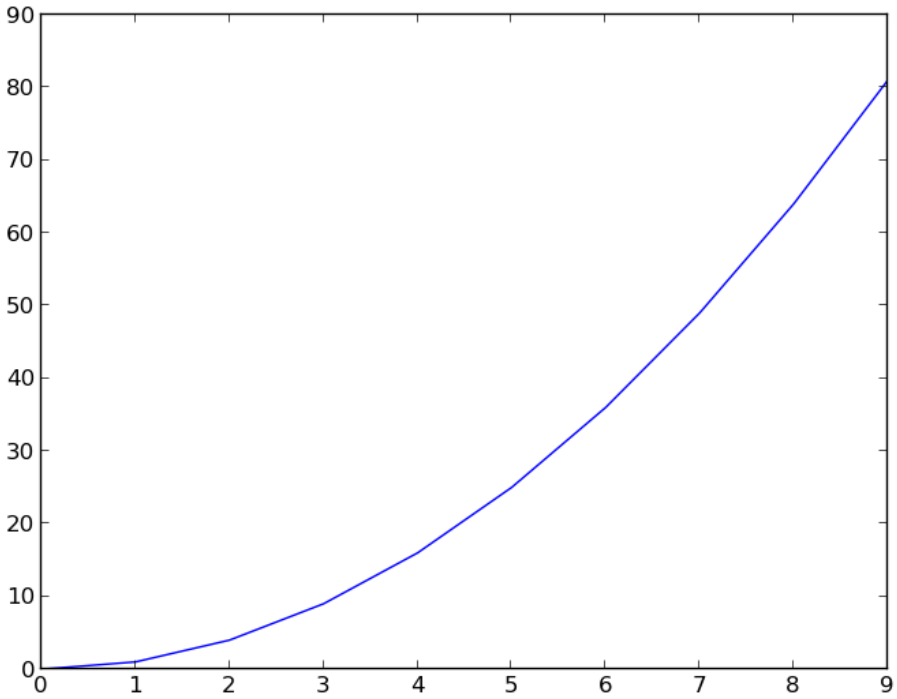
`data.txt` has not been cleaned. It contains some empty lines and some non-numeric entries. We will filter these out.

```
import csv

inFileName='data.txt' # Store file name in a string

inFile=csv.reader(open(inFileName,'r'),delimiter="\t")
# Open file for reading, fields split by tabs

values1=[]
values2=[]
# Empty lists to store first and second column

for line in inFile:

        if line[0].isdigit() and line[1].isdigit() and len(line)==2:
        # Check that both columns are numbers and no missing entries
```

```
        # before putting into arrays
                values1.append(float(line[0]))
                values2.append(float(line[1]))
                # The parts of the line are strings, convert to float
        else:
                print 'SKIPPING',line
# Puts entries into lists
```

3 Line Plots

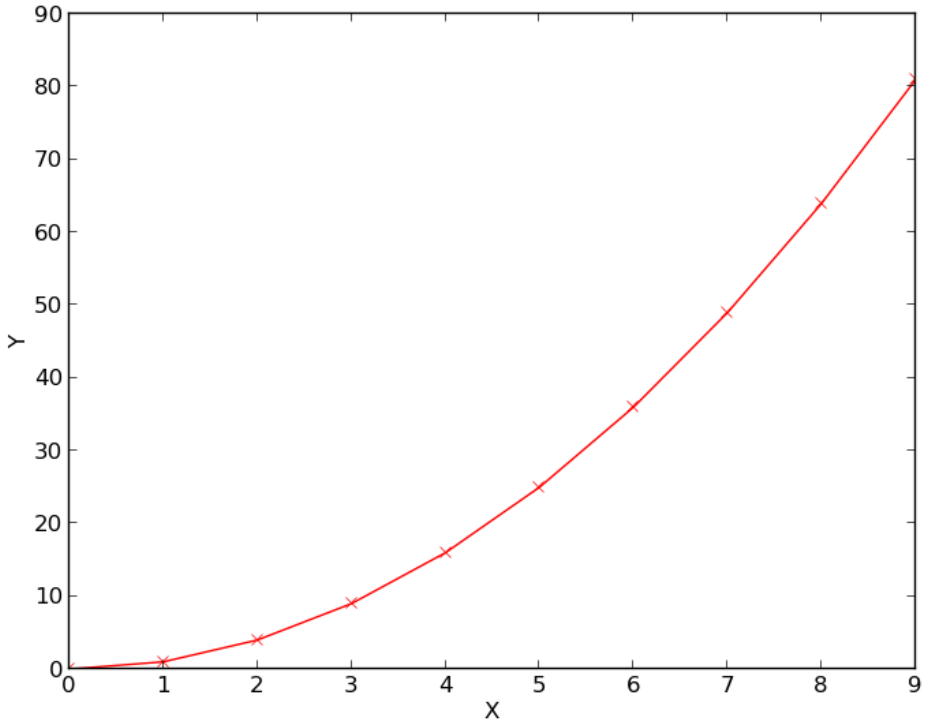
```
import matplotlib.pyplot as plt
# Import plotting functions, give it a shorter name

xValues=[i for i in range(10)]
yValues=[x*x for x in xValues]
# Make data, f(x)=x**2

plt.plot(xValues,yValues)
# Basic plot of list values
plt.savefig('plot.png')
# Save a copy of plt
plt.show()
# Show plot
```



This gives us the somewhat basic plot above. Lets add some labels, mark the points and change the colour to red. (You can see some possible plot markers here [11])

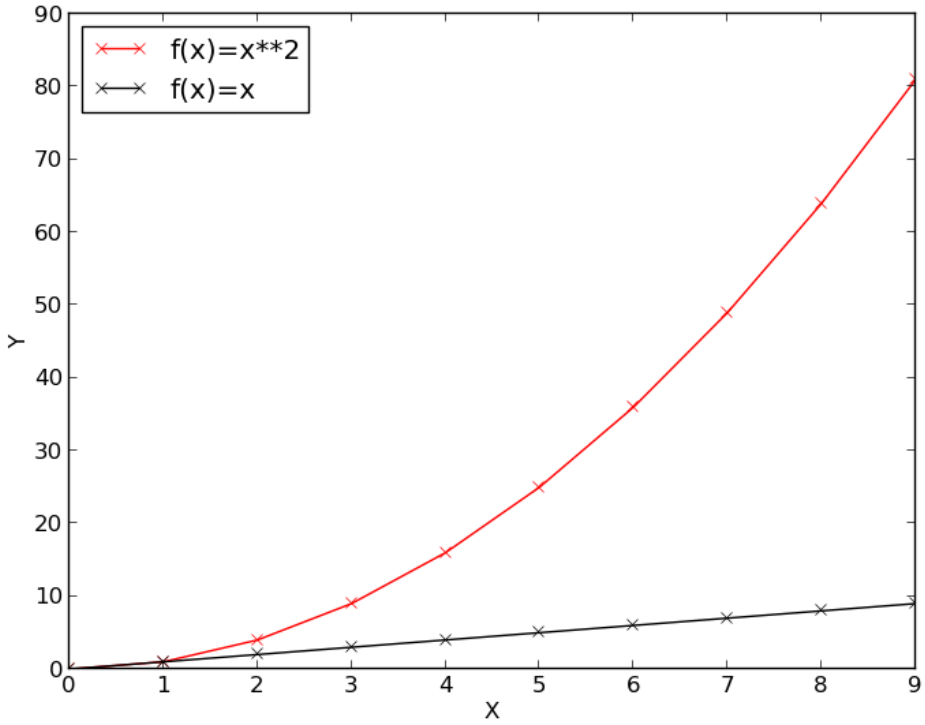
```
import matplotlib.pyplot as plt
```

```
xValues=[i for i in range(10)]
yValues=[x*x for x in xValues]

plt.xlabel('X')
plt.xlabel('Y')

plt.plot(xValues,yValues,marker='x',linestyle='-',color='red')

plt.savefig('plot.png')
plt.show()
```



Finally we might want to plot 2 different functions to compare them side by side. So aswell as plotting $f(x) = x^2$ let's plot $f(x) = x$ for comparison. We also need to add labels to each function so we can distinguish them.

```
import matplotlib.pyplot as plt

xValues=[i for i in range(10)]
yValues=[x*x for x in xValues]

yValues2=[x for x in xValues]

plt.xlabel('X')
plt.xlabel('Y')

plt.plot(xValues,yValues,marker='x',linestyle='-',color='red',label='f(x)=x**2')
plt.plot(xValues,yValues,marker='x',linestyle='-',color='red',label='f(x)=x')

plt.legend(loc='upper left')

plt.savefig('plot.png')
plt.show()
```

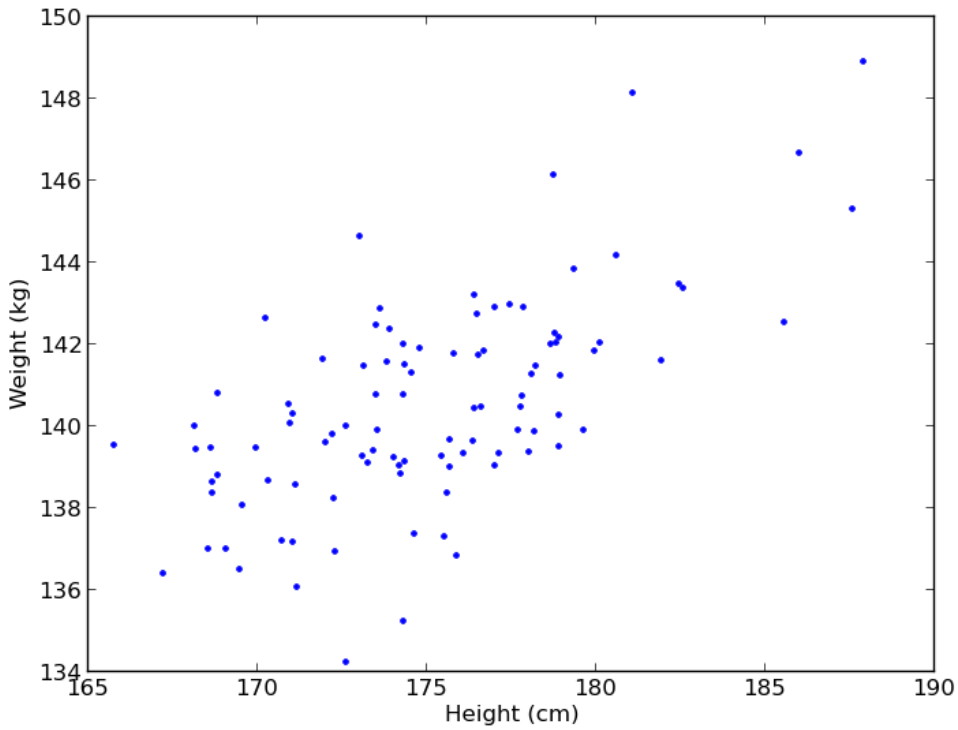

f(x)=x**2
f(x)=x
Y
X
0
10
20
30
40
50
60
70
80
90
0
1
2
3
4
5
6
7
8
9

4 Scatter Plots

Scatter plots show relationships between 2 different variables, as opposed to line plots which tend to show how a variable changes as a function of an independent variable such as time. This is a simple way to examine how one variable is related to another and also how each variable is distributed.

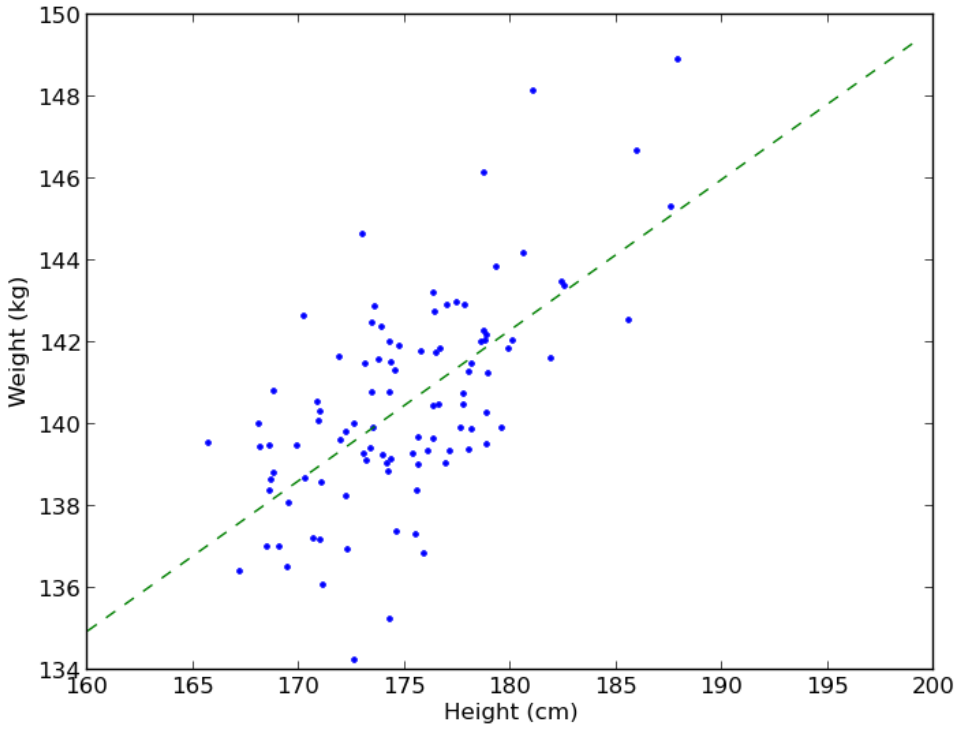
A classic example of such a *bivariate* distribution is the heights and weights of a group of people. `heights_weights.txt` contains heights and weights of 100 people. In the line plot example we used lists, in this example we will use NumPy arrays. We start by making a basic scatter plot.

```
import matplotlib.pyplot as plt
import numpy as np

data=np.loadtxt('heights_weights.txt',delimiter='\t')

heights=data[:,0]
weights=data[:,1]
# Heights is first column
# Weights is second column

plt.plot(heights,weights,'.')
# Each point is a small dot
plt.xlabel('Height (cm)')
plt.ylabel('Weight (kg)')

plt.show()
```




```
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats
# We need SciPy stats library for linear regression
```

```
data=np.loadtxt('heights_weights.txt',delimiter='\t')

heights=data[:,0]
weights=data[:,1]

plt.plot(heights,weights,'.')

plt.xlabel('Height (cm)')
plt.ylabel('Weight (kg)')

slope,intercept,rValue,pValue,stdErr=scipy.stats.linregress(heights,weights)
# Fit data, linregress(...) returns 5 values

xFit=np.linspace(160,200)
# Make array of values spanning range in x
yFit=intercept+(xFit*slope)
# Make array of y values
plt.plot(xFit,yFit,linestyle='--')
# Superimpose line plot

print 'SLOPE',slope
print 'INTERCEPT',intercept
print 'r',rValue
# Inspect values

plt.savefig('scatter_plot.png')
plt.show()
```

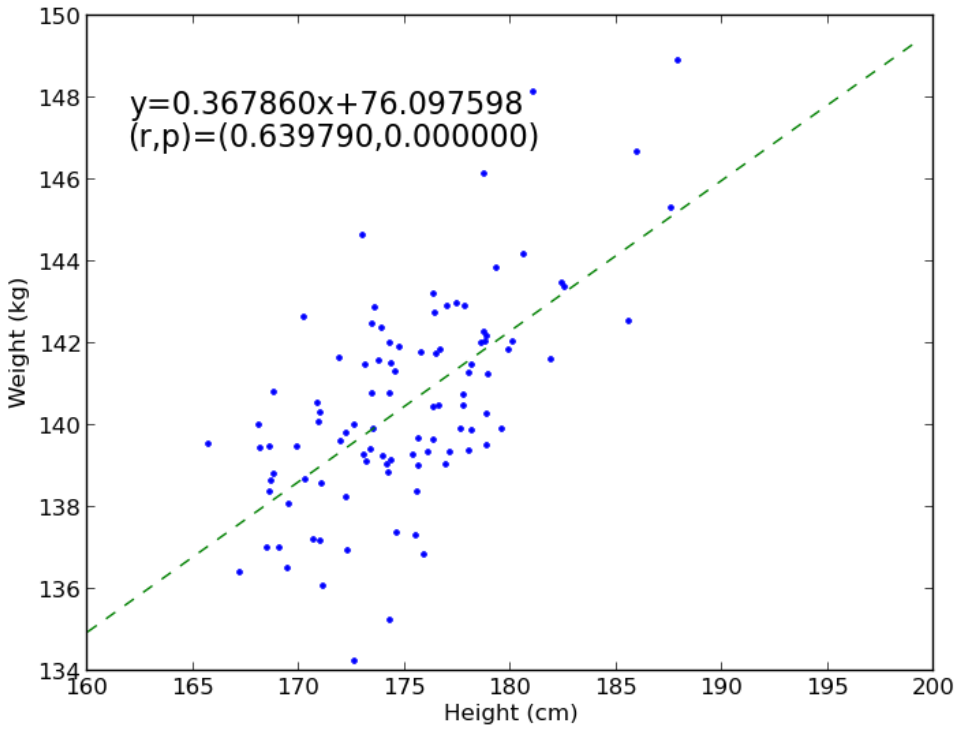


We find an r value of 0.84 which is a pretty strong correlation and a p value of $< 10^{-12}$ which shows it is virtually impossible that the true regression line has zero gradient. Finally we will add a nice caption to the plot using a formatted label.

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats

inFile=csv.reader(open('heights_weights.txt','r'),delimiter='\t')

data=np.loadtxt('heights_weights.txt',delimiter='\t')

heights=data[:,0]
weights=data[:,1]

plt.plot(heights,weights,'.')

plt.xlabel('Height (cm)')
plt.ylabel('Weight (kg)')

slope,intercept,rValue,pValue,stdErr=scipy.stats.linregress(heights,weights)

xFit=np.linspace(160,200)
yFit=intercept+(xFit*slope)

plt.plot(xFit,yFit,linestyle='--')

print 'SLOPE',slope
print 'INTERCEPT',intercept
print 'r',rValue
print 'P',pValue

plt.annotate('y=%fx+%f' % (slope,intercept),xy=(0.05,0.85),xycoords='axes fraction',size=16)
plt.annotate('(r,p)=(%f,%f)' % (rValue,pValue),xy=(0.05,0.8),xycoords='axes fraction',size=16)
plt.savefig('scatter_plot.png')
plt.show()
```

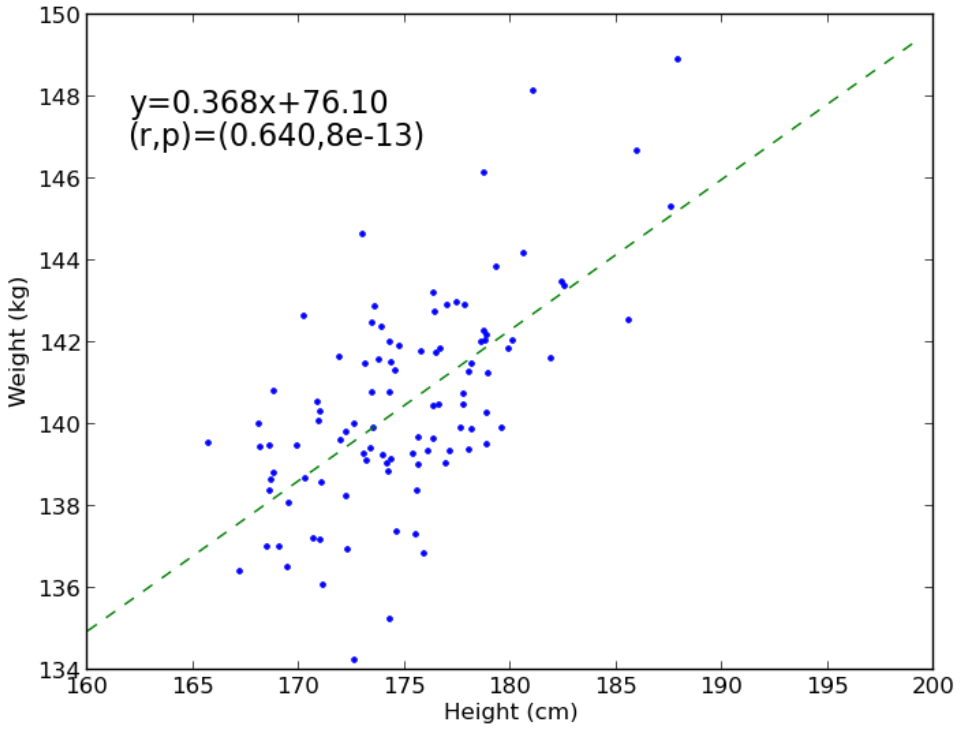
Now we have a caption. First of all we specified a formated string `y=%fx\+%f % (slope, intercept)`, the variables in brackets `slope` and `intercept` replace the first and second `%f`; the `f` tells Python the variable is a `float`, we would use `%d` for an integer. Then we specified its position with `xy`. We can specify the position in a number of ways, in this case it is as a fraction of the axes bounds. So (0,0) is the bottom left of the graph and (1,1) the top right corner. However the caption doesn't look great, there are too many digits for the slope and too few for the p-value. So let's try again, this time specifying the precision (you can read about formating strings here [3]). The syntax might be familiar to you from C.

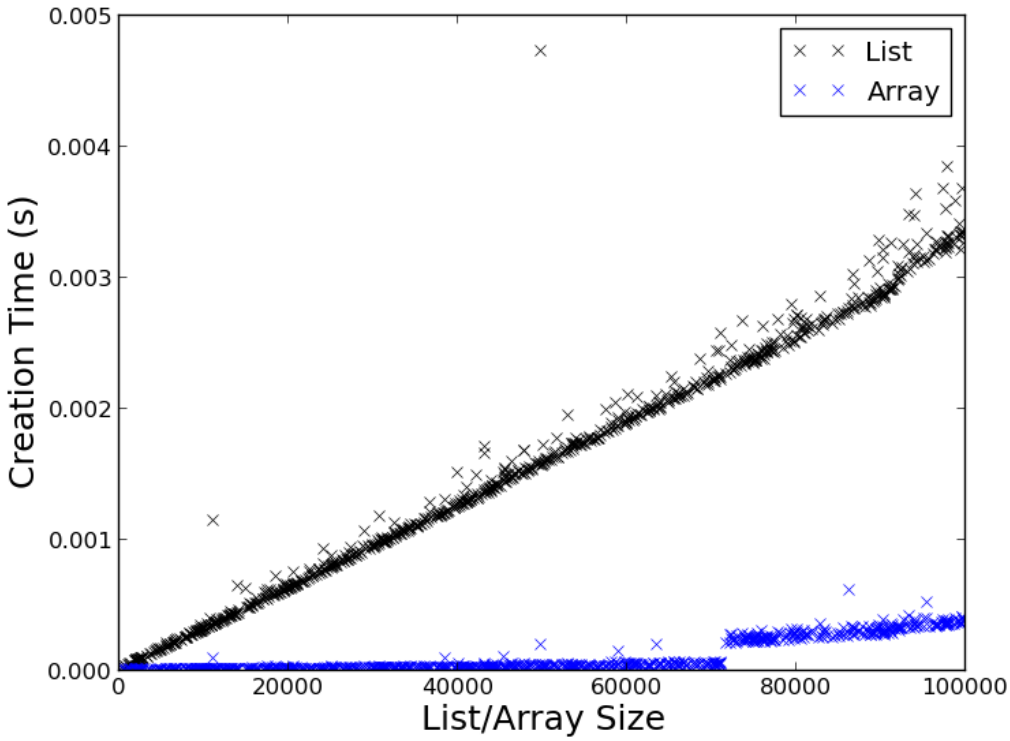
```
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats

data=np.loadtxt('heights_weights.txt',delimiter='\t')

heights=data[:,0]
weights=data[:,1]

plt.plot(heights,weights,'.')

plt.xlabel('Height (cm)')
plt.ylabel('Weight (kg)')

slope,intercept,rValue,pValue,stdErr=scipy.stats.linregress(heights,weights)

xFit=np.linspace(160,200)
yFit=intercept+(xFit*slope)

plt.plot(xFit,yFit,linestyle='--')
```




```
print 'SLOPE',slope
print 'INTERCEPT',intercept
print 'r',rValue
print 'P',pValue

plt.annotate('y=%.3fx+%2.2f'%(slope,intercept),xy=(0.05,0.85),xycoords='axes fraction',size=16)
# .3f prints a float with no numbers before decimal and 3 after. 2.2 gives 2 before and 2 after
plt.annotate('(r,p)=(%.3f,%.0e)'%(rValue,pValue),xy=(0.05,0.8),xycoords='axes fraction',size=16)
# .0e prints a number in exponential format
plt.savefig('scatter_plot.png')
plt.show()
```

Starting with the data in file `TIMES_SIZES.txt` reproduce the plot shown below. This is a systematic investigation of how long it takes to create both a list and an array of a given size.

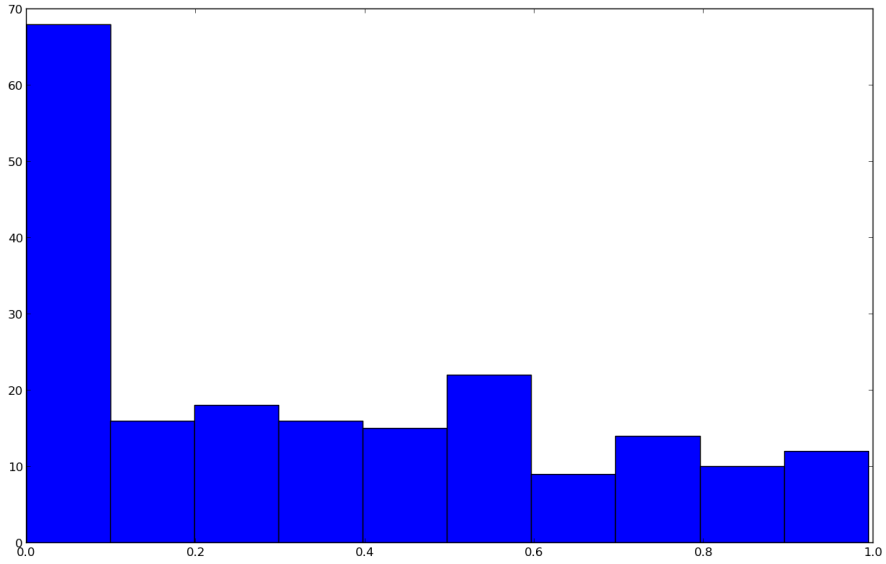

y=0.368x+76.10
(r,p)=(0.640,8e-13)
Weight (kg)
Height (cm)
150
148
146
144
142
140
138
136
134
160
165
170
175
180
185
190
195
200


List
Array
Creation Time (s)
List/Array Size
0.005
0.004
0.003
0.002
0.001
0.000
0
20000
40000
60000
80000
100000

5 Histograms

Histograms are a common method to display *distributions* of data over an interval. The interval is divided into *bins* to visualise the distribution.

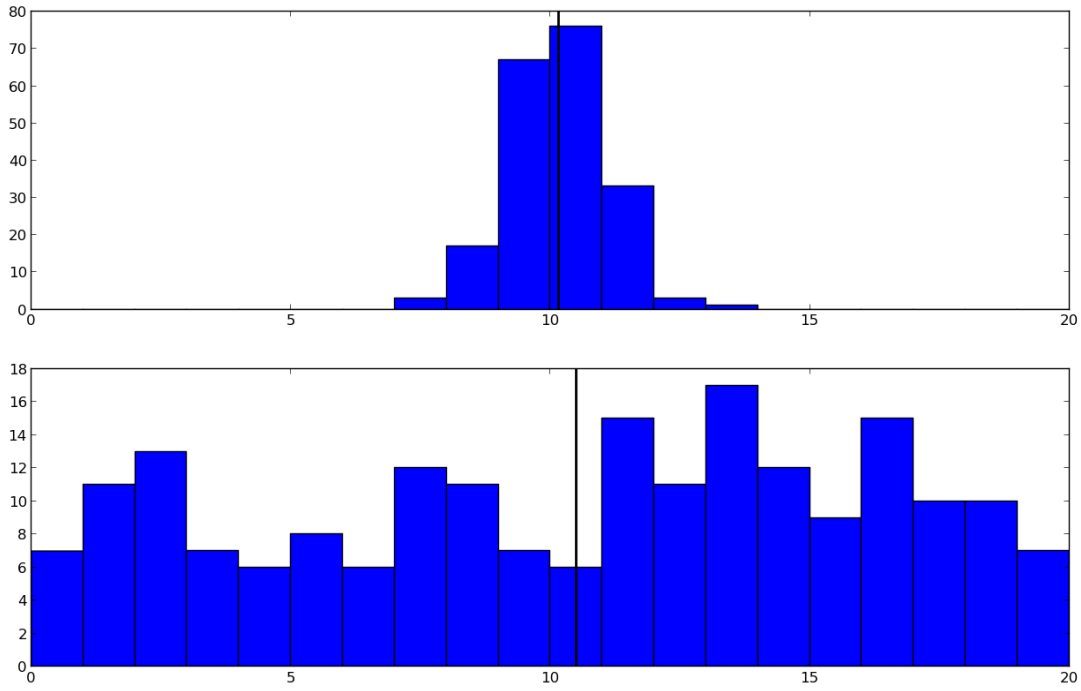
```
import numpy as np
import matplotlib.pyplot as plt

nSamples=200
samples=np.random.power(0.5,size=(nSamples))
# np.random.power samples a power law
# distribution with exponent 0.5

plt.hist(samples)
# By default plots with 10 bins
plt.show()
```



In general it is useful to visualise how data is distributed as a simple mean may obscure some details as the following examples demonstrate.

```
import numpy as np
import matplotlib.pyplot as plt

nSamples=200

normalSamples=np.random.normal(10,1,nSamples)
# Sample normal dist with mean 10
uniformSamples=np.random.uniform(0,20,nSamples)
# Sample uniformly in range 0-20

fig=plt.figure()
ax=fig.add_subplot(211)
# Divide the plot area into 2; first plot

ax.hist(normalSamples,bins=20,range=(0,20))
# Specify number of bins and range
ax.axvline(np.mean(normalSamples),linewidth=2,color='k')
```

```
# Plot histogram and mark the mean
plt.xlim(0,20)
# Plot full interval

ax2=fig.add_subplot(212)
# Second plot

ax2.hist(uniformSamples,bins=20,range=(0,20))
# Specify SAME range to compare plots
ax2.axvline(np.mean(uniformSamples),linewidth=2,color='k')
# Plot histogram and mark the mean

plt.xlim(0,20)
# Plot full interval

plt.show()
```

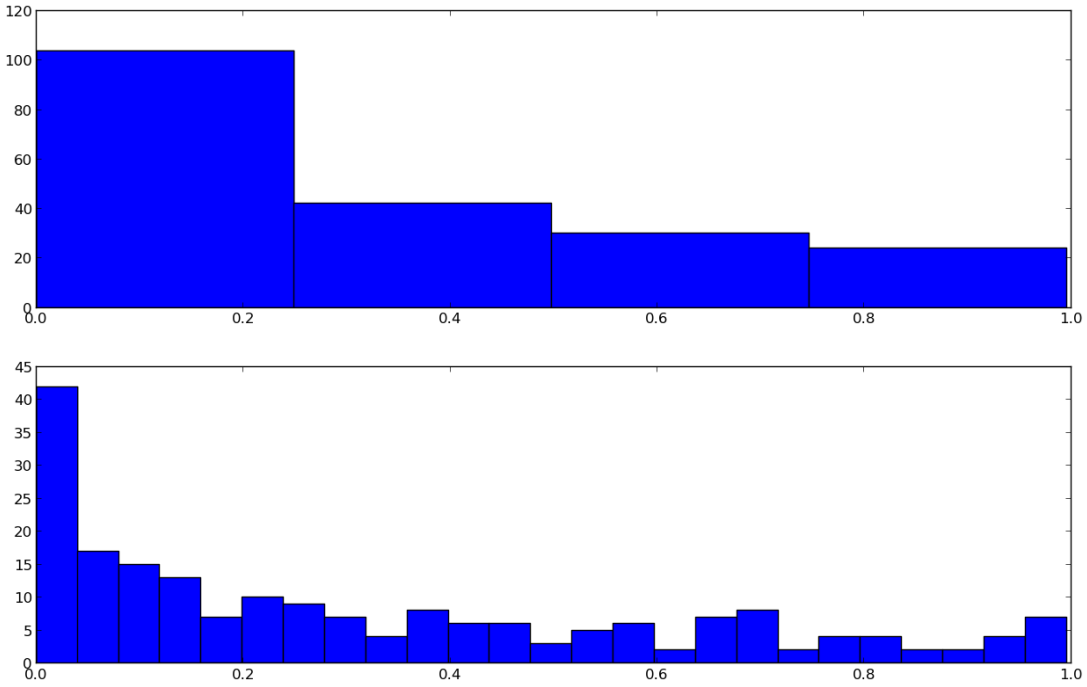


Here we compare a set of samples from two very different distributions (a normal and a uniform distribution). If we simply took the mean of these samples, we would conclude that they are comparable; the black lines are very close. But clearly their behaviours are very different. The probability of a value in the range [15-16] is much smaller in the top, normal distribution for example.

Further, the choice of the number of bins is more of an art than a science. If there are too many bins then the occupation of each bin will be small and dominated by noise and some bins may even be empty. On the other hand, if the interval is divided into too few bins then it will be difficult to observe a trend.

```
import numpy as np
import matplotlib.pyplot as plt

nSamples=200

samples=np.random.power(0.5,size=(nSamples))

fig=plt.figure()
ax=fig.add_subplot(211)
# Make array of plots, 2 rows
```

```
# 1 column, plot 1st element of array

ax.hist(samples,bins=4)
# Plot with few bins

ax2=fig.add_subplot(212)
# Now plot 2nd element of
# plot array

ax2.hist(samples,bins=25)
# Plot with many bins

plt.show()
```

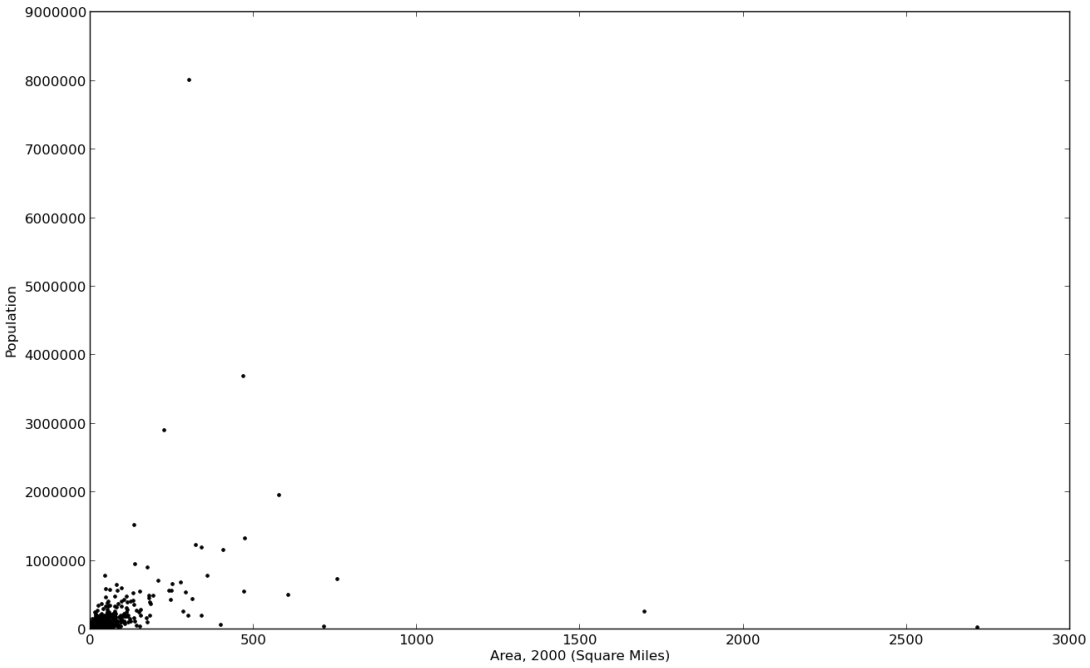


6 Log-Log Plots

When plotting data with very large ranges, it can be hard to meaningfully compare different regions of the plot. Especially if there are very large outliers as is typical with heavy-tailed distributions.

Consider the data in `AREA_POP_cleaned.txt`. This file contains the area and population of a number of American cities (limited to those with population 25,000 or more, taken from [14]).

```
import numpy as np
import matplotlib.pyplot as plt

data=np.loadtxt('AREA_POP_cleaned.txt',delimiter=' ')

plt.plot(data[:,0],data[:,1],'k.')
# Plot first column against second

plt.xlabel('Area, 2000 (Square Miles)')
plt.ylabel('Population')

plt.show()
```



This plot is very uninformative, the majority of the points are all bunched in the lower left corner and it is not clear if there is any trend between the two variables. Instead we change the scale of each axis from linear to logarithmic.

```
import numpy as np
import matplotlib.pyplot as plt

data=np.loadtxt('AREA_POP_cleaned.txt',delimiter=' ')

plt.loglog(data[:,0],data[:,1],'k.')
# Now try log-log scale

plt.xlabel('Area, 2000 (Square Miles)')
plt.ylabel('Population')

plt.show()
```

7 Final Thoughts

The matplotlib API is worth investing time in learning to use, once you have a good grasp of it's features you will see that it is capable of doing pretty much anything you want. If you can combine it with the ability to do final point and click editing using a standalone program then, with a little bit of creativity, you can start to make some excellent data-art. As an example, the figure below was created using only matplotlib (the code to create it is listed in the appendix)

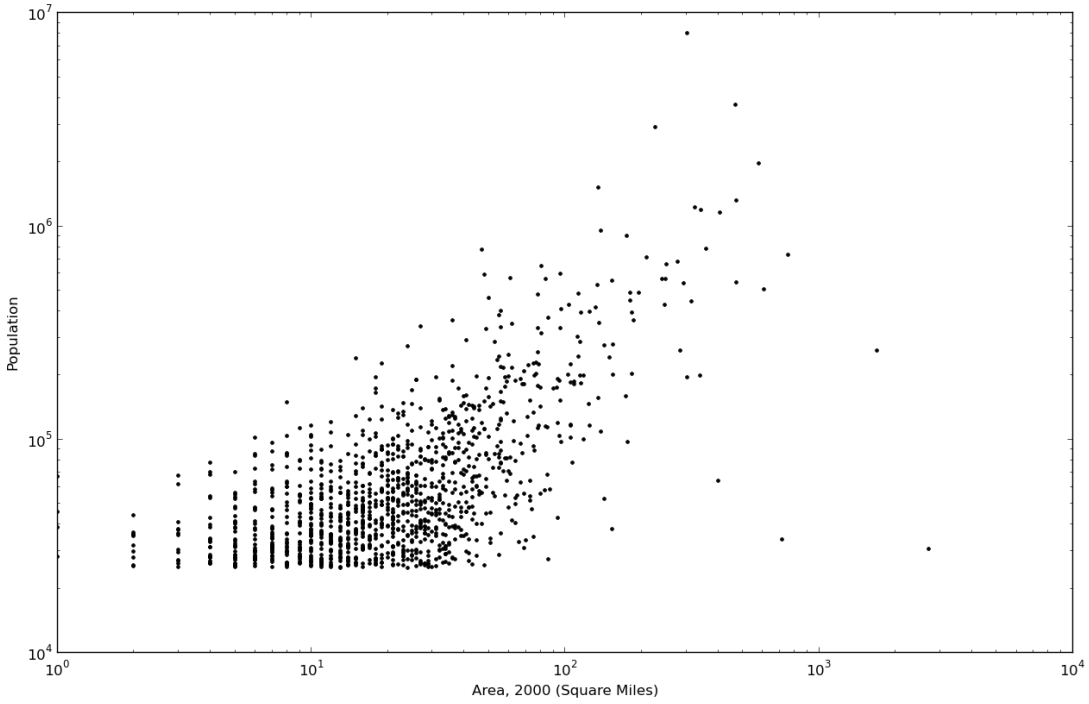
Other places to get inspiration are FlowingData [2] and Information is Beautiful [6]. Also you can browse the gallery of different plots on the matplotlib website, along with the source code [10].

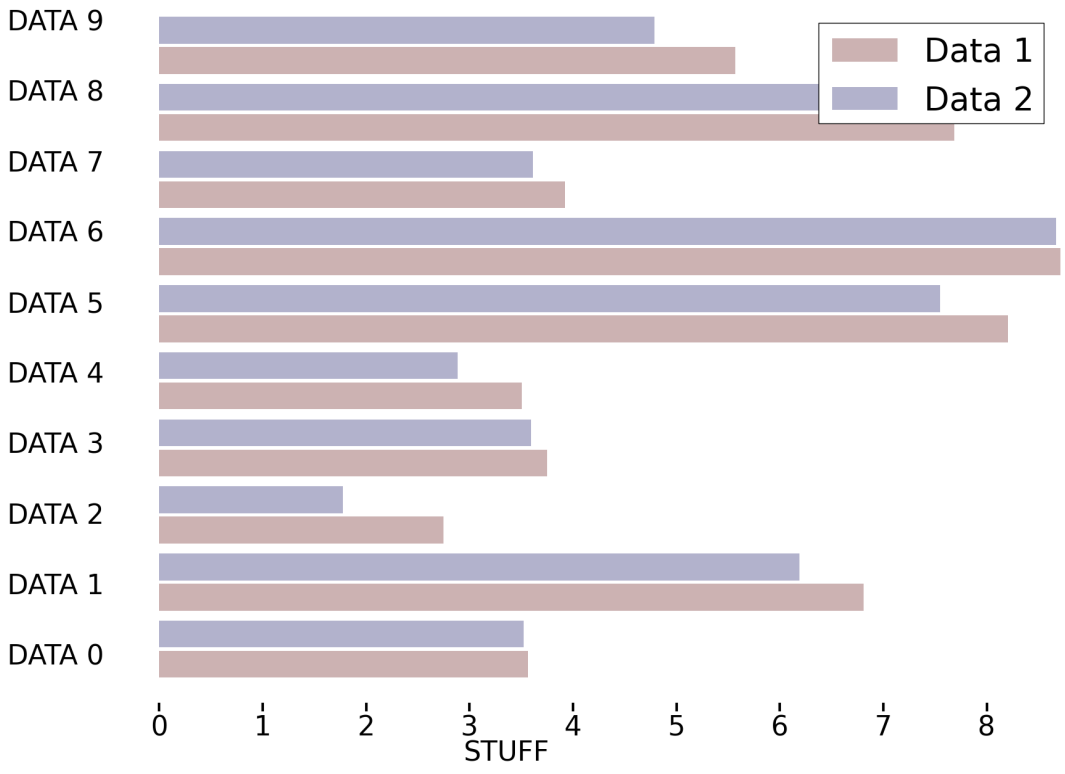
References

[1] Choosing a Good Graph. http://extremepresentation.typepad.com/blog/2006/09/choosing_a_good.html.

[2] Flowing Data. http://www.flowingdata.com/.

[3] Formatting Python Strings. http://www.diveintopython.net/native_data_types/formatting_strings.html.





[4] GNU Image Manipulation Program. http://www.gimp.org/.

[5] Imagemagick. http://www.imagemagick.org/Usage/.

[6] Information is Beautiful. http://www.informationisbeautiful.net/.

[7] Inkscape. http://inkscape.org/.

[8] Introduction to Python Lists. http://www.pythonforbeginners.com/python-lists-cheat-sheet/.

[9] Matplotlib. http://www.matplotlib.org/.

[10] Matplotlib Examples. http://matplotlib.org/gallery.html.

[11] Matplotlib Line Plot Styles. http://matplotlib.org/examples/pylab_examples/line_styles.html.

[12] Python Modules. http://www.pythonforbeginners.com/python-modules/.

[13] Taxonomy of a Data Scientist. http://www.dataists.com/2010/09/a-taxonomy-of-data-science/.

[14] US City Data. http://www.census.gov/statab/ccdb/ccdbcityplace.html.

[15] Why did MIT Switch to Python. http://news.ycombinator.com/item?id=602307.

A Further Reading

1. Python Cheatsheet, http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html
2. NumPy for MATLAB Users, http://www.scipy.org/NumPy_for_Matlab_Users
3. NumPy for R Users, http://mathesaurus.sourceforge.net/r-numpy.html

B Pretty Bar Chart

```
import matplotlib.pyplot as plt
import matplotlib.lines as mpllines
from matplotlib.ticker import FixedLocator
import numpy as np

plt.rc('font',size=32)
plt.rc('xtick.major',size=10)
# Change global tick size and font size

size=10
# Number of collumns

xVals=np.arange(size)

yVals1=np.random.uniform(1,10,size=size)
yVals2=yVals1+np.random.uniform(-1,1,size=size)
# Create some random data

dataMax=max(max(yVals1),max(yVals2))

plt.barh(xVals,yVals1,height=0.4,color=(0.8,0.7,0.7),linewidth=0,label='Data 1')
plt.barh(xVals+0.45,yVals2,height=0.4,color=(0.7,0.7,0.8),linewidth=0,label='Data 2')
# Plot bars with thickness 0.4
# Shift second histogram to side
# by 0.45

for i in range(size):
        plt.annotate(r'DATA '+str(i),xy=(0.05,(float(i)/size)+(2.0/(3*size))),xycoords='axes fraction')
# Annotate each column

plt.xlim(-2,dataMax)
plt.ylim(-0.5,size)
# Change axis range, space out a bit
```

```
frame=plt.gca()
frame.axes.set_frame_on(False)
frame.axes.get_yaxis().set_visible(False)
frame.axes.get_xaxis().set_visible(True)
# Hide y axis

plt.xlabel(r'STUFF')

lines=frame.axes.get_xticklines()

for line in lines:
        line.set_marker(mpllines.TICKUP)
        line.set_markeredgewidth(3)
# Set tick width

frame.axes.xaxis.tick_bottom()
# Turn off ticks on upper x-axis

frame.axes.xaxis.set_major_locator(FixedLocator(range(0,int(dataMax)+1)))
# Fix tick locations

plt.legend()

plt.show()
```