python_data_science_intermediate

April 5, 2019

1 Python Basics Intermediate

1.1 Data visualization with matplotlib

Starting with line plots

1.1.1 Line Plot

```
In [1]: # First got to get the data
        import pandas as pd
        import io
        import requests
        bricsURL = "https://assets.datacamp.com/production/repositories/287/datasets/b60fb5bdb
        carsURL = "https://assets.datacamp.com/production/repositories/287/datasets/79b3c22c47
        gapminderURL = "https://assets.datacamp.com/production/repositories/287/datasets/5b1e4
        br=requests.get(bricsURL).content
        cr=requests.get(carsURL).content
        gr=requests.get(gapminderURL).content
        brics=pd.read_csv(io.StringIO(br.decode('utf-8')))
        cars=pd.read_csv(io.StringIO(cr.decode('utf-8')))
        gapminder=pd.read_csv(io.StringIO(gr.decode('utf-8')))

        # Or use the read_csv command below
        # cars = pd.read_csv('cars.csv')

        print(brics)

        l = list(brics['country'])
        print(type(l))
        print(l)
```

```
  Unnamed: 0       country    capital    area  population
0          BR        Brazil   Brasilia   8.516      200.40
1          RU        Russia     Moscow  17.100      143.50
2          IN         India  New Delhi   3.286     1252.00
3          CH         China    Beijing   9.597     1357.00
4          SA  South Africa   Pretoria   1.221       52.98
```

```
<class 'list'>
['Brazil', 'Russia', 'India', 'China', 'South Africa']
```

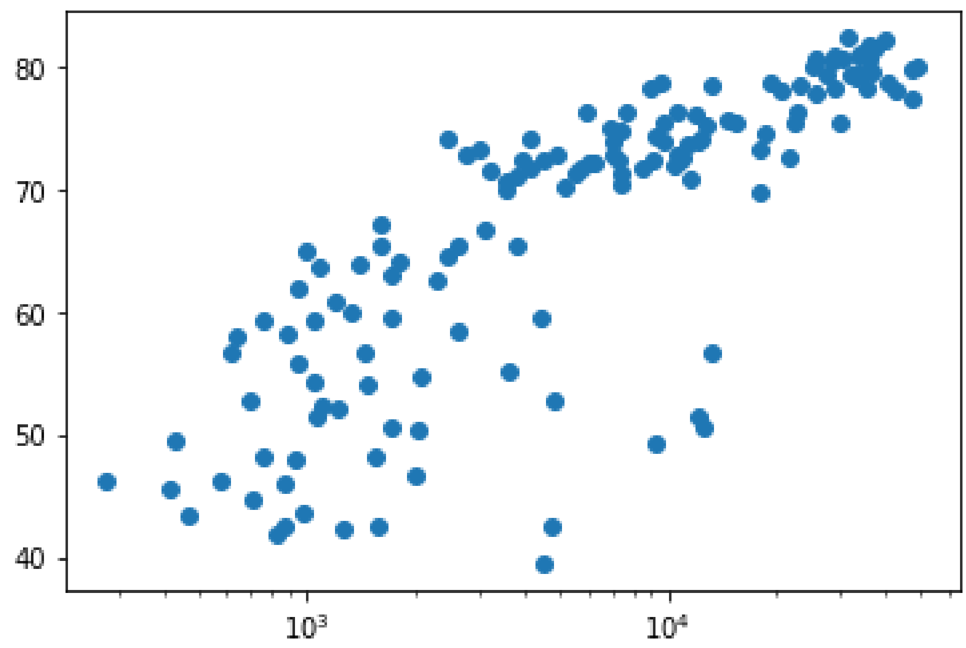
```
In [2]: # First import pyplot subpackage of matplotlib
        import matplotlib.pyplot as plt

        # Create some example data to plot
        year = list(range(1950,2101))
        pop = list(range(0,151))

        # Make a line plot: year on the x-axis, pop on the y-axis
        plt.plot(year,pop)

        # Display the plot with plt.show()
        plt.show()
```

```
<Figure size 640x480 with 1 Axes>
```

1.1.2 Scatterplot

When you have a time scale along the horizontal axis, the line plot is your friend. But in many other cases, when you're trying to assess if there's a correlation between two variables, for example, the scatter plot is the better choice.

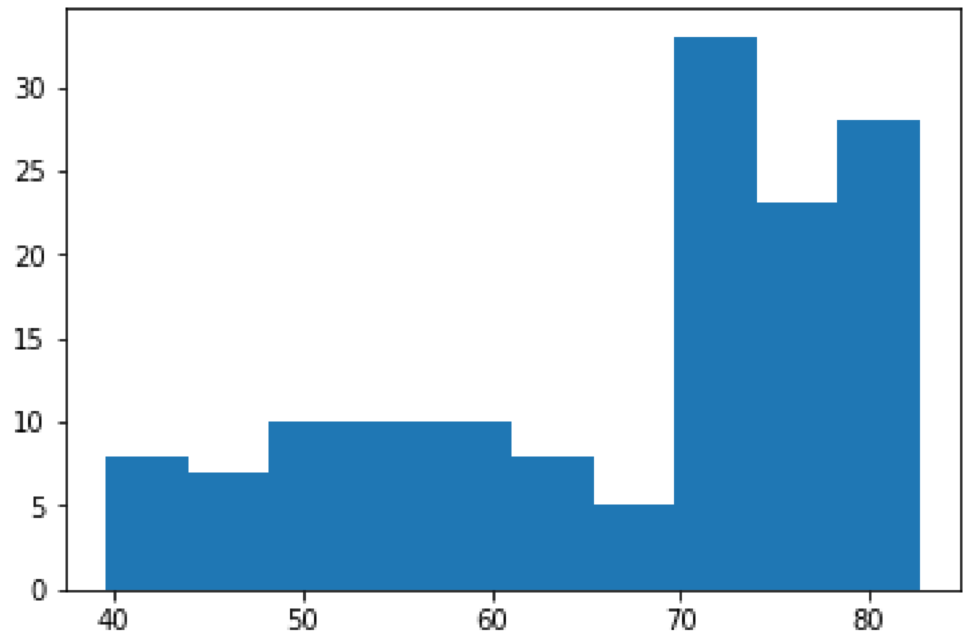
```
In [3]: # Set Variables
        gdp_cap = list(gapminder['gdp_cap'])
        life_exp = list(gapminder['life_exp'])

        # Call scatterplot
        plt.scatter(gdp_cap, life_exp)

        # Put the x-axis on a logarithmic scale
        plt.xscale('log')

        # Show plot
        plt.show()
```



1.1.3 Histograms

Now we're going to do histograms. These help to get the idea of the distribution of the data

```
In [4]: # Create histogram of life_exp data
        plt.hist(life_exp)

        # Display histogram
        plt.show()
```

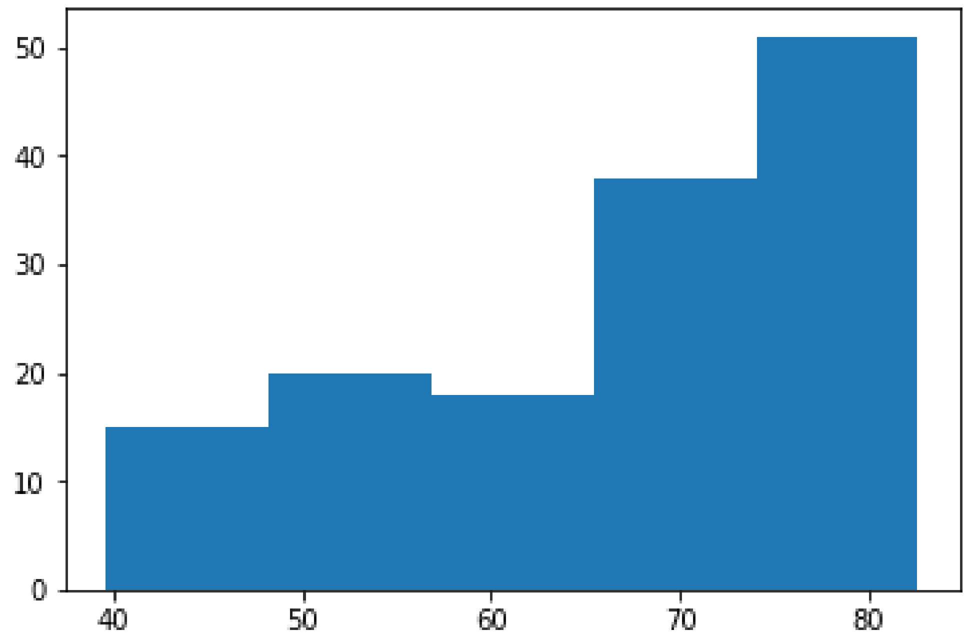


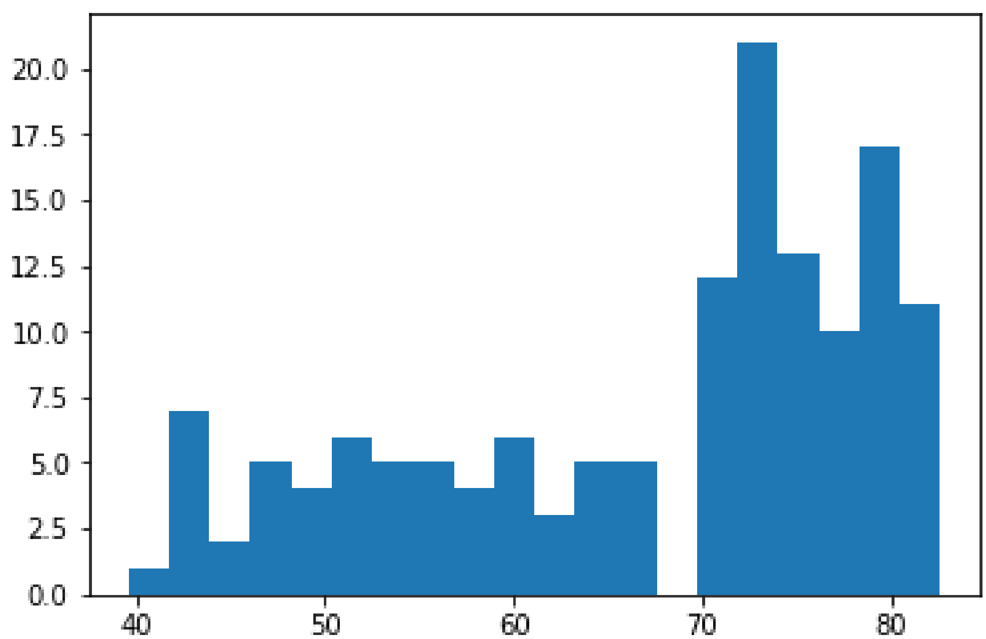
The number of bins is pretty important. Too few bins will oversimplify reality and won't show you the details. Too many bins will overcomplicate reality and won't show the bigger picture.

To control the number of bins to divide your data in, you can set the `bins` argument.

In the code below, `plt.clf` cleans up the display so you can start fresh

In [5]:
```
# Build histogram with 5 bins
plt.hist(life_exp, bins = 5)

# Show and clean up plot
plt.show()
plt.clf()

# Build histogram with 20 bins
plt.hist(life_exp, bins = 20)

# Show and clean up again
plt.show()
plt.clf()
```





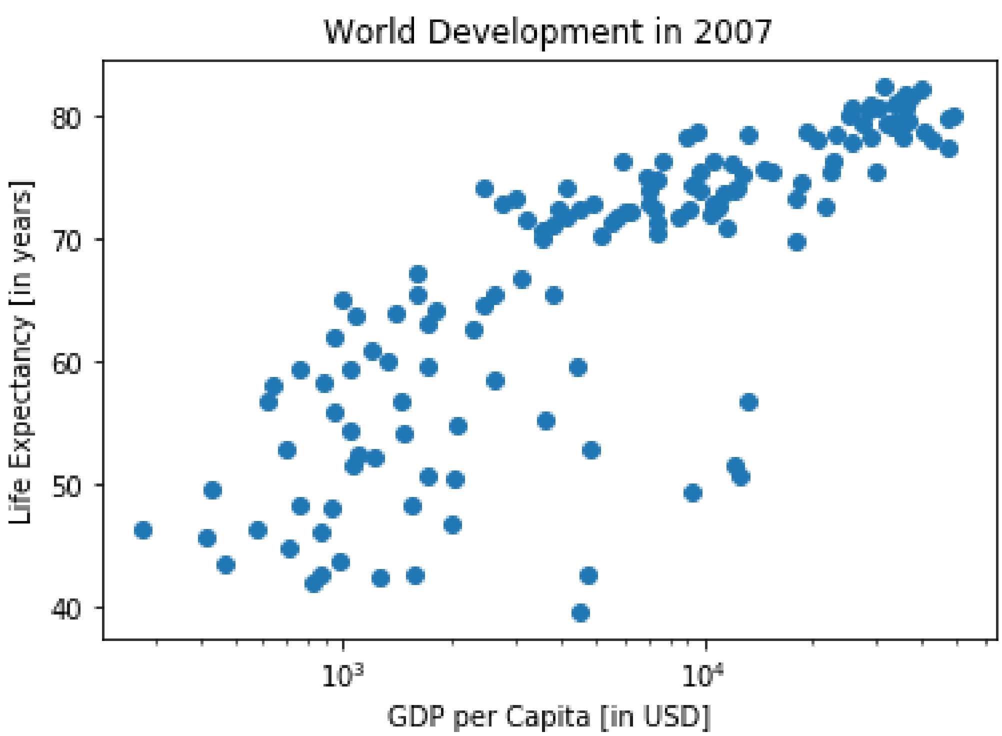

```
<Figure size 432x288 with 0 Axes>
```

1.1.4 Customization

How to you customize depends on the data itself and the stoy you want to tell
Some common options include:

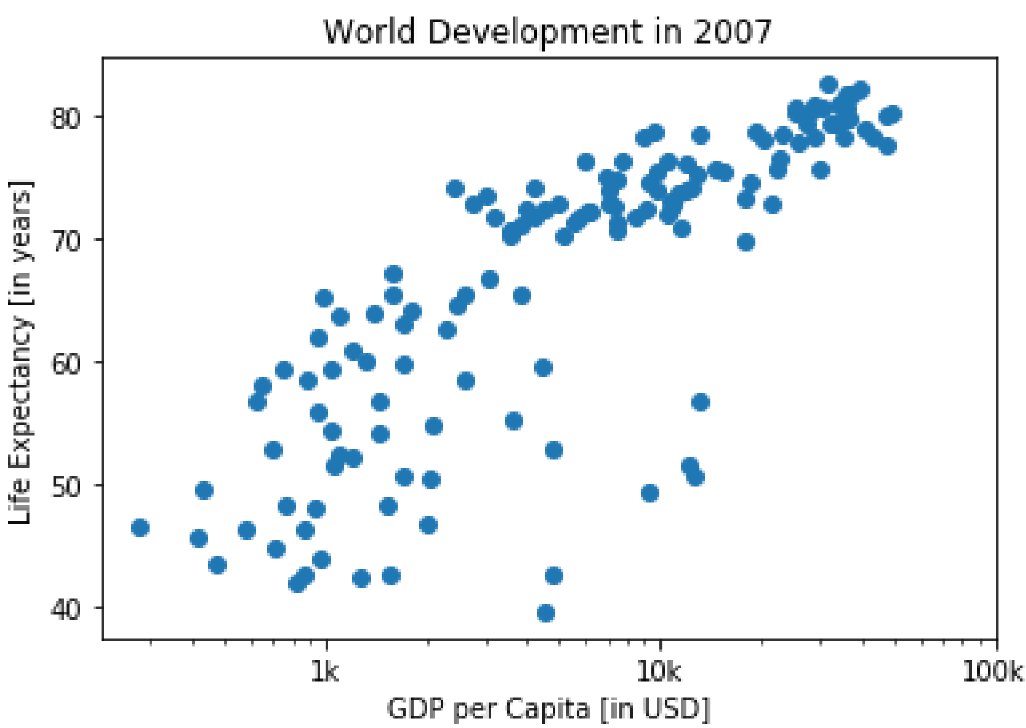
- plt.xlabel()
- plt.ylabel()
- plt.title()
- plt.yticks()

In [6]:
```
# Basic scatter plot, log scale
plt.scatter(gdp_cap, life_exp)
plt.xscale('log')

# Strings
xlab = 'GDP per Capita [in USD]'
ylab = 'Life Expectancy [in years]'
title = 'World Development in 2007'

# Add axis labels
plt.xlabel(xlab)
plt.ylabel(ylab)

# Add title
plt.title(title)

# After customizing, display the plot
plt.show()
```



The tick values 1000, 10000 and 100000 should be replaced by 1k, 10k and 100k. `plt.yticks()` and `plt.xticks()` have inputs of which tick marks to show and what they can be replaced as

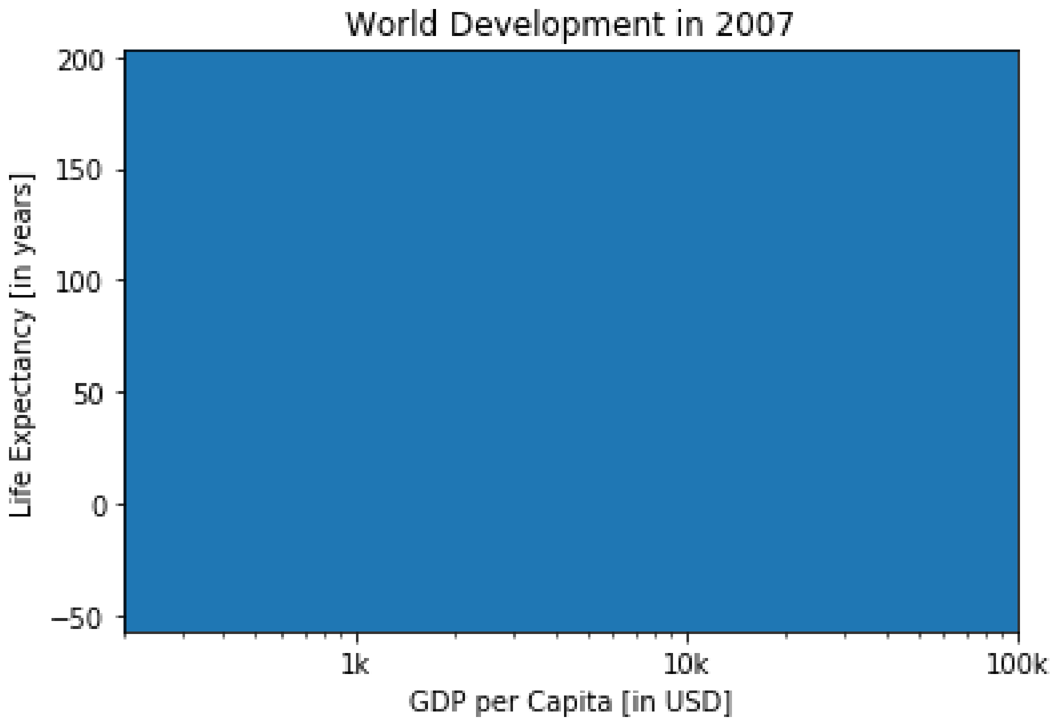
```
In [7]: # Scatter plot
        plt.scatter(gdp_cap, life_exp)

        # Previous customizations
        plt.xscale('log')
        plt.xlabel('GDP per Capita [in USD]')
        plt.ylabel('Life Expectancy [in years]')
        plt.title('World Development in 2007')

        # Definition of tick_val and tick_lab
        tick_val = [1000, 10000, 100000]
        tick_lab = ['1k', '10k', '100k']

        # Adapt the ticks on the x-axis
        plt.xticks(tick_val, tick_lab)

        # After customizing, display the plot
        plt.show()
```



Bubble Sizes in scatterplots Right now, the scatter plot is just a cloud of blue dots, indistinguishable from each other. Let's change this. Wouldn't it be nice if the size of the dots corresponds to the population?

To accomplish this, there is a list pop loaded in your workspace. It contains population numbers for each country expressed in millions. You can see that this list is added to the scatter method, as the argument s, for size.

In [8]:
```
# Import data
pop = list(gapminder['population'])

# Import numpy as np
import numpy as np

# Store pop as a numpy array: np_pop
np_pop = np.array(pop)

# Double np_pop
np_pop = np_pop * 2

# Update: set s argument to np_pop
plt.scatter(gdp_cap, life_exp, s = np_pop)

# Previous customizations
plt.xscale('log')
plt.xlabel('GDP per Capita [in USD]')
plt.ylabel('Life Expectancy [in years]')
plt.title('World Development in 2007')
plt.xticks([1000, 10000, 100000],['1k', '10k', '100k'])

# Display the plot
plt.show()
```



Coloring The next step is making the plot more colorful! To do this, a list `col` has been created for you. It's a list with a color for each corresponding country, depending on the continent the country is part of.

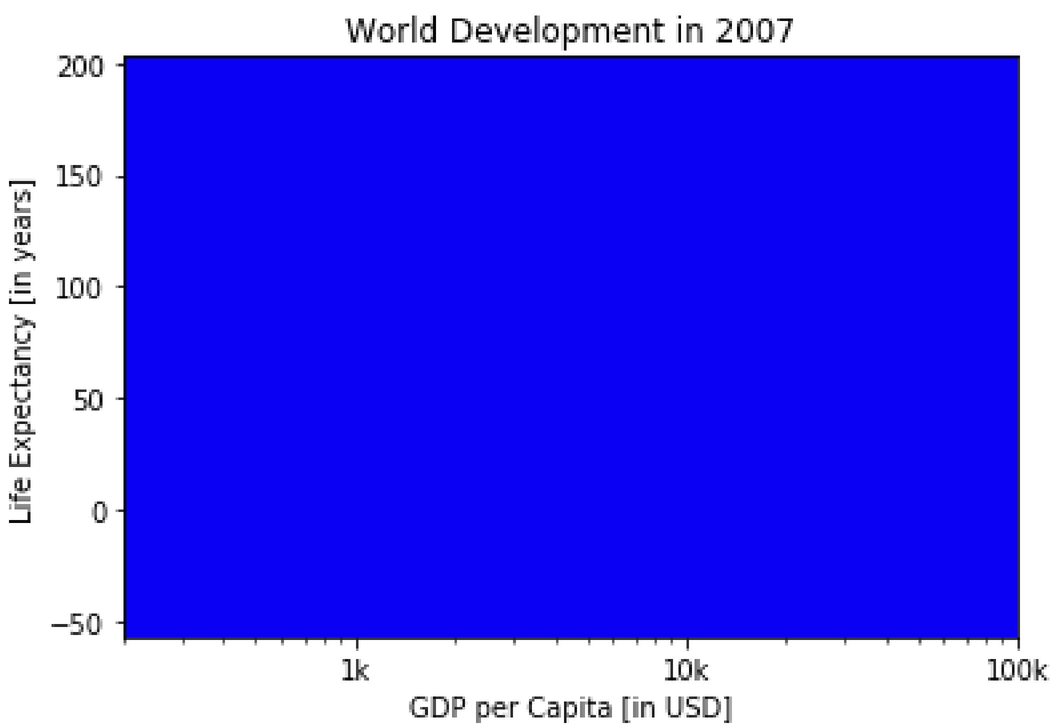
How did we make the list `col` you ask? The Gapminder data contains a list continent with the `continent` each country belongs to. A dictionary is constructed that maps continents onto colors:

```
dict = {
    'Asia':'red',
    'Europe':'green',
    'Africa':'blue',
    'Americas':'yellow',
    'Oceania':'black'
}
```

Change the opacity of the bubbles by setting the `alpha` argument to 0.8 inside `plt.scatter()`. Alpha can be set from 0-1, where 0 is totally transparent, and one is not at all transparent.

```
In [9]: # Set the data to use
        colors = {
            'Asia':'red',
            'Europe':'green',
            'Africa':'blue',
            'Americas':'yellow',
            'Oceania':'black'
```

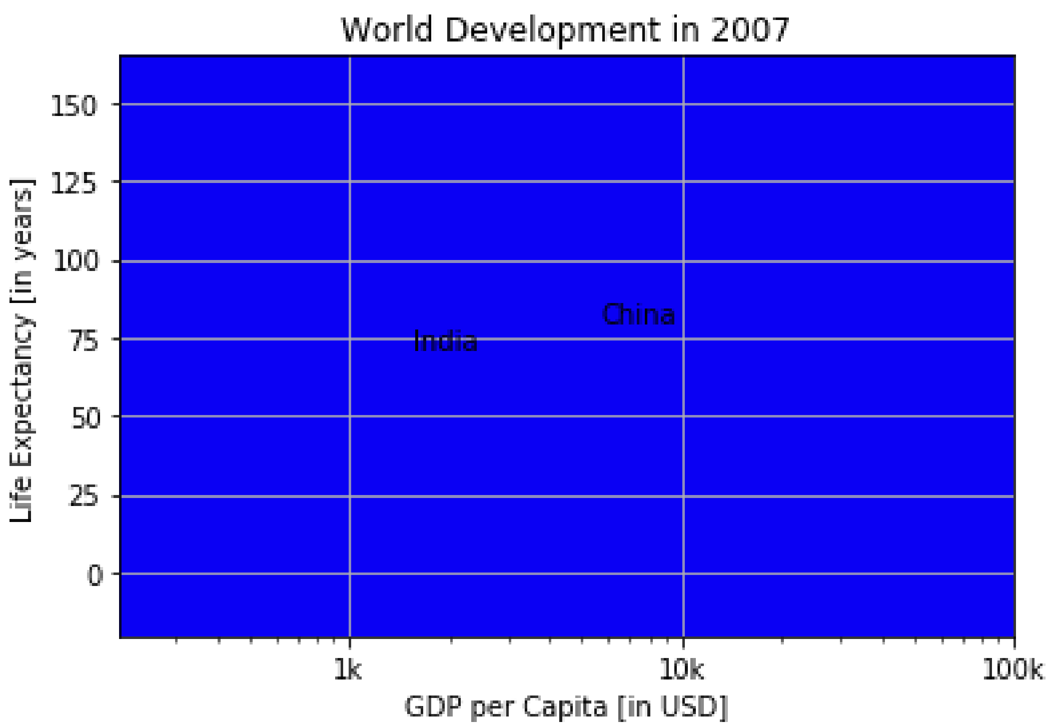
```
}
cont = list(gapminder['cont'])
col = []
for thisCont in cont:
    thisColor = colors[str(thisCont)]
    col.append(thisColor)

# Specify c and alpha inside plt.scatter()
plt.scatter(x = gdp_cap, y = life_exp, s = np.array(pop) * 2, c = col, alpha = 0.8)

# Previous customizations
plt.xscale('log')
plt.xlabel('GDP per Capita [in USD]')
plt.ylabel('Life Expectancy [in years]')
plt.title('World Development in 2007')
plt.xticks([1000,10000,100000], ['1k','10k','100k'])

# Show the plot
plt.show()
```



We can still customize even more with the

- `plt.text()` - Add text to the plot
- `plt.grid(True)` - Add grid lines to the plot

In [10]:
```
# Scatter plot
plt.scatter(x = gdp_cap, y = life_exp, s = np.array(pop), c = col, alpha = 0.8)

# Previous customizations
plt.xscale('log')
plt.xlabel('GDP per Capita [in USD]')
plt.ylabel('Life Expectancy [in years]')
plt.title('World Development in 2007')
plt.xticks([1000,10000,100000], ['1k','10k','100k'])

# Additional customizations
plt.text(1550, 71, 'India')
plt.text(5700, 80, 'China')

# Add grid() call
plt.grid(True)

# Show the plot
plt.show()
```



1.2 Dictionaries

Dictionaries are a great way to index/reference values and their corresponding values Below is an example, using lists, of a non-efficient way to index a variable and its' corresponding value.

To the the right capital of Germany, you have to index two different lists

In [11]:
```
# Definition of countries and capital
countries = ['spain', 'france', 'germany', 'norway']
capitals = ['madrid', 'paris', 'berlin', 'oslo']

# Get index of 'germany': ind_ger
ind_ger = countries.index('germany')

# Use ind_ger to print out capital of Germany
print(capitals[ind_ger])
```

```
berlin
```

Now here's how you create a Dictionary!!

```
my_dict = {
   "key1":"value1",
   "key2":"value2",
}
```

Each `key` has an assigned `value`

In [12]:
```
# Now here's a Dictionary!!

# Definition of countries and capital
countries = ['spain', 'france', 'germany', 'norway']
capitals = ['madrid', 'paris', 'berlin', 'oslo']

# From string in countries and capitals, create dictionary europe
europe = { 'spain':'madrid', 'france' : 'paris', 'germany' : 'berlin', 'norway' : 'os

# Print europe
print(europe)
```

```
{'spain': 'madrid', 'france': 'paris', 'germany': 'berlin', 'norway': 'oslo'}
```

You can index a value based on its' key. You can also get all the keys in a dictionary

In [13]:
```
# Definition of dictionary
europe = {'spain':'madrid', 'france':'paris', 'germany':'berlin', 'norway':'oslo' }

# Print out the keys in europe
print(europe.keys())

# Print out value that belongs to key 'norway'
print(europe['norway'])
```

```
dict_keys(['spain', 'france', 'germany', 'norway'])
oslo
```

Keys have to be "immutable objects", meaning they can't be changed once defined. Examples of immuatable objects include:

- strings
- floats
- booleans
- integers

However list contents can be changed so they can't be keys of dictionaries
LISTS VS DICTIONARIES:

- Lists are good when you have a collection of values and the order of those values matter
- Dictionaries are used as lookup tables

You can also append dictionaries and there are many different ways of referencing values or checking if keys exist

```
In [14]: # Definition of dictionary
         europe = {'spain':'madrid', 'france':'paris', 'germany':'berlin', 'norway':'oslo' }

         # Add italy to europe
         europe['italy'] = 'rome'

         # Check that the key 'italy' exists in the dictionary
         print('italy' in europe)

         # Add poland to europe
         europe['poland'] = 'warsaw'

         # Print europe
         print(europe)
```

```
True
{'spain': 'madrid', 'france': 'paris', 'germany': 'berlin', 'norway': 'oslo', 'italy': 'rome',
```

```
In [15]: # Definition of dictionary
         europe = {'spain':'madrid', 'france':'paris', 'germany':'bonn',
                   'norway':'oslo', 'italy':'rome', 'poland':'warsaw',
                   'australia':'vienna' }

         # Correct the value of germany
         europe['germany'] = 'berlin'

         # Remove australia from dict
```

```
del(europe['australia'])

# Print europe
print(europe)
```

```
{'spain': 'madrid', 'france': 'paris', 'germany': 'berlin', 'norway': 'oslo', 'italy': 'rome',
```

Remember lists? They could contain anything, even other lists. Well, for dictionaries the same holds. Dictionaries can contain key:value pairs where the values are again dictionaries.

It's perfectly possible to chain square brackets to select elements. To fetch the population for Spain from europe, for example, you need:

`europe['spain']['population']`

```
In [16]: # Dictionary of dictionaries
         europe = { 'spain': { 'capital':'madrid', 'population':46.77 },
                    'france': { 'capital':'paris', 'population':66.03 },
                    'germany': { 'capital':'berlin', 'population':80.62 },
                    'norway': { 'capital':'oslo', 'population':5.084 } }


         # Print out the capital of France
         print(europe['france']['capital'])

         # Create sub-dictionary data
         data = {'capital' : 'rome', 'population' : 59.83}

         # Add data to europe under key 'italy'
         europe['italy'] = data

         # Print europe
         print(europe)
```

```
paris
{'spain': {'capital': 'madrid', 'population': 46.77}, 'france': {'capital': 'paris', 'populati
```

2 Pandas

Pandas allows multiple data types in a single dataframe. It's built on numpy. Each column can old a different type

It's basically a way to store tabular data where you can label the rows and the columns. One way to build a DataFrame is from a dictionary.

Primarily done using `pd.DataFrame()`

2.1 Dataframes from lists

```
In [17]: # Pre-defined lists
         names = ['United States', 'Australia', 'Japan', 'India', 'Russia', 'Morocco', 'Egypt']
```

```
dr =  [True, False, False, False, True, True, True]
cpc = [809, 731, 588, 18, 200, 70, 45]

# Import pandas as pd
import pandas as pd

# Create dictionary my_dict with three key:value pairs: my_dict
my_dict = {'country' : names, 'drives_right' : dr, 'cars_per_cap' : cpc}

# Build a DataFrame cars from my_dict: cars
cars = pd.DataFrame(my_dict)

# Print cars
print(cars)
```

```
         country  drives_right  cars_per_cap
0  United States          True           809
1      Australia         False           731
2          Japan         False           588
3          India         False            18
4         Russia          True           200
5        Morocco          True            70
6          Egypt          True            45
```

2.2 row_labeling with `.index()`

Have you noticed that the row labels (i.e. the labels for the different observations) were automatically set to integers from 0 up to 6?

To solve this a list row_labels has been created. You can use it to specify the row labels of the cars DataFrame. You do this by setting the index attribute of cars, that you can access as cars.index.

```
In [18]: # Build cars DataFrame
         names = ['United States', 'Australia', 'Japan', 'India', 'Russia', 'Morocco', 'Egypt']
         dr =  [True, False, False, False, True, True, True]
         cpc = [809, 731, 588, 18, 200, 70, 45]
         dict = { 'country':names, 'drives_right':dr, 'cars_per_cap':cpc }
         cars = pd.DataFrame(dict)
         print(cars)

         # Definition of row_labels
         row_labels = ['US', 'AUS', 'JAP', 'IN', 'RU', 'MOR', 'EG']

         # Specify row labels of cars
         cars.index = row_labels

         # Print cars again
         print(cars)
```

```
         country  drives_right  cars_per_cap
0  United States          True           809
1      Australia         False           731
2          Japan         False           588
3          India         False            18
4         Russia          True           200
5        Morocco          True            70
6          Egypt          True            45
           country  drives_right  cars_per_cap
US   United States          True           809
AUS      Australia         False           731
JAP          Japan         False           588
IN           India         False            18
RU          Russia          True           200
MOR        Morocco          True            70
EG           Egypt          True            45
```

2.3 Importing csv files

To import CSV data into Python as a Pandas DataFrame you can use `read_csv()`

```
In [19]: # Import the cars.csv data: cars
         cars = pd.read_csv('cars.csv')

         # Print out cars
         print(cars)
```

```
  Unnamed: 0  cars_per_cap        country  drives_right
0         US           809  United States          True
1        AUS           731      Australia         False
2        JAP           588          Japan         False
3         IN            18          India         False
4         RU           200         Russia          True
5        MOR            70        Morocco          True
6         EG            45          Egypt          True
```

Your `read_csv()` call to import the CSV data didn't generate an error, but the output is not entirely what we wanted. The row labels were imported as another column without a name.

Remember `index_col`, an argument of `read_csv()`, that you can use to specify which column in the CSV file should be used as a row label? Well, that's exactly what you need here!

```
In [20]: # Fix import by including index_col
         cars = pd.read_csv('cars.csv', index_col = 0)

         # Print out cars
         print(cars)
```

```
     cars_per_cap        country  drives_right
US            809  United States          True
AUS           731      Australia         False
JAP           588          Japan         False
IN             18          India         False
RU            200         Russia          True
MOR            70        Morocco          True
EG             45          Egypt          True
```

2.4 Advanced indexing using `loc()` and `iloc()`

The simplest way to index is using brackets `[]`, but it's not the most powerful way. It also has an issue with the data type it outputs, a Pandas series. You can use two brackets `[[]]` to output a Pandas dataframe

- `[]` - Outputs Pandas series
- `[[]]` - Outputs Pandas DataFrame

In [21]:
```
# Import cars data
cars = pd.read_csv('cars.csv', index_col = 0)

# Print out country column as Pandas Series
print(cars['country'])

# Print out country column as Pandas DataFrame
print(cars[['country']])

# Print out DataFrame with country and drives_right columns
print(cars[['country', 'drives_right']])
```

```
US     United States
AUS        Australia
JAP            Japan
IN             India
RU            Russia
MOR          Morocco
EG             Egypt
Name: country, dtype: object
           country
US   United States
AUS      Australia
JAP          Japan
IN           India
RU          Russia
MOR        Morocco
EG           Egypt
           country  drives_right
US   United States          True
```

```
AUS      Australia         False
JAP          Japan         False
IN           India         False
RU          Russia          True
MOR        Morocco          True
EG           Egypt          True
```

You can also index rows using square brackets [] and the rows you want to output

```
In [22]: # Print out first 3 observations
         print(cars[0:3])

         # Print out fourth, fifth and sixth observation
         print(cars[3:6])
```

```
     cars_per_cap        country  drives_right
US            809  United States          True
AUS           731      Australia         False
JAP           588          Japan         False
     cars_per_cap  country  drives_right
IN             18    India         False
RU            200   Russia          True
MOR            70  Morocco          True
```

2.4.1 `loc()` and `iloc()`

With `loc` and `iloc` you can do practically any data selection operation on DataFrames you can think of.

- `loc` - label-based, which means that you have to specify rows and columns based on their row and column labels.
- `iloc` - integer index based, so you have to specify rows and columns by their integer index like you did in the previous exercise.

Each pair of commands here gives the same result.

```
cars.loc['RU']
cars.iloc[4]

cars.loc[['RU']]
cars.iloc[[4]]

cars.loc[['RU', 'AUS']]
cars.iloc[[4, 1]]
```

```
In [23]: # Output Pandas Series
         cars.loc['RU']
         cars.iloc[4]
```

```
Out[23]: cars_per_cap       200
         country         Russia
         drives_right      True
         Name: RU, dtype: object
```

```
In [24]: # Output Pandas DataFrame
         cars.loc[['RU']]
         cars.iloc[[4]]
```

```
Out[24]:     cars_per_cap country  drives_right
         RU           200  Russia          True
```

```
In [25]: # Output 2 indices in a single DataFrame object
         cars.loc[['RU', 'AUS']]
         cars.iloc[[4, 1]]
```

```
Out[25]:      cars_per_cap    country  drives_right
         RU            200     Russia          True
         AUS           731  Australia         False
```

```
In [26]: # Print out observation for Japan
         print(cars.loc['JAP'])

         # Print out observations for Australia and Egypt
         print(cars.loc[['AUS', 'EG']])
```

```
cars_per_cap      588
country         Japan
drives_right    False
Name: JAP, dtype: object
     cars_per_cap    country  drives_right
AUS           731  Australia         False
EG             45      Egypt          True
```

loc and iloc also allow you to select both rows and columns from a DataFrame. Below are paired commands produce the same result.

```
cars.loc['IN', 'cars_per_cap']
cars.iloc[3, 0]

cars.loc[['IN', 'RU'], 'cars_per_cap']
cars.iloc[[3, 4], 0]

cars.loc[['IN', 'RU'], ['cars_per_cap', 'country']]
cars.iloc[[3, 4], [0, 1]]
```

```
In [27]: # Produce a single integer
         cars.loc['IN', 'cars_per_cap']
         cars.iloc[3, 0]
```

```
Out[27]: 18
```

```
In [28]: # Get the column for two different indices
         cars.loc[['IN', 'RU'], 'cars_per_cap']
         cars.iloc[[3, 4], 0]
```

```
Out[28]: IN     18
         RU    200
         Name: cars_per_cap, dtype: int64
```

```
In [29]: # Output DataFrame of indices and corresponding info
         cars.loc[['IN', 'RU'], ['cars_per_cap', 'country']]
         cars.iloc[[3, 4], [0, 1]]
```

```
Out[29]:     cars_per_cap country
         IN            18   India
         RU           200  Russia
```

```
In [30]: # Print out drives_right value of Morocco
         print(cars.loc['MOR', 'drives_right'])

         # Print sub-DataFrame
         print(cars.loc[['RU', 'MOR'], ['country', 'drives_right']])
```

```
True
     country  drives_right
RU    Russia          True
MOR  Morocco          True
```

It's also possible to select only columns with `loc` and `iloc`. In both cases, you simply put a slice going from beginning to end in front of the comma:

```
cars.loc[:, 'country']
cars.iloc[:, 1]

cars.loc[:, ['country','drives_right']]
cars.iloc[:, [1, 2]]
```

```
In [31]: cars.loc[:, 'country']
         cars.iloc[:, 1]
```

```
Out[31]: US     United States
         AUS        Australia
         JAP            Japan
         IN             India
         RU            Russia
         MOR          Morocco
         EG             Egypt
         Name: country, dtype: object
```

```
In [32]: cars.loc[:, ['country','drives_right']]
         cars.iloc[:, [1, 2]]
```

```
Out[32]:            country  drives_right
         US   United States          True
         AUS      Australia         False
         JAP          Japan         False
         IN           India         False
         RU          Russia          True
         MOR        Morocco          True
         EG           Egypt          True
```

```
In [33]: # Print out drives_right column as Series
         print(cars.loc[:, 'drives_right'])

         # Print out drives_right column as DataFrame
         print(cars.loc[:, ['drives_right']])

         # Print out cars_per_cap and drives_right as DataFrame
         print(cars.loc[:,['cars_per_cap', 'drives_right']])
```

```
US      True
AUS    False
JAP    False
IN     False
RU      True
MOR     True
EG      True
Name: drives_right, dtype: bool
     drives_right
US           True
AUS         False
JAP         False
IN          False
RU           True
MOR          True
EG           True
     cars_per_cap  drives_right
US            809          True
AUS           731         False
JAP           588         False
IN             18         False
RU            200          True
MOR            70          True
EG             45          True
```

3 Logic, Control Flow, Filtering

3.1 Comparison Operators

Very simple and done before

In [34]:
```
# Comparison of booleans
print(True == False)

# Comparison of integers
print(-5*15 != 75)

# Comparison of strings
print("pyscript" == "PyScript")

# Compare a boolean with an integer
print(1 == True)
```

```
False
True
False
True
```

In [35]:
```
# Comparison of integers
x = -3 * 6
print(x >= -10)

# Comparison of strings
y = "test"
"test" <= y

# Comparison of booleans
True > False
```

```
False
```

Out[35]: True

You can also easily compare arrays to each other

In [36]:
```
# Create arrays
import numpy as np
my_house = np.array([18.0, 20.0, 10.75, 9.50])
your_house = np.array([14.0, 24.0, 14.25, 9.0])

# my_house greater than or equal to 18
print(my_house >= 18)
```

```
# my_house less than your_house
print(my_house < your_house)
```

```
[ True  True False False]
[False  True  True False]
```

Numpy also has capabilities for this that makes it easier and/or more powerful
You can also use boolean operators `and`, `or`, `not`

```
In [37]: # Define variables
         my_kitchen = 18.0
         your_kitchen = 14.0

         # my_kitchen bigger than 10 and smaller than 18?
         print(my_kitchen > 10 and my_kitchen < 18)

         # my_kitchen smaller than 14 or bigger than 17?
         print(my_kitchen < 14 or my_kitchen > 17)

         # Double my_kitchen smaller than triple your_kitchen?
         print(my_kitchen*2 < your_kitchen *3)
```

```
False
True
True
```

To use with numpy, you have to use:

- `np.logical_and()`
- `np.logical_or()`
- `np.logical_not()`

Ex:

```
np.logical_and(your_house > 13,
               your_house < 15)
```

```
In [38]: # Create arrays
         my_house = np.array([18.0, 20.0, 10.75, 9.50])
         your_house = np.array([14.0, 24.0, 14.25, 9.0])

         # my_house greater than 18.5 or smaller than 10
         print(np.logical_or(my_house > 18.5, my_house < 10))

         # Both my_house and your_house smaller than 11
         print(np.logical_and(my_house < 11, your_house < 11))
```

```
[False  True False  True]
[False False False  True]
```

3.2 if, else, elif

```
In [39]: # Define variables
         room = "bed"
         area = 14.0

         # if-elif-else construct for room
         if room == "kit" :
             print("looking around in the kitchen.")
         elif room == "bed":
             print("looking around in the bedroom.")
         else :
             print("looking around elsewhere.")

         # if-elif-else construct for area
         if area > 15 :
             print("big place!")
         elif area > 10:
             print("medium size, nice!")
         else :
             print("pretty small.")
```

```
looking around in the bedroom.
medium size, nice!
```

3.3 Filtering Pandas DataFrames

```
In [40]: # Extract drives_right column as Series: dr
         dr = cars['drives_right'] == True

         # Use dr to subset cars: sel
         sel = cars[dr]

         # Print sel
         print(sel)
```

```
     cars_per_cap        country  drives_right
US            809  United States          True
RU            200         Russia          True
MOR            70        Morocco          True
EG             45          Egypt          True
```

Above, we can skip making the dr variable

```
In [41]: # Convert code to a one-liner
         sel = cars[cars['drives_right']]
```

```
# Print sel
print(sel)
```

```
     cars_per_cap        country  drives_right
US            809  United States          True
RU            200         Russia          True
MOR            70        Morocco          True
EG             45          Egypt          True
```

```
In [42]: # Create car_maniac: observations that have a cars_per_cap over 500
         cpc = cars['cars_per_cap']
         many_cars = cpc > 500
         car_maniac = cars[many_cars]

         # Print car_maniac
         print(car_maniac)
```

```
     cars_per_cap        country  drives_right
US            809  United States          True
AUS           731      Australia         False
JAP           588          Japan         False
```

You can also use the numpy logical operators as before

```
In [43]: # Create medium: observations with cars_per_cap between 100 and 500
         cpc = cars['cars_per_cap']
         between = np.logical_and(cpc > 100, cpc < 500)
         medium = cars[between]

         # Print medium
         print(medium)
```

```
    cars_per_cap country  drives_right
RU           200  Russia          True
```

3.4 Loops

3.4.1 While Loop

```
In [44]: # Initialize offset
         offset = 8

         # Code the while loop
         while offset != 0:
             print("correcting...")
             offset = offset - 1
             print(offset)
```

```
correcting...
7
correcting...
6
correcting...
5
correcting...
4
correcting...
3
correcting...
2
correcting...
1
correcting...
0
```

3.4.2 For Loop

In [45]:
```python
# areas list
areas = [11.25, 18.0, 20.0, 10.75, 9.50]

for a in areas :
    print(a)
```

```
11.25
18.0
20.0
10.75
9.5
```

Enumerate Using a for loop to iterate over a list only gives you access to every list element in each run, one after the other. If you also want to access the index information, so where the list element you're iterating over is located, you can use `enumerate()`.

As an example, have a look at how the for loop from the video was converted:

```
fam = [1.73, 1.68, 1.71, 1.89]
for index, height in enumerate(fam) :
    print("person " + str(index)
```

In [46]:
```python
# areas list
areas = [11.25, 18.0, 20.0, 10.75, 9.50]

# Change for loop to use enumerate() and update print()
for index, area in enumerate(areas) :
    print("room " + str(index) + ": " + str(area))
```

```
room 0: 11.25
room 1: 18.0
room 2: 20.0
room 3: 10.75
room 4: 9.5
```

```
In [47]: # house list of lists
         house = [["hallway", 11.25],
                  ["kitchen", 18.0],
                  ["living room", 20.0],
                  ["bedroom", 10.75],
                  ["bathroom", 9.50]]

         # Build a for loop from scratch
         for room in house:
             print("the " + room[0] + " is" + str(room[1]) + " sqm")
```

```
the hallway is11.25 sqm
the kitchen is18.0 sqm
the living room is20.0 sqm
the bedroom is10.75 sqm
the bathroom is9.5 sqm
```

3.4.3 Looping over Data Structures

For Dictionaries and Numpy Arrays

For Dictionaries you need the `items()` method

- Note: Dictionaries are inherently unordered, so the order of output from looping over a dictionary may not match the sequence within the dictionary itself

```
world = { "afghanistan":30.55,
          "albania":2.77,
          "algeria":39.21 }

for key, value in world.items() :
    print(key + " -- " + str(value))
```

For 1D Numpy Arrays

`for x in my_array :`

For 2D Numpy Arrays

`for x in np.nditer(my_array) :`

A 2D numpy array is made up of single numpy arrays. Iterating over a 2D array will print out one of the entire 1D arrays that composes it

```
In [48]: # Definition of dictionary
         europe = {'spain':'madrid', 'france':'paris', 'germany':'berlin',
```

```
                'norway':'oslo', 'italy':'rome', 'poland':'warsaw', 'austria':'vienna' }

    # Iterate over europe
    # Note how order of output is NOT in a particular order
    for key, value in europe.items():
        print("the capital of " + key + " is " + value)
```

```
the capital of spain is madrid
the capital of france is paris
the capital of germany is berlin
the capital of norway is oslo
the capital of italy is rome
the capital of poland is warsaw
the capital of austria is vienna
```

```
In [49]: # For loop over my_house
         for x in my_house:
             print(str(x) + " area")


         our_houses = np.array([my_house, your_house])
         # For loop over our_houses
         for x in np.nditer(our_houses):
             print(str(x), end = "---")

         print("\n")
         # Note, the default output of the end argument is \n for new line

         # Printing a 2D numpy array with regular for loop
         for x in our_houses:
             print(str(x) + " area")
```

```
18.0 area
20.0 area
10.75 area
9.5 area
18.0---20.0---10.75---9.5---14.0---24.0---14.25---9.0---

[18.   20.   10.75  9.5 ] area
[14.   24.   14.25  9.  ] area
```

3.4.4 Looping over Pandas DataFrames

Need to explicitly say you want to iterate over rows using `iterrows()` method
Or you can use the `apply()` method
Use one depending on needs

In [50]:
```
# Import cars data
cars = pd.read_csv('cars.csv', index_col = 0)
# index_col sets the index of dataframe to be something other than numbers 0 to n-1

# Iterate over rows of cars
for lab, row in cars.iterrows():
    print(lab, end = '-----\n') # Prints label, the row identifier/index
    print(row) # Prints row contents
```

```
US-----
cars_per_cap              809
country         United States
drives_right             True
Name: US, dtype: object
AUS-----
cars_per_cap          731
country         Australia
drives_right        False
Name: AUS, dtype: object
JAP-----
cars_per_cap      588
country         Japan
drives_right    False
Name: JAP, dtype: object
IN-----
cars_per_cap       18
country         India
drives_right    False
Name: IN, dtype: object
RU-----
cars_per_cap       200
country         Russia
drives_right      True
Name: RU, dtype: object
MOR-----
cars_per_cap        70
country         Morocco
drives_right       True
Name: MOR, dtype: object
EG-----
cars_per_cap       45
country         Egypt
drives_right     True
Name: EG, dtype: object
```

In [51]:
```
# Can make it so only SOME of the rows contents are output using indexing
for lab, row in cars.iterrows() :
    print(lab + ": " + str(row['cars_per_cap']))
```

```
US: 809
AUS: 731
JAP: 588
IN: 18
RU: 200
MOR: 70
EG: 45
```

```
In [52]: # Adding column to DataFrame using .loc method
         # Import cars data
         import pandas as pd
         cars = pd.read_csv('cars.csv', index_col = 0)

         # Code for loop that adds COUNTRY column
         for lab, row in cars.iterrows():
             cars.loc[lab, 'COUNTRY'] = row['country'].upper()

         # Print cars
         print(cars)
```

```
     cars_per_cap        country  drives_right        COUNTRY
US            809  United States          True  UNITED STATES
AUS           731      Australia         False      AUSTRALIA
JAP           588          Japan         False          JAPAN
IN             18          India         False          INDIA
RU            200         Russia          True         RUSSIA
MOR            70        Morocco          True        MOROCCO
EG             45          Egypt          True          EGYPT
```

Using `iterrows()` to iterate over every observation of a Pandas DataFrame is easy to understand, but not very efficient. On every iteration, you're creating a new Pandas Series.

If you want to add a column to a DataFrame by calling a function on another column, the `iterrows()` method in combination with a for loop is not the preferred way to go. Instead, you'll want to use `apply()`.

Compare the `iterrows()` version with the `apply()` version to get the same result in the brics DataFrame:

```
for lab, row in brics.iterrows() :
    brics.loc[lab, "name_length"] = len(row["country"])

brics["name_length"] = brics["country"].apply(len)
```

```
In [53]: cars = pd.read_csv('cars.csv', index_col = 0)

         # Use .apply(str.upper)
         # It's a bit different as upper is a method
         cars['COUNTRY'] = cars['country'].apply(str.upper)
```

```
print(cars)
```

```
     cars_per_cap        country  drives_right        COUNTRY
US            809  United States          True  UNITED STATES
AUS           731      Australia         False      AUSTRALIA
JAP           588          Japan         False          JAPAN
IN             18          India         False          INDIA
RU            200         Russia          True         RUSSIA
MOR            70        Morocco          True        MOROCCO
EG             45          Egypt          True          EGYPT
```

3.5 Blending It All Together

For a series of events, you can calculate chance of an outcome:

1. Calculate chance Analytically
2. Simulate it many times and see what fraction of simulations the outcome occurs
   - This is called Hacker Statistics!

Random numbers in numpy
`np.random.rand()` - Generates pseudo-random numbers
`np.random.seed(123)` - Can set seed of pseudo-random numbers (i.e. make it so calling the same seed produces the same results). It ensures reproducibility.
Another is `np.random.randint()`

```
In [54]: # Set the seed
         np.random.seed(123)

         # Generate and print random float
         print(np.random.rand())

         # Set the seed
         np.random.seed(123)

         # Generate and print random float
         print(np.random.rand())
```

```
0.6964691855978616
0.6964691855978616
```

```
In [55]: # Use randint() to simulate a dice
         # Like everything in python, the last number is EXclusive
         import numpy as np
         print(np.random.randint(1,7))
```

```
# Use randint() again
print(np.random.randint(1,7))
```

```
3
5
```

```
In [56]: # Numpy is imported, seed is set
         np.random.seed(123)

         # Starting step
         step = 50

         # Roll the dice
         dice = np.random.randint(1,7)

         # Finish the control construct
         if dice <= 2 :
             step = step - 1
         elif dice >= 3 and dice <= 5 :
             step = step + 1
         else :
             step = step + np.random.randint(1,7)

         # Print out dice and step
         print(dice)
         print(step)
```

```
6
53
```

If you throw dice 100 times to determine an outcome, you'd have a succession of random steps, or a *random walk*. Well known concept in Science. For a Random Walk, each next sequence has to be based on the previous outcomes.

Flipping a coin is random, but the sequence isn't a random walk.

A Random Walk Game: You and a friend are playing a game of who can reach the top of the stairs first. Rolling 1 or 2 means a step down, rolling 3-5 is a step up, rolling a 6 means rerolling and stepping upward equal to the amount rerolled. This is the code above. THIS is a random walk because the step you are on each turn is dependent upon the previous position, but each roll is random

Below is the random walk game again

```
In [57]: # Numpy is imported, seed is set
         import numpy as np
         np.random.seed(123)

         # Initialize random_walk
```

```
random_walk = [0]

# Complete the ___
for x in range(100) :
    # Set step: last element in random_walk
    step = random_walk[-1]

    # Roll the dice
    dice = np.random.randint(1,7)

    # Determine next step
    if dice <= 2:
        step = step - 1
    elif dice <= 5:
        step = step + 1
    else:
        step = step + np.random.randint(1,7)

    # append next_step to random_walk
    random_walk.append(step)

# Print random_walk
print(random_walk)
```

```
[0, 3, 4, 5, 4, 5, 6, 7, 6, 5, 4, 3, 2, 1, 0, -1, 0, 5, 4, 3, 4, 3, 4, 5, 6, 7, 8, 7, 8, 7, 8,
```

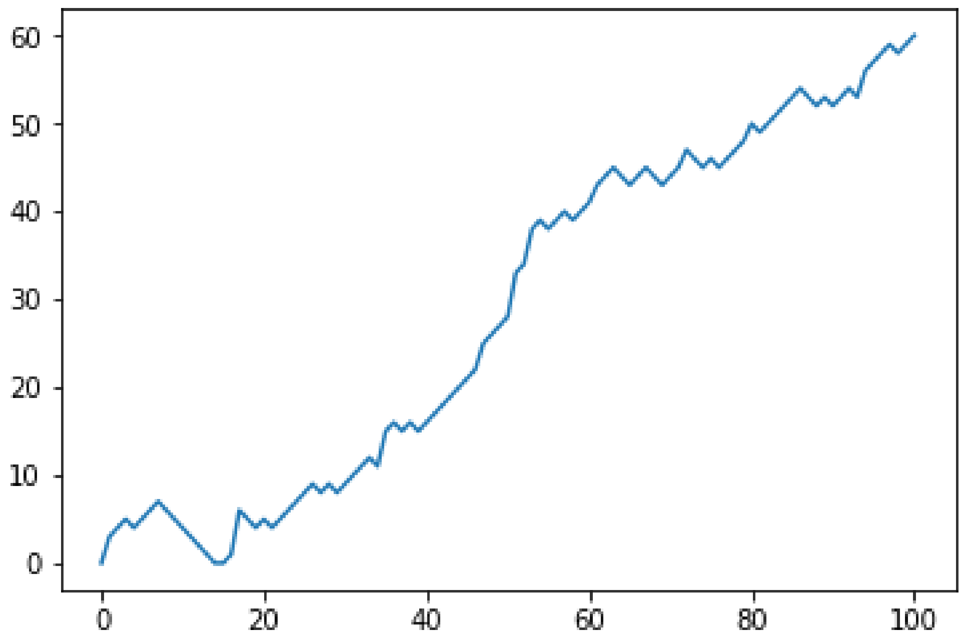
To make sure that `step` doesn't go below zero (as there are no negative steps) you can use the `max()` function in the if-elif-else chain

```
if dice <= 2:
        # Replace below: use max to make sure step can't go below 0
        step = max(0,step - 1)
    elif dice <= 5:
        step = step + 1
    else:
        step = step + np.random.randint(1,7)
```

We can also show the outcome of random walk using matplotlib

```
In [58]: np.random.seed(123)
         # Initialization
         random_walk = [0]

         for x in range(100) :
             step = random_walk[-1]
             dice = np.random.randint(1,7)

             if dice <= 2:
```

```
            step = max(0, step - 1)
        elif dice <= 5:
            step = step + 1
        else:
            step = step + np.random.randint(1,7)

        random_walk.append(step)

    # Import matplotlib.pyplot as plt
    import matplotlib.pyplot as plt

    # Plot random_walk
    plt.plot(random_walk)

    # Show the plot
    plt.show()
```



3.5.1 Distribution of Random Walk

What is the chance that you end up on the 60th step? If you simulate this game 10,000 times, you will get 10,000 endpoints. It's actually a distribution of final steps! You can then visualize how often it occurs that the final step is the 60th. In other words, in 10,000 games, how often do you land on the 60th step?

```
In [59]: # Numpy is imported; seed is set
```

```
# Initialize all_walks
all_walks = []

# Simulate random walk 10 times
for i in range(10) :

    # Code from before
    random_walk = [0]
    for x in range(100) :
        step = random_walk[-1]
        dice = np.random.randint(1,7)

        if dice <= 2:
            step = max(0, step - 1)
        elif dice <= 5:
            step = step + 1
        else:
            step = step + np.random.randint(1,7)
        random_walk.append(step)

    # Append random_walk to all_walks
    all_walks.append(random_walk)

# Print all_walks
print(all_walks)
```

```
[[0, 4, 3, 2, 4, 3, 4, 6, 7, 8, 13, 12, 13, 14, 15, 16, 17, 16, 21, 22, 23, 24, 23, 22, 21, 20
```

Below are graphed the results of the game.

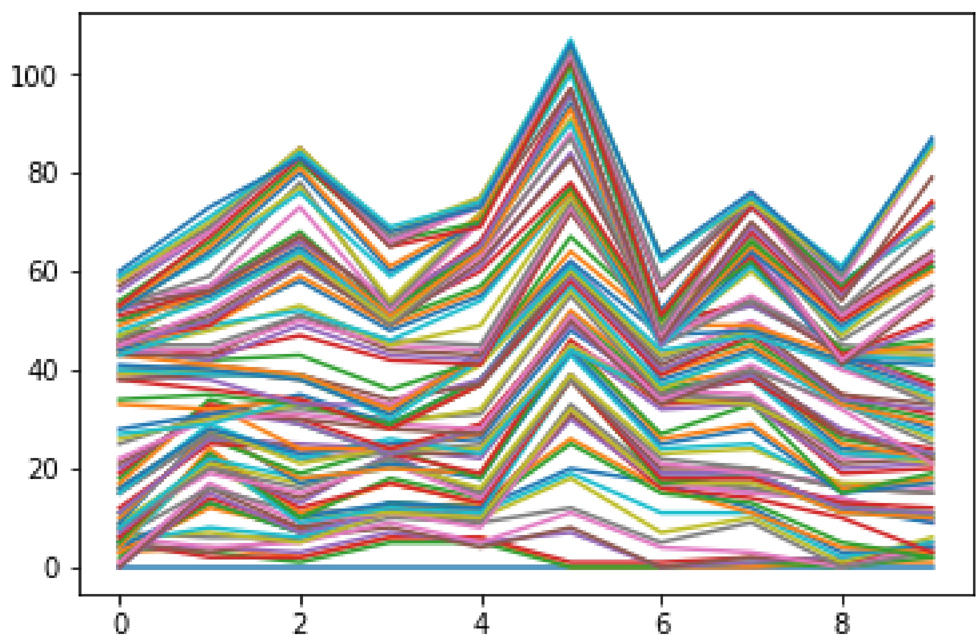
The variable `all_walks` is a list containing more lists. Each sub-list is one possible sequence of the game. Each sub-list contains 100 elements. There are 10 sub-lists

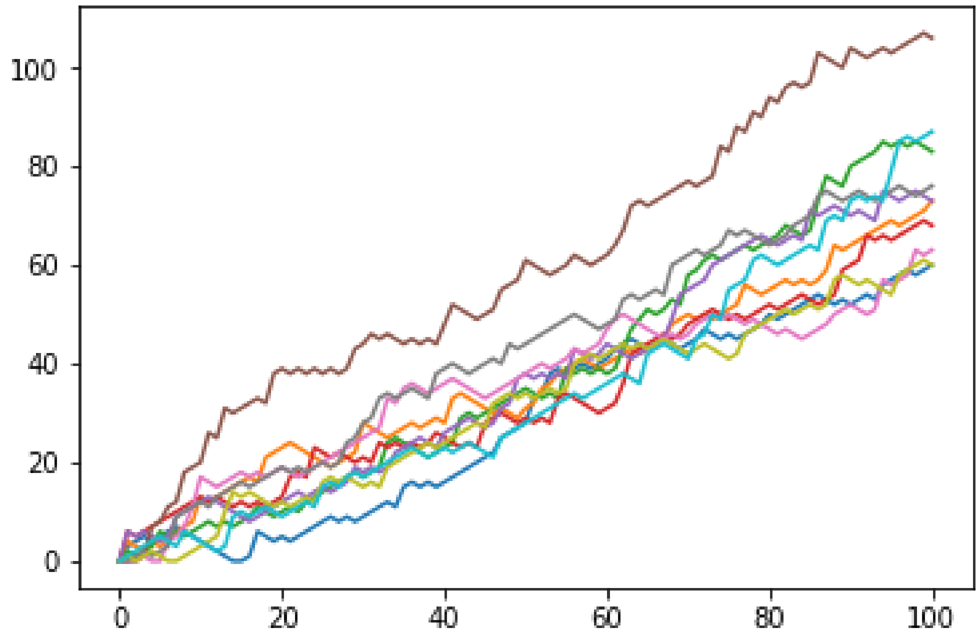
Conversely,

`np_aw_t`, the transposed version of `all_walks`, contains 100 sub-lists each with 10 elements. Every row represents the position after 1 throw for the 10 random walks.

```
In [60]: # numpy and matplotlib imported, seed set.
         np.random.seed(123)

         # initialize and populate all_walks
         all_walks = []
         for i in range(10) :
             random_walk = [0]
             for x in range(100) :
                 step = random_walk[-1]
                 dice = np.random.randint(1,7)
                 if dice <= 2:
                     step = max(0, step - 1)
```

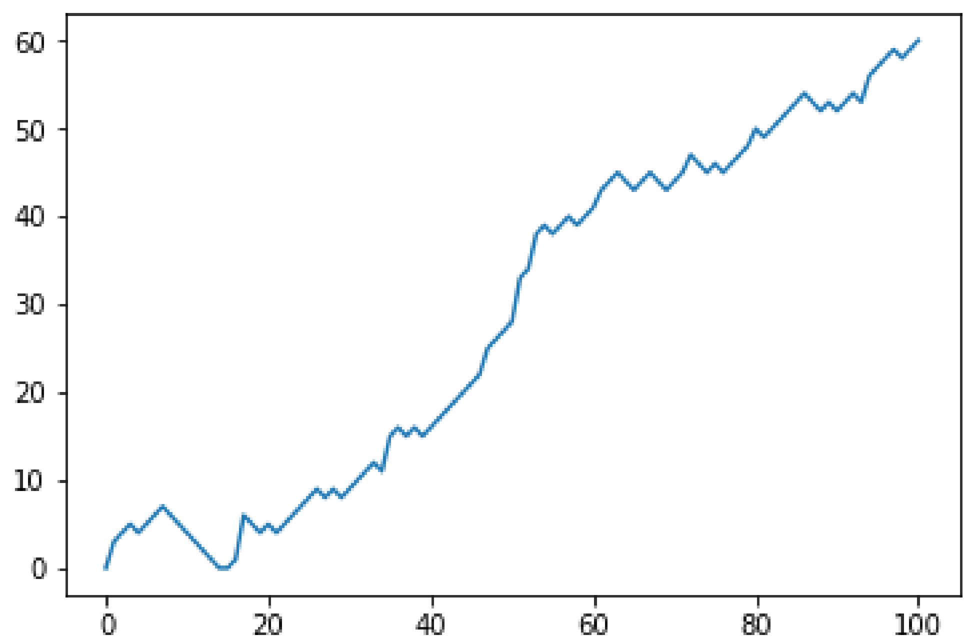
```
        elif dice <= 5:
            step = step + 1
        else:
            step = step + np.random.randint(1,7)
        random_walk.append(step)
    all_walks.append(random_walk)

# Convert all_walks to Numpy array: np_aw
np_aw = np.array(all_walks)

# Plot np_aw and show
plt.plot(np_aw)
plt.show()

# Clear the figure
plt.clf()

# Transpose np_aw: np_aw_t
np_aw_t = np.transpose(np_aw)

# Plot np_aw_t and show
plt.plot(np_aw_t)
plt.show()
```





Above, you can see the results. `plt.plot()` treats the first element of each sub-list as part of a group, and those numbers are what form a single line.

```
In [61]: plt.plot(np_aw[0])
         plt.show()
```

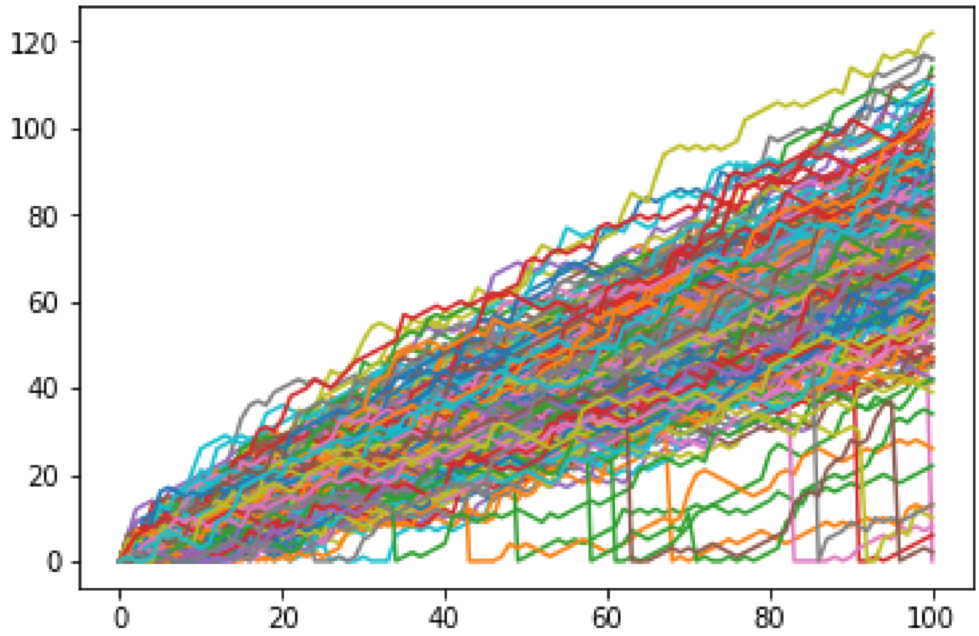


```
In [62]: # The first list is the beginning step
         # The second list is what the outcome of rolling the die once is etc.
         print(np_aw_t)
         print(len(np_aw_t))
```

```
[[ 0  0  0 ...  0  0  0]
 [ 3  4  2 ...  0  1  1]
 [ 4  3  1 ...  1  0  2]
 ...
 [58 70 85 ... 74 60 85]
 [59 71 84 ... 75 61 86]
 [60 73 83 ... 76 60 87]]
101
```

There's still something we forgot! You're a bit clumsy and you have a 0.1% chance of falling down. That calls for another random number generation. Basically, you can generate a random float between 0 and 1. If this value is less than or equal to 0.001, you should reset step to 0.

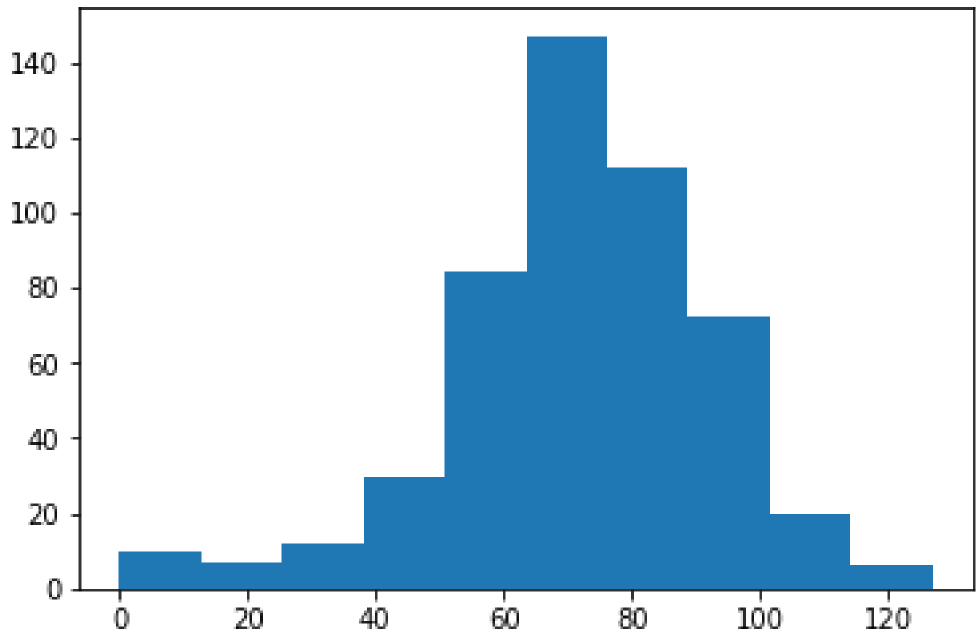
```
In [63]: # numpy and matplotlib imported, seed set
         np.random.seed(123)

         # Simulate random walk 250 times
         all_walks = []
         for i in range(250) :
             random_walk = [0]
             for x in range(100) :
                 step = random_walk[-1]
                 dice = np.random.randint(1,7)
                 if dice <= 2:
                     step = max(0, step - 1)
                 elif dice <= 5:
                     step = step + 1
                 else:
                     step = step + np.random.randint(1,7)

                 # Implement clumsiness
                 if np.random.rand() <= 0.001 :
                     step = 0

                 random_walk.append(step)
             all_walks.append(random_walk)

         # Create and plot np_aw_t
         np_aw_t = np.transpose(np.array(all_walks))
         plt.plot(np_aw_t)
         plt.show()
```



We still don't know the chance of going to the 60th step. Let's plot the distribution of the last step across trials

```
In [64]: # numpy and matplotlib imported, seed set
         np.random.seed(123)

         # Simulate random walk 500 times
         all_walks = []
         for i in range(500) :
             random_walk = [0]
             for x in range(100) :
                 step = random_walk[-1]
                 dice = np.random.randint(1,7)
                 if dice <= 2:
                     step = max(0, step - 1)
                 elif dice <= 5:
                     step = step + 1
                 else:
                     step = step + np.random.randint(1,7)
                 if np.random.rand() <= 0.001 :
                     step = 0
                 random_walk.append(step)
             all_walks.append(random_walk)

         # Create and plot np_aw_t
         np_aw_t = np.transpose(np.array(all_walks))
```

```
# Select last row from np_aw_t: ends
ends = np_aw_t[-1]

# Plot histogram of ends, display plot
plt.hist(ends)
plt.show()
```



The histogram above was created from a Numpy array ends, that contains 500 integers. Each integer represents the end point of a random walk. To calculate the chance that this end point is greater than or equal to 60, you can count the number of integers in ends that are greater than or equal to 60 and divide that number by 500, the total number of simulations.

```
In [65]: print( sum(ends >= 60)/500 * 100 )
```

```
78.4
```

```
In [ ]:
```