

Intermediate Python

January 9, 2020

1 Intermediate Python

Course Description

Learning Python is crucial for any aspiring data science practitioner. Learn to visualize real data with Matplotlib's functions and get acquainted with data structures such as the dictionary and the pandas DataFrame. After covering key concepts such as boolean logic, control flow, and loops in Python, you'll be ready to blend together everything you've learned to solve a case study using hacker statistics.

1.0.1 1. Matplotlib

Data visualization is a key skill for aspiring data scientists. Matplotlib makes it easy to create meaningful and insightful plots. In this chapter, you'll learn how to build various types of plots, and customize them to be more visually appealing and interpretable.

1.0.2 1.1 Line plot (1)

With matplotlib, you can create a bunch of different plots in Python. The most basic plot is the line plot. A general recipe is given here. `~~~ import matplotlib.pyplot as plt plt.plot(x,y) plt.show()`
~~~~

In the video, you already saw how much the world population has grown over the past years. Will it continue to do so? The world bank has estimates of the world population for the years 1950 up to 2100. The years are loaded in your workspace as a list called `year`, and the corresponding populations as a list called `pop`.

This course touches on a lot of concepts you may have forgotten, so if you ever need a quick refresher, download the Python for data science Cheat Sheet and keep it handy!

### Instructions

- `print()` the last item from both the `year` and the `pop` list to see what the predicted population for the year 2100 is. Use two `print()` functions.
- Before you can start, you should import `matplotlib.pyplot` as `plt`. `pyplot` is a sub-package of `matplotlib`, hence the dot.
- Use `plt.plot()` to build a line plot. `year` should be mapped on the horizontal axis, `pop` on the vertical axis. Don't forget to finish off with the `show()` function to actually display the plot.

```
[1]: #import pandas as pd
#gapminder = pd.read_csv("../Data/gapminder.csv")

# df = gapminder[["Year", "population"]]
#df_grouped = df.groupby('Year').sum() / 1e9
#year = df_grouped.index.to_list()
#pop = df_grouped.population.to_list()

# pop_year = gapminder.groupby('Year')['population'].agg('sum') / 1e9
#year = pop_year.index
#pop = pop_year.to_list()

import numpy as np

pop = np.genfromtxt('../Data/pop.txt')
pop = pop.tolist()

year = np.genfromtxt('../Data/year.txt')
year = year.tolist()
```

```
[2]: # Print the last item from year and pop
print(year[-1])
print(pop[-1])

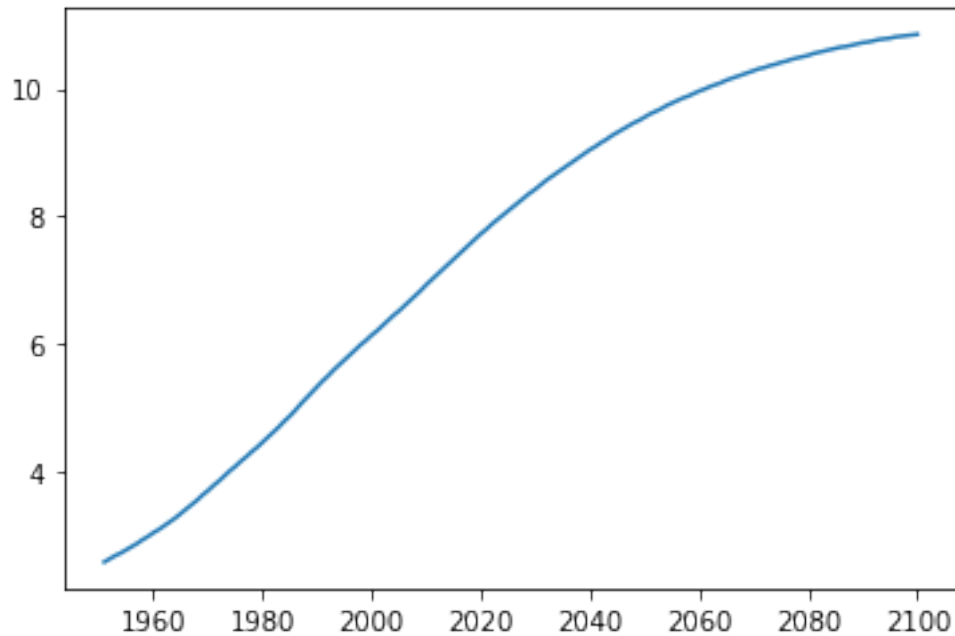
# Import matplotlib.pyplot as plt
import matplotlib.pyplot as plt

# 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()
```

2100.0

10.85



### 1.0.3 1.2 Line Plot (2): Interpretation

Have another look at the plot you created in the previous exercise; it's shown on the right. Based on the plot, in approximately what year will there be more than ten billion human beings on this planet?

#### Instructions

Possible Answers - 2040 - **2060** - 2085 - 2095

### 1.0.4 1.3 Line plot (3)

Now that you've built your first line plot, let's start working on the data that professor Hans Rosling used to build his beautiful bubble chart. It was collected in 2007. Two lists are available for you:

- `life_exp` which contains the life expectancy for each country and
- `gdp_cap`, which contains the GDP per capita (i.e. per person) for each country expressed in US Dollars.

GDP stands for Gross Domestic Product. It basically represents the size of the economy of a country. Divide this by the population and you get the GDP per capita.

`matplotlib.pyplot` is already imported as `plt`, so you can get started straight away.

#### Instructions

- Print the last item from both the list `gdp_cap`, and the list `life_exp`; it is information about Zimbabwe.
- Build a line chart, with `gdp_cap` on the x-axis, and `life_exp` on the y-axis. Does it make sense to plot this data on a line plot?

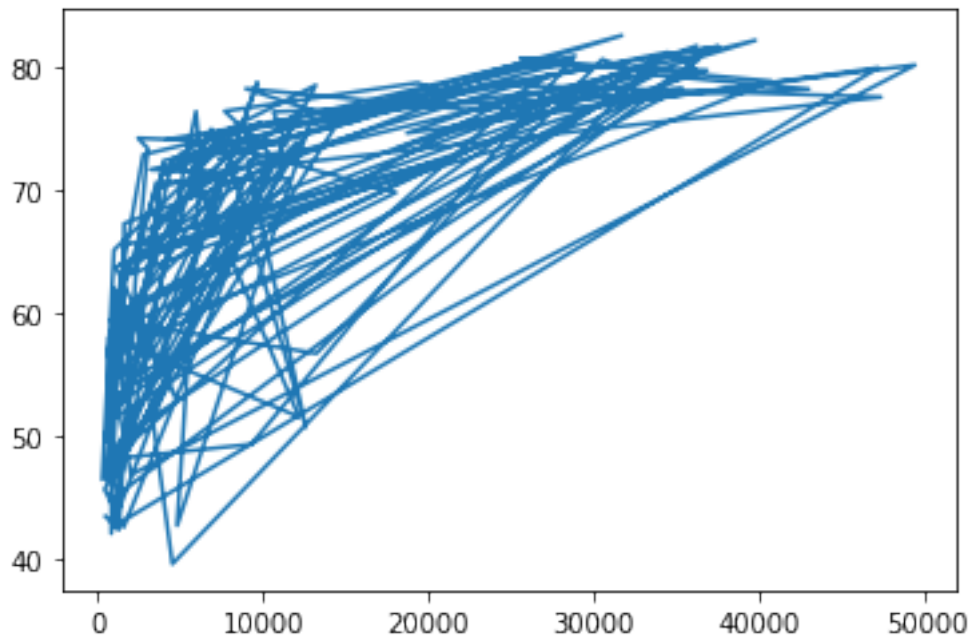
- Don't forget to finish off with a `plt.show()` command, to actually display the plot.

```
[3]: import pandas as pd
gapminder2007 = pd.read_csv('./Data/gapminder2007.csv', index_col=0)
gdp_cap = gapminder2007.gdp_cap.to_list()
life_exp = gapminder2007.life_exp.to_list()
pop = (gapminder2007.population / 1000000).to_list()
# Print the last item of gdp_cap and life_exp
print(gdp_cap[-1])
print(life_exp[-1])

# Make a line plot, gdp_cap on the x-axis, life_exp on the y-axis
plt.plot(gdp_cap, life_exp)

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

469.70929810000007  
43.486999999999995



*Well done, but this doesn't look right. Let's build a plot that makes more sense.*

### 1.0.5 1.4 Scatter Plot (1)

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. Below is an example of how to build a scatter plot. ~~~

`import matplotlib.pyplot as plt` `plt.scatter(x,y)` `plt.show()` ~~~ Let's continue with the `gdp_cap` versus `life_exp` plot, the GDP and life expectancy data for different countries in 2007. Maybe a scatter plot will be a better alternative?

Again, the `matplotlib.pyplot` package is available as `plt`.

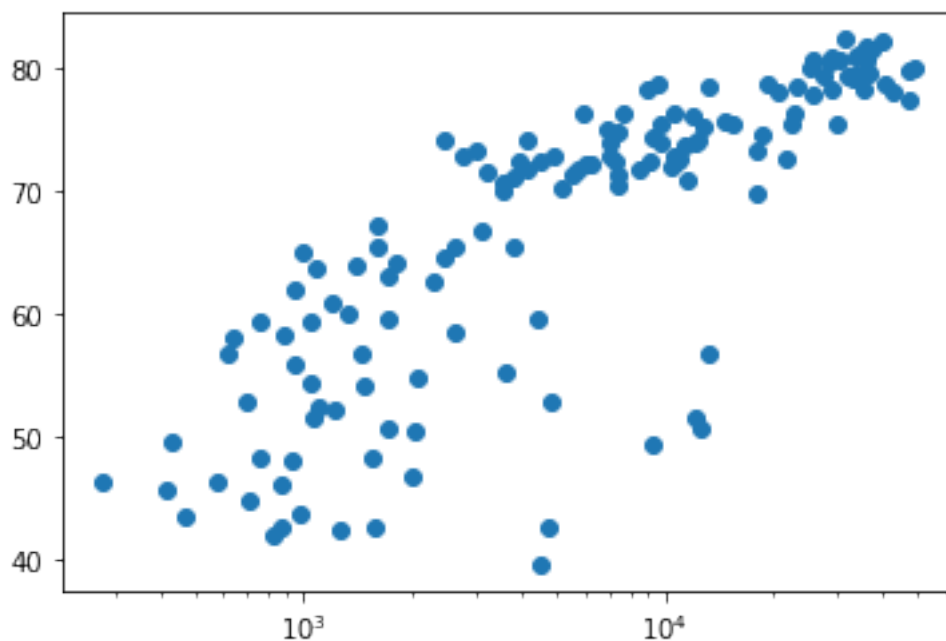
### Instructions

- Change the line plot that's coded in the script to a scatter plot.
- A correlation will become clear when you display the GDP per capita on a logarithmic scale. Add the line `plt.xscale('log')`.
- Finish off your script with `plt.show()` to display the plot.

```
[4]: # Change the line plot below to a scatter plot
plt.scatter(gdp_cap, life_exp)

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

# Show plot
plt.show()
```



### 1.0.6 1.5 Scatter plot (2)

In the previous exercise, you saw that the higher GDP usually corresponds to a higher life expectancy. In other words, there is a positive correlation.

Do you think there's a relationship between population and life expectancy of a country? The list

`life_exp` from the previous exercise is already available. In addition, now also `pop` is available, listing the corresponding populations for the countries in 2007. The populations are in millions of people.

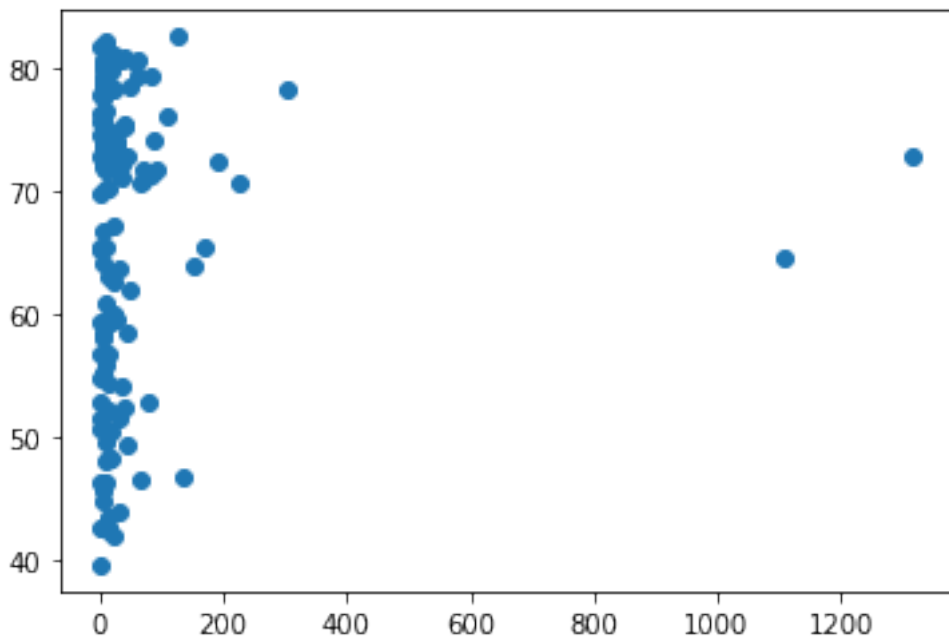
### Instructions

- Start from scratch: `import matplotlib.pyplot as plt`.
- Build a scatter plot, where `pop` is mapped on the horizontal axis, and `life_exp` is mapped on the vertical axis.
- Finish the script with `plt.show()` to actually display the plot. Do you see a correlation?

```
[5]: # Import package
import matplotlib.pyplot as plt

# Build Scatter plot
plt.scatter(pop, life_exp)

# Show plot
plt.show()
```



*Nice! There's no clear relationship between population and life expectancy, which makes perfect sense.*

### 1.0.7 1.6 Build a histogram (1)

`life_exp`, the list containing data on the life expectancy for different countries in 2007, is available in your Python shell.

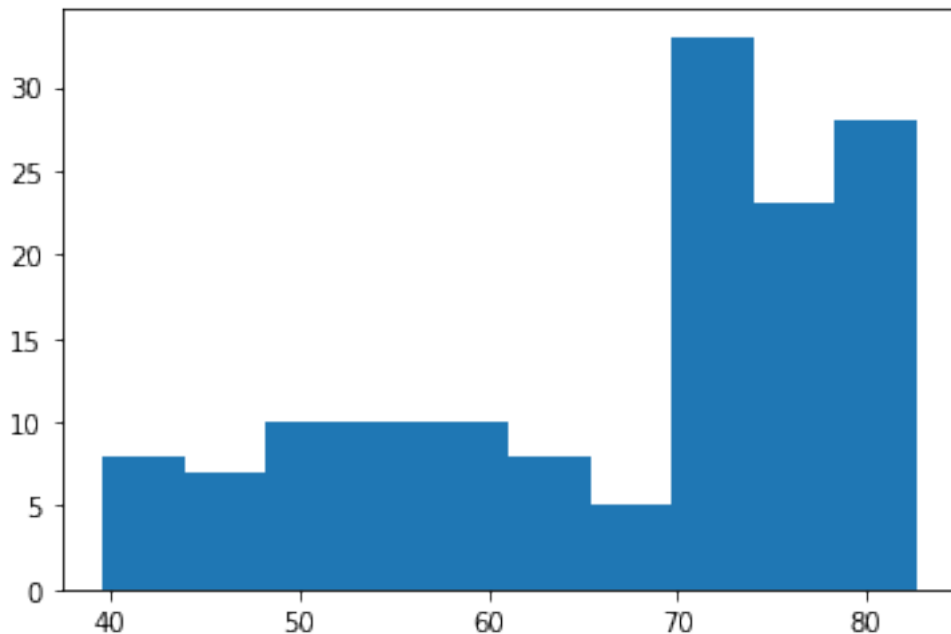
To see how life expectancy in different countries is distributed, let's create a histogram of `life_exp`. `matplotlib.pyplot` is already available as `plt`.

### Instructions

- Use `plt.hist()` to create a histogram of the values in `life_exp`. Do not specify the number of bins; Python will set the number of bins to 10 by default for you.
- Add `plt.show()` to actually display the histogram. Can you tell which bin contains the most observations?

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

# Display histogram
plt.show()
```



### 1.0.8 1.7 Build a histogram (2): bins

In the previous exercise, you didn't specify the number of bins. By default, Python sets the number of bins to 10 in that case. 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.

That's exactly what you'll do in this exercise. You'll be making two plots here. The code in the script already includes `plt.show()` and `plt.clf()` calls; `plt.show()` displays a plot; `plt.clf()` cleans it up again so you can start afresh.

As before, `life_exp` is available and `matplotlib.pyplot` is imported as `plt`.

### Instructions

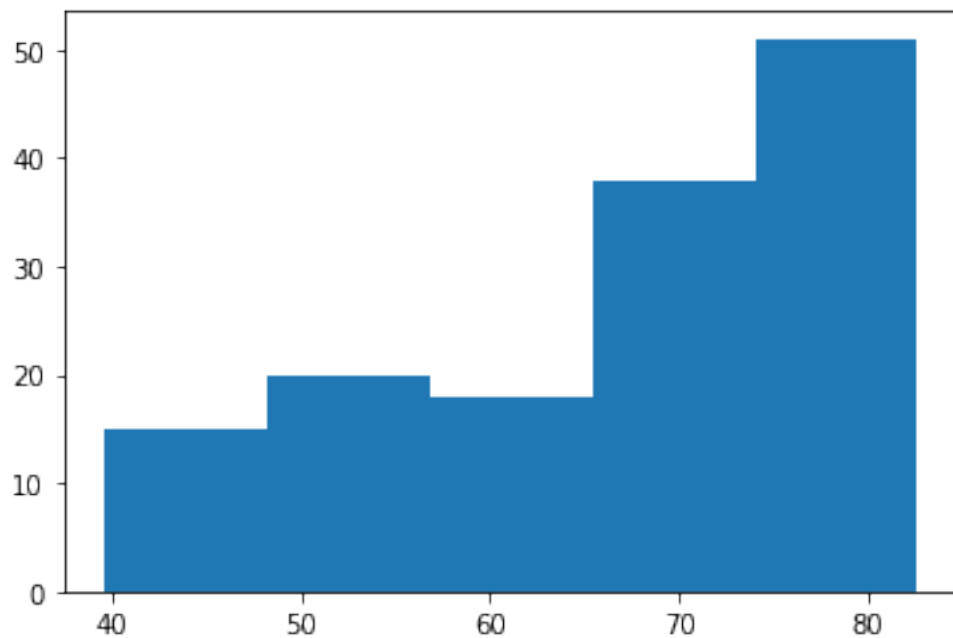
- Build a histogram of `life_exp`, with 5 bins. Can you tell which bin contains the most observations?
- Build another histogram of `life_exp`, this time with 20 bins. Is this better?

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

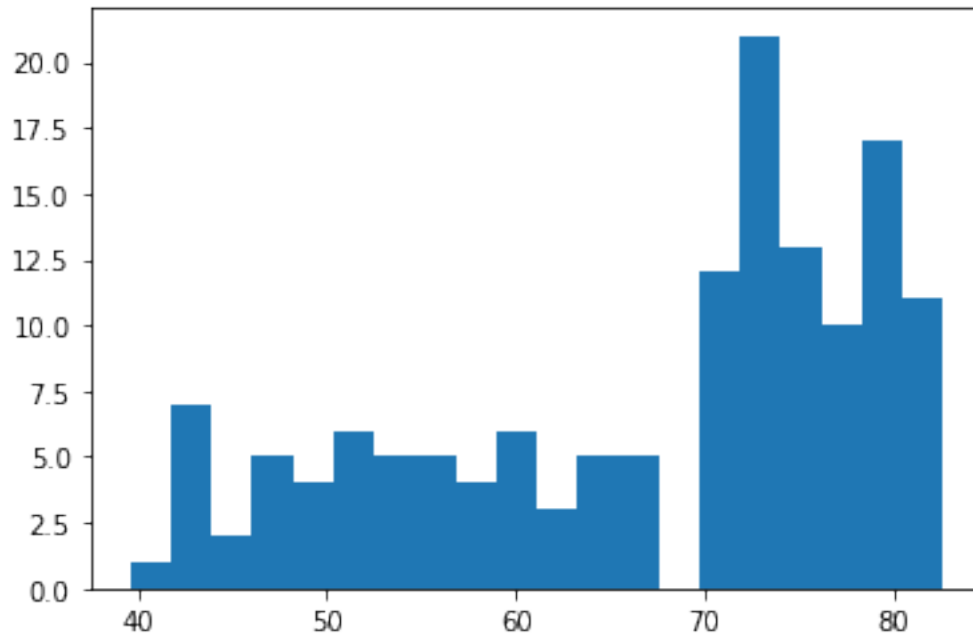
# Show and clean up plot
plt.show()
plt.clf()

# 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>

### 1.0.9 1.8 Build a histogram (3): compare

In the video, you saw population pyramids for the present day and for the future. Because we were using a histogram, it was very easy to make a comparison.

Let's do a similar comparison. `life_exp` contains life expectancy data for different countries in 2007. You also have access to a second list now, `life_exp1950`, containing similar data for 1950. Can you make a histogram for both datasets?

You'll again be making two plots. The `plt.show()` and `plt.clf()` commands to render everything nicely are already included. Also `matplotlib.pyplot` is imported for you, as `plt`.

#### Instructions

- Build a histogram of `life_exp` with 15 bins.
- Build a histogram of `life_exp1950`, also with 15 bins. Is there a big difference with the histogram for the 2007 data?

```
[8]: life_exp1950 = np.genfromtxt('./Data/life_exp1950.txt')
     life_exp1950 = life_exp1950.tolist()
```

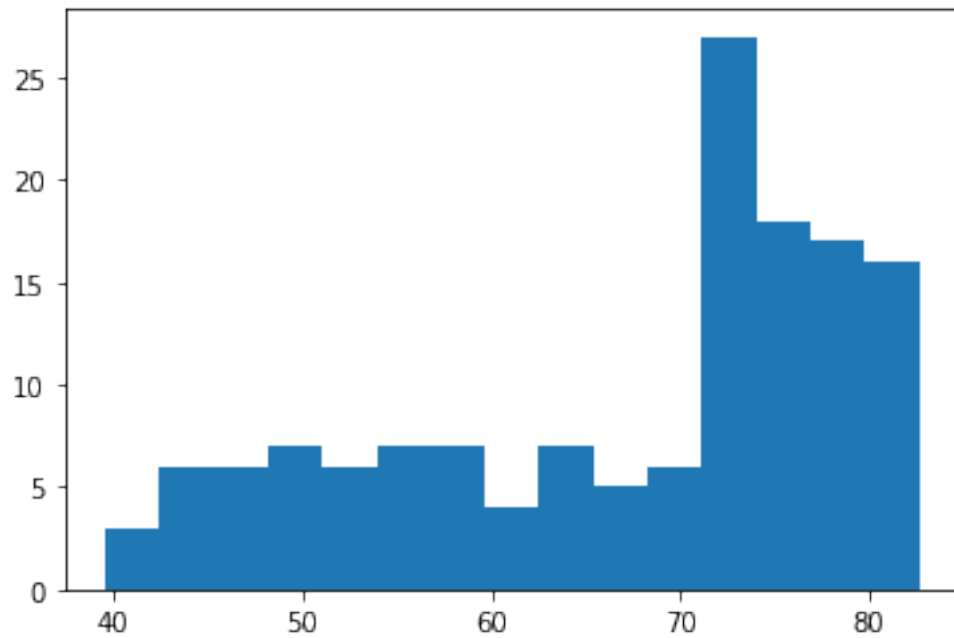
```
[9]: # Histogram of life_exp, 15 bins
     plt.hist(life_exp, bins=15)

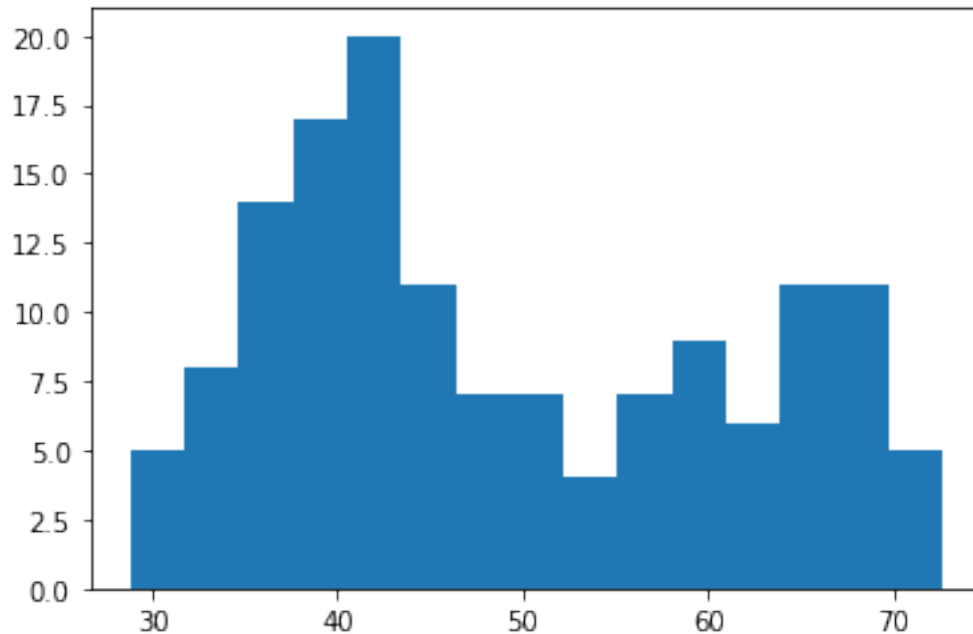
     # Show and clear plot
```

```
plt.show()
plt.clf()

# Histogram of life_exp1950, 15 bins
plt.hist(life_exp1950, bins=15)

# Show and clear plot again
plt.show()
plt.clf()
```





<Figure size 432x288 with 0 Axes>

*do you notice anything interesting?*

#### 1.0.10 1.9 Choose the right plot (1)

You're a professor teaching Data Science with Python, and you want to visually assess if the grades on your exam follow a particular distribution. Which plot do you use?

**Instructions**

**Possible Answers** - Line plot - Scatter plot - **Histogram**

#### 1.0.11 1.10 Choose the right plot (2)

You're a professor in Data Analytics with Python, and you want to visually assess if longer answers on exam questions lead to higher grades. Which plot do you use?

**Instructions**

**Possible Answers** - Line plot - **Scatter plot** - Histogram

#### 1.0.12 1.11 Labels

It's time to customize your own plot. This is the fun part, you will see your plot come to life!

You're going to work on the scatter plot with world development data: GDP per capita on the x-axis (logarithmic scale), life expectancy on the y-axis. The code for this plot is available in the script.

As a first step, let's add axis labels and a title to the plot. You can do this with the `xlabel()`, `ylabel()` and `title()` functions, available in `matplotlib.pyplot`. This sub-package is already imported as `plt`.

### Instructions

- The strings `xlab` and `ylab` are already set for you. Use these variables to set the label of the x- and y-axis.
- The string `title` is also coded for you. Use it to add a title to the plot.
- After these customizations, finish the script with `plt.show()` to actually display the plot.

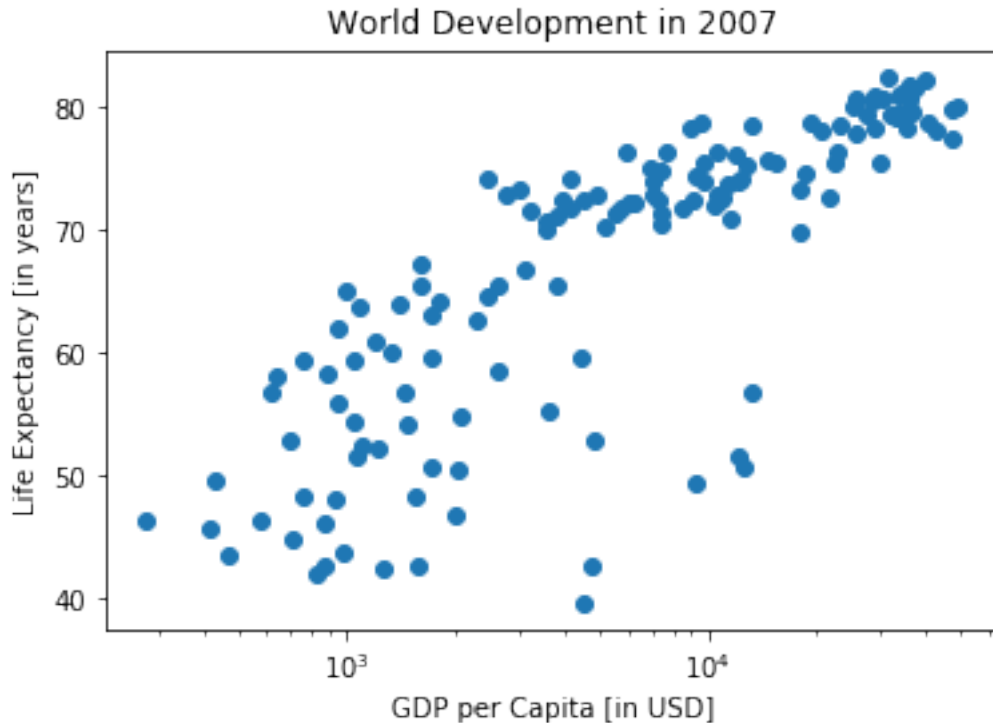
```
[10]: # Basic scatter plot, log scale
plt.scatter(gdp_cap, life_exp)
plt.xscale('log')

# 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()
```



### 1.0.13 1.12 Ticks

The customizations you’ve coded up to now are available in the script, in a more concise form.

In the video, Filip has demonstrated how you could control the y-ticks by specifying two arguments:

```
~~~ plt.yticks([0,1,2], ["one","two","three"]) ~~~
```

In this example, the ticks corresponding to the numbers 0, 1 and 2 will be replaced by one, two and three, respectively.

Let’s do a similar thing for the x-axis of your world development chart, with the `xticks()` function. The tick values 1000, 10000 and 100000 should be replaced by 1k, 10k and 100k. To this end, two lists have already been created for you: `tick_val` and `tick_lab`.

#### Instructions

- Use `tick_val` and `tick_lab` as inputs to the `xticks()` function to make the the plot more readable.
- As usual, display the plot with `plt.show()` after you’ve added the customizations.

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

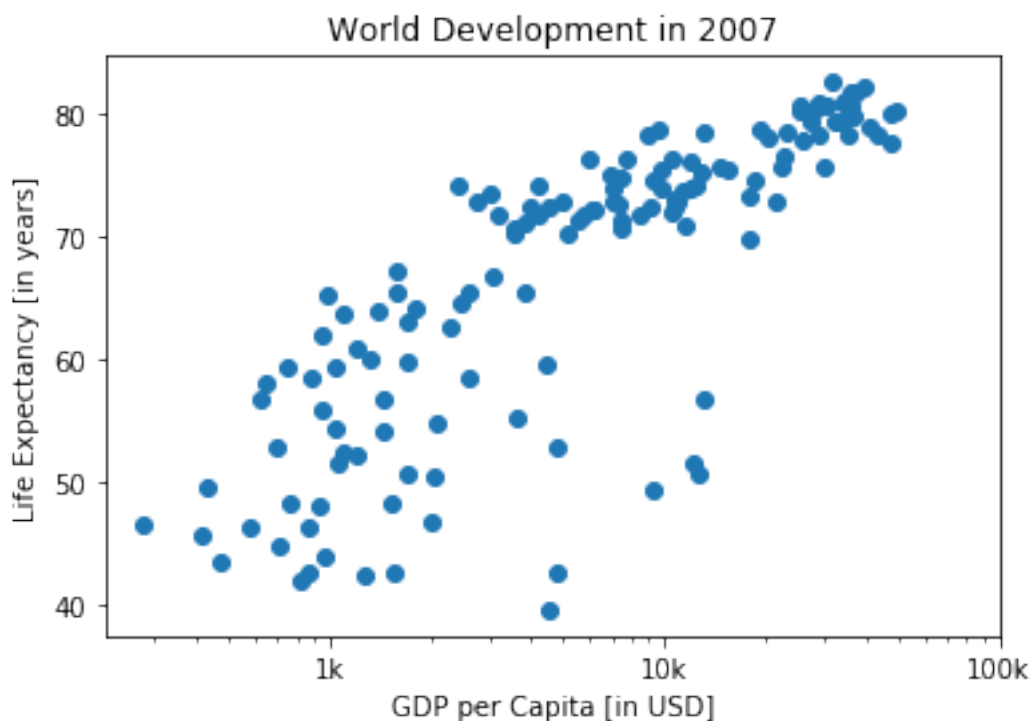
Previous customizations
plt.xscale('log')
plt.xlabel('GDP per Capita [in USD]')
```

```
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()
```



#### 1.0.14 1.13 Sizes

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.

#### Instructions

- Run the script to see how the plot changes.

- Looks good, but increasing the size of the bubbles will make things stand out more.
  - Import the `numpy` package as `np`.
  - Use `np.array()` to create a numpy array from the list `pop`. Call this Numpy array `np_pop`.
  - Double the values in `np_pop` setting the value of `np_pop` equal to `np_pop * 2`. Because `np_pop` is a Numpy array, each array element will be doubled.
  - Change the `s` argument inside `plt.scatter()` to be `np_pop` instead of `pop`.

```
[12]: # Import numpy as np
import numpy as np

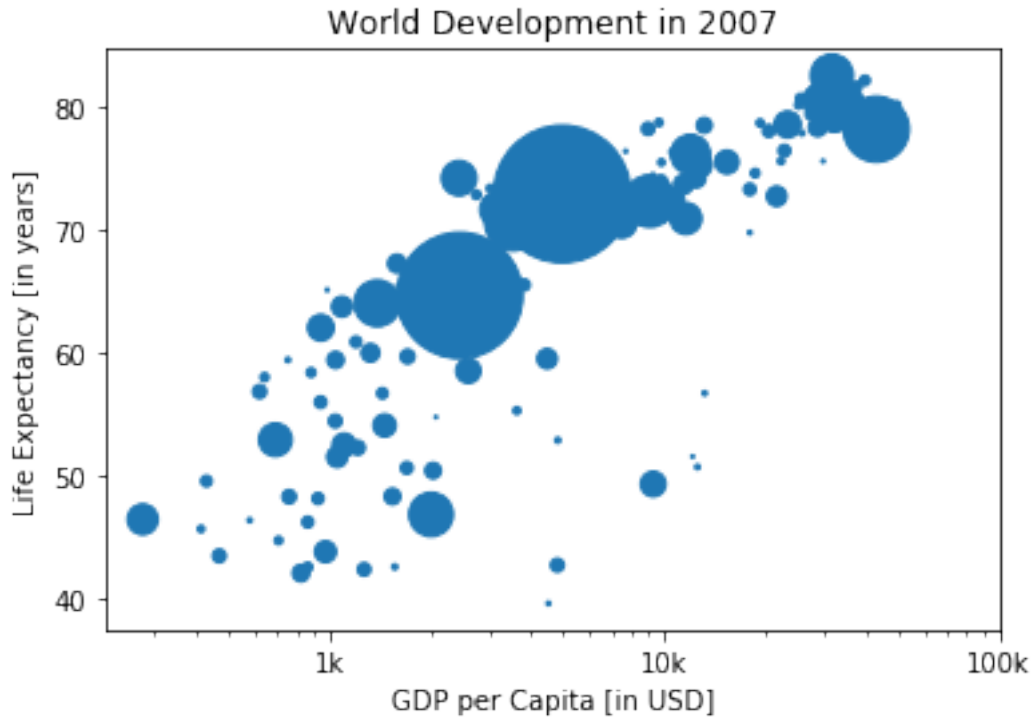
Store pop as a numpy array: np_pop
np_pop = np.array(pop)

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()
```



### 1.0.15 1.14 Colors

The code you've written up to now is available in the script on the right.

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' }
~~~
```

Nothing to worry about now; you will learn about dictionaries in the next chapter.

#### Instructions

- Add `c = col` to the arguments of the `plt.scatter()` function.
- Change the opacity of the bubbles by setting the `alpha` argument to 0.8 inside `plt.scatter()`. Alpha can be set from zero to one, where zero is totally transparent, and one is not at all transparent.

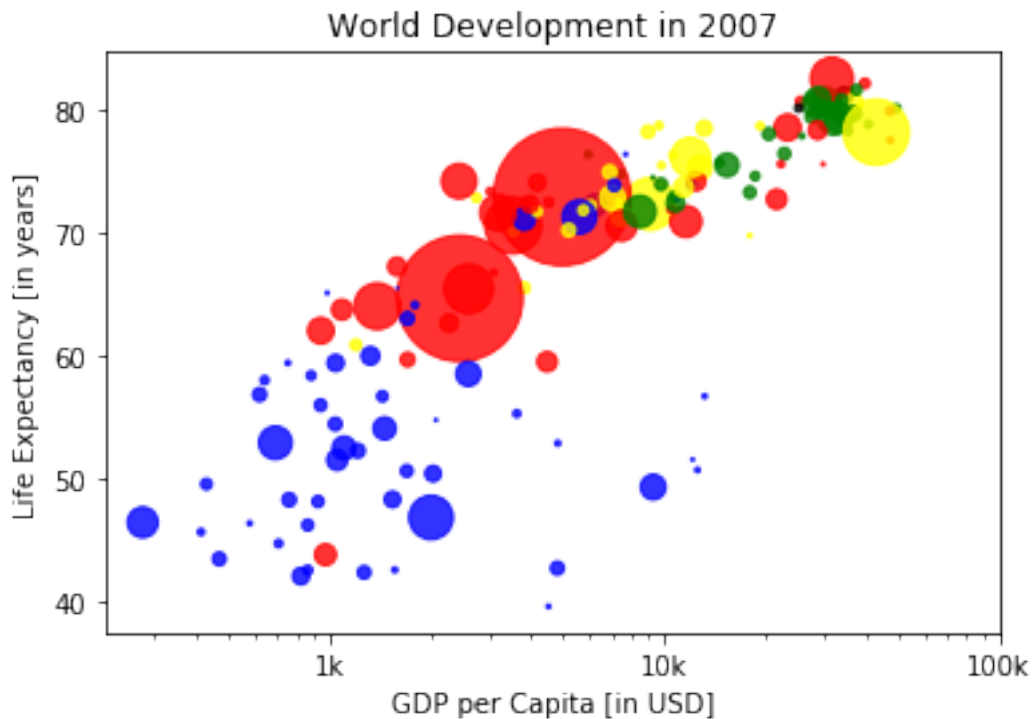
```
[13]: col = pd.read_csv('./Data/col.txt', header=None, quotechar="'')[0].to_list()
```

```
[14]: # 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()
```



### 1.0.16 1.15 Additional Customizations

If you have another look at the script, under `# Additional Customizations`, you'll see that there are two `plt.text()` functions now. They add the words "India" and "China" in the plot.

#### Instructions

- Add `plt.grid(True)` after the `plt.text()` calls so that gridlines are drawn on the plot.

```
[15]: # Scatter plot
plt.scatter(x = gdp_cap, y = life_exp, s = np.array(pop) * 2, c = col, alpha = 0.8)

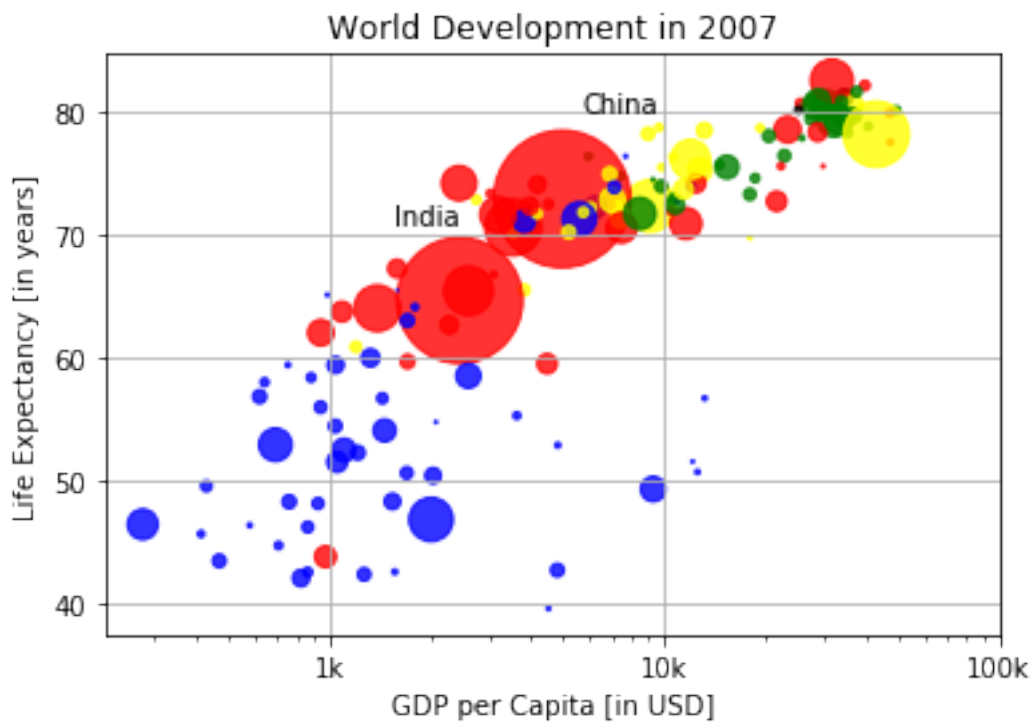
Previous customizations
```

```
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'])

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

Add grid() call
plt.grid(True)

Show the plot
plt.show()
```



*Beautiful! A visualization only makes sense if you can interpret it properly. Let's do that in the next exercise.*

### 1.0.17 1.16 Interpretation

If you have a look at your colorful plot, it's clear that people live longer in countries with a higher GDP per capita. No high income countries have really short life expectancy, and no low income countries have very long life expectancy. Still, there is a huge difference in life expectancy between

countries on the same income level. Most people live in middle income countries where difference in lifespan is huge between countries; depending on how income is distributed and how it is used.

What can you say about the plot?

### Instructions

**Possible Answers - The countries in blue, corresponding to Africa, have both low life expectancy and a low GDP per capita.** - There is a negative correlation between GDP per capita and life expectancy. - China has both a lower GDP per capita and lower life expectancy compared to India.

## 1.0.18 2. Dictionaries & Pandas

Learn about the dictionary, an alternative to the Python list, and the pandas DataFrame, the de facto standard to work with tabular data in Python. You will get hands-on practice with creating and manipulating datasets, and you'll learn how to access the information you need from these data structures.

### 1.0.19 2.1 Motivation for dictionaries

To see why dictionaries are useful, have a look at the two lists defined on the right. `countries` contains the names of some European countries. `capitals` lists the corresponding names of their capital.

### Instructions

- Use the `index()` method on `countries` to find the index of `'germany'`. Store this index as `ind_ger`.
- Use `ind_ger` to access the capital of Germany from the `capitals` list. Print it out.

```
[16]: # 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

### 1.0.20 2.2 Create dictionary

The `countries` and `capitals` lists are again available in the script. It's your job to convert this data to a dictionary where the country names are the keys and the capitals are the corresponding values. As a refresher, here is a recipe for creating a dictionary: `~~~ my_dict = { "key1": "value1", "key2": "value2", } ~~~` In this recipe, both the keys and the values are strings. This will also be the case for this exercise.

### Instructions

- With the strings in `countries` and `capitals`, create a dictionary called `europe` with 4 key:value pairs. Beware of capitalization! Make sure you use lowercase characters everywhere.
- Print out `europe` to see if the result is what you expected.

[ ]:

### 1.0.21 2.3 Access dictionary

If the keys of a dictionary are chosen wisely, accessing the values in a dictionary is easy and intuitive. For example, to get the capital for France from `europe` you can use: `~~ europe['france'] ~~`. Here, `'france'` is the key and `'paris'` the value is returned. **Instructions**

- Check out which keys are in `europe` by calling the `keys()` method on `europe`. Print out the result.
- Print out the value that belongs to the key `'norway'`.

```
[17]: # 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
```

### 1.0.22 2.4 Dictionary Manipulation (1)

If you know how to access a dictionary, you can also assign a new value to it. To add a new key-value pair to `europe` you can use something like this: `~~~ europe['iceland'] = 'reykjavik' ~~~`

#### Instructions

- Add the key `'italy'` with the value `'rome'` to `europe`.
- To assert that `'italy'` is now a key in `europe`, print out `'italy'` in `europe`.
- Add another key:value pair to `europe`: `'poland'` is the key, `'warsaw'` is the corresponding value.
- Print out `europe`.

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

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

```
Print out italy in europe
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', 'poland': 'warsaw'}
```

*Well done! Europe is growing by the minute! Did you notice that the order of the printout is not the same as the order in the dictionary's definition? That's because dictionaries are inherently unordered.*

### 1.0.23 2.5 Dictionary Manipulation (2)

Somebody thought it would be funny to mess with your accurately generated dictionary. An adapted version of the `europe` dictionary is available in the script on the right.

Can you clean up? Do not do this by adapting the definition of `europe`, but by adding Python commands to the script to update and remove key:value pairs.

#### Instructions

- The capital of Germany is not 'bonn'; it's 'berlin'. Update its value.
- Australia is not in Europe, Austria is! Remove the key 'australia' from `europe`.
- Print out `europe` to see if your cleaning work paid off.

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

Update capital of germany
europe['germany'] = 'berlin'

Remove australia
del(europe['australia'])

Print europe
print(europe)
```

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

### 1.0.24 2.6 Dictionaryception

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.

As an example, have a look at the script where another version of `europe` - the dictionary you've been working with all along - is coded. The keys are still the country names, but the values are dictionaries that contain more information than just the capital.

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'] ~~`

#### Instructions

- Use chained square brackets to select and print out the capital of France.
- Create a dictionary, named `data`, with the keys `'capital'` and `'population'`. Set them to `'rome'` and `59.83`, respectively.
- Add a new key-value pair to `europe`; the key is `'italy'` and the value is `data`, the dictionary you just built.

```
[20]: # 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', 'population': 66.03}, 'germany': {'capital': 'berlin', 'population':
80.62}, 'norway': {'capital': 'oslo', 'population': 5.084}, 'italy': {'capital':
'rome', 'population': 59.83}}
```

### 1.0.25 2.7 Dictionary to DataFrame (1)

Pandas is an open source library, providing high-performance, easy-to-use data structures and data analysis tools for Python. Sounds promising!

The DataFrame is one of Pandas' most important data structures. 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.

In the exercises that follow you will be working with vehicle data from different countries. Each observation corresponds to a country and the columns give information about the number of vehicles per capita, whether people drive left or right, and so on.

Three lists are defined in the script:

- **names**, containing the country names for which data is available.
- **dr**, a list with booleans that tells whether people drive left or right in the corresponding country.
- **cpc**, the number of motor vehicles per 1000 people in the corresponding country.

Each dictionary key is a column label and each value is a list which contains the column elements.

#### Instructions

- Import **pandas** as **pd**.
- Use the pre-defined lists to create a dictionary called **my\_dict**. There should be three key value pairs:
  - key 'country' and value **names**.
  - key 'drives\_right' and value **dr**.
  - key 'cars\_per\_cap' and value **cpc**.
- Use **pd.DataFrame()** to turn your dict into a DataFrame called **cars**.
- Print out **cars** and see how beautiful it is.

```
[21]: # 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           |

Good job! Notice that the columns of cars can be of different types. This was not possible with 2D Numpy arrays!

### 1.0.26 2.8 Dictionary to DataFrame (2)

The Python code that solves the previous exercise is included on the right. 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`.

#### Instructions

- Hit *Run Code* to see that, indeed, the row labels are not correctly set.
- Specify the row labels by setting `cars.index` equal to `row_labels`.
- Print out `cars` again and check if the row labels are correct this time.

```
[22]: import pandas as pd

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           |

### 1.0.27 2.9 CSV to DataFrame (1)

Putting data in a dictionary and then building a DataFrame works, but it's not very efficient. What if you're dealing with millions of observations? In those cases, the data is typically available as files with a regular structure. One of those file types is the CSV file, which is short for "comma-separated values".

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

Let's explore this function with the same cars data from the previous exercises. This time, however, the data is available in a CSV file, named `cars.csv`. It is available in your current working directory, so the path to the file is simply `'cars.csv'`.

#### Instructions

- To import CSV files you still need the `pandas` package: import it as `pd`.
- Use `pd.read_csv()` to import `cars.csv` data as a DataFrame. Store this dataframe as `cars`.
- Print out `cars`. Does everything look OK?

```
[23]: # Import pandas as pd
import pandas as pd

Import the cars.csv data: cars
cars = pd.read_csv('./Data/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  |

*Nice job! Looks nice, but not exactly what we expected. Let's fix this in the next exercise.*

### 1.0.28 2.10 CSV to DataFrame (2)

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!

Python code that solves the previous exercise is already included; can you make the appropriate changes to fix the data import?

#### Instructions

- Run the code with Submit Answer and assert that the first column should actually be used as row labels.
- Specify the `index_col` argument inside `pd.read_csv()`: set it to 0, so that the first column is used as row labels.
- Has the printout of `cars` improved now?

```
[24]: # Import pandas as pd
import pandas as pd

Fix import by including index_col
cars = pd.read_csv('./Data/cars.csv', index_col = 0)

Print out cars
print(cars)
```

|     | <code>cars_per_cap</code> | <code>country</code> | <code>drives_right</code> |
|-----|---------------------------|----------------------|---------------------------|
| 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                      |

### 1.0.29 2.11 Square Brackets (1)

In the video, you saw that you can index and select Pandas DataFrames in many different ways. The simplest, but not the most powerful way, is to use square brackets.

In the sample code on the right, the same cars data is imported from a CSV files as a Pandas DataFrame. To select only the `cars_per_cap` column from `cars`, you can use:

~~ cars['cars\_per\_cap'] cars[['cars\_per\_cap']] ~~ The single bracket version gives a Pandas Series, the double bracket version gives a Pandas DataFrame.

## Instructions

- Use single square brackets to print out the `country` column of `cars` as a Pandas Series.
- Use double square brackets to print out the `country` column of `cars` as a Pandas DataFrame.
- Use double square brackets to print out a DataFrame with both the `country` and `drives_right` columns of `cars`, in this order.

```
[25]: # Import cars data
import pandas as pd
cars = pd.read_csv('./Data/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
```

### 1.0.30 2.12 Square Brackets (2)

Square brackets can do more than just selecting columns. You can also use them to get rows, or observations, from a DataFrame. The following call selects the first five rows from the `cars` DataFrame: `cars[0:5]` The result is another DataFrame containing only the rows you specified.

Pay attention: You can only select rows using square brackets if you specify a slice, like `0:4`. Also, you're using the integer indexes of the rows here, not the row labels!

#### Instructions

- Select the first 3 observations from `cars` and print them out.
- Select the fourth, fifth and sixth observation, corresponding to row indexes 3, 4 and 5, and print them out.

```
[26]: # Import cars data
import pandas as pd
cars = pd.read_csv('./Data/cars.csv', index_col = 0)

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         |

*Good job. You can get interesting information, but using square brackets to do indexing is rather limited. Experiment with more advanced techniques in the following exercises.*

### 1.0.31 2.13 loc and iloc (1)

With `loc` and `iloc` you can do practically any data selection operation on DataFrames you can think of. `loc` is label-based, which means that you have to specify rows and columns based on their row and column labels. `iloc` is integer index based, so you have to specify rows and columns by their integer index like you did in the previous exercise.

Try out the following commands in the IPython Shell to experiment with `loc` and `iloc` to select observations. 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]]` ~~~ As before, code is included that imports the cars data as a Pandas DataFrame.

#### Instructions

- Use `loc` or `iloc` to select the observation corresponding to Japan as a Series. The label of this row is JPN, the index is 2. Make sure to print the resulting Series.
- Use `loc` or `iloc` to select the observations for Australia and Egypt as a DataFrame. You can find out about the labels/indexes of these rows by inspecting `cars` in the IPython Shell. Make sure to print the resulting DataFrame.

```
[27]: # Import cars data
import pandas as pd
cars = pd.read_csv('./Data/cars.csv', index_col = 0)

Print out observation for Japan
print(cars.iloc[2]) # as Series

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

```
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
```

### 1.0.32 2.14 loc and iloc (2)

`loc` and `iloc` also allow you to select both rows and columns from a DataFrame. To experiment, try out the following commands in the IPython Shell. Again, 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]]` ~~~ **Instructions**

- Print out the `drives_right` value of the row corresponding to Morocco (its row label is MOR)
- Print out a sub-DataFrame, containing the observations for Russia and Morocco and the columns `country` and `drives_right`.

```
[28]: # Import cars data
import pandas as pd
cars = pd.read_csv('./data/cars.csv', index_col = 0)

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 |

Great! You might wonder if you can also combine label-based selection the `loc` way and index-based selection the `iloc` way. You can! It's done with `ix`, but we won't go into that here.

### 1.0.33 2.15 loc and iloc (3)

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]]` ~~~ **Instructions**

- Print out the `drives_right` column as a Series using `loc` or `iloc`.
- Print out the `drives_right` column as a DataFrame using `loc` or `iloc`.
- Print out both the `cars_per_cap` and `drives_right` column as a DataFrame using `loc` or `iloc`.

```
[29]: # Import cars data
import pandas as pd
cars = pd.read_csv('./data/cars.csv', index_col = 0)

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  |

### 1.1 3. Logic, Control Flow and Filtering

Boolean logic is the foundation of decision-making in Python programs. Learn about different comparison operators, how to combine them with Boolean operators, and how to use the Boolean outcomes in control structures. You'll also learn to filter data in pandas DataFrames using logic.

#### 1.1.1 3.1 Equality

To check if two Python values, or variables, are equal you can use `==`. To check for inequality, you need `!=`. As a refresher, have a look at the following examples that all result in `True`. Feel free to try them out in the IPython Shell. `2 == (1 + 1)` “intermediate” `!=` “python” `True != False` “Python” `!=` “python”

When you write these comparisons in a script, you will need to wrap a `print()` function around them to see the output.

#### Instructions

- In the editor on the right, write code to see if `True` equals `False`.
- Write Python code to check if `-5 * 15` is not equal to 75.
- Ask Python whether the strings `"pyscript"` and `"PyScript"` are equal.
- What happens if you compare booleans and integers? Write code to see if `True` and 1 are equal.

```
[30]: # 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(True == 1)
```

```
False
True
False
True
```

*The last comparison worked fine because actually, a boolean is a special kind of integer: `True` corresponds to 1, `False` corresponds to 0.*

### 1.1.2 3.2 Greater and less than

In the video, Filip also talked about the less than and greater than signs, `<` and `>` in Python. You can combine them with an equals sign: `<=` and `>=`. Pay attention: `<=` is valid syntax, but `=<` is not.

All Python expressions in the following code chunk evaluate to `True`: `3 < 4` `3 <= 4` `"alpha" <= "beta"`

Remember that for string comparison, Python determines the relationship based on alphabetical order.

#### Instructions

- Write Python expressions, wrapped in a `print()` function, to check whether:
  - `x` is greater than or equal to `-10`. `x` has already been defined for you.
  - `"test"` is less than or equal to `y`. `y` has already been defined for you.
  - `True` is greater than `False`.

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

Comparison of strings
y = "test"
print("test" <= y)

Comparison of booleans
print(True > False)
```

False

True

True

### 1.1.3 3.3 Compare arrays

Out of the box, you can also use comparison operators with Numpy arrays.

Remember `areas`, the list of area measurements for different rooms in your house from Introduction to Python? This time there's two Numpy arrays: `my_house` and `your_house`. They both contain the areas for the kitchen, living room, bedroom and bathroom in the same order, so you can compare them.

#### Instructions

Using comparison operators, generate boolean arrays that answer the following questions:

- Which areas in `my_house` are greater than or equal to 18?
- You can also compare two Numpy arrays element-wise. Which areas in `my_house` are smaller than the ones in `your_house`?
- Make sure to wrap both commands in a `print()` statement so that you can inspect the output!



```
[32]: # 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]
```

Good job. It appears that the living room and bedroom in `my_house` are smaller than the corresponding areas in `your_house`.

#### 1.1.4 3.4 and, or, not (1)

A boolean is either 1 or 0, `True` or `False`. With boolean operators such as `and`, `or` and `not`, you can combine these booleans to perform more advanced queries on your data.

In the sample code on the right, two variables are defined: `my_kitchen` and `your_kitchen`, representing areas.

#### Instructions

- Write Python expressions, wrapped in a `print()` function, to check whether:
  - `my_kitchen` is bigger than 10 and smaller than 18.
  - `my_kitchen` is smaller than 14 or bigger than 17.
  - double the area of `my_kitchen` is smaller than triple the area of `your_kitchen`.

```
[33]: # 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
```

### 1.1.5 3.5 and, or, not (2)

To see if you completely understood the boolean operators, have a look at the following piece of Python code: `~~~ x = 8 y = 9 not(not(x < 3) and not(y > 14 or y > 10)) ~~~` What will the result be if you execute these three commands in the IPython Shell?

NB: Notice that `not` has a higher priority than `and` and `or`, it is executed first.

#### Instructions

**Possible Answers** - True - False - Running these commands will result in an error.

*Correct! `x < 3` is False. `y > 14` or `y > 10` is False as well. If you continue working like this, simplifying from inside outwards, you'll end up with False.*

### 1.1.6 3.6 Boolean operators with Numpy

Before, the operational operators like `<` and `>=` worked with Numpy arrays out of the box. Unfortunately, this is not true for the boolean operators `and`, `or`, and `not`.

To use these operators with Numpy, you will need `np.logical_and()`, `np.logical_or()` and `np.logical_not()`. Here's an example on the `my_house` and `your_house` arrays from before to give you an idea: `~~~ np.logical_and(my_house > 13, your_house < 15) ~~~`

#### Instructions

- Generate boolean arrays that answer the following questions:
- Which areas in `my_house` are greater than 18.5 or smaller than 10?
- Which areas are smaller than 11 in both `my_house` and `your_house`? Make sure to wrap both commands in `print()` statement, so that you can inspect the output.

```
[34]: # 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 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]
```

### 1.1.7 3.7 Warmup

To experiment with `if` and `else` a bit, have a look at this code sample: `~~~ area = 10.0 if(area < 9) : print("small") elif(area < 12) : print("medium") else : print("large") ~~~` What will the output be if you run this piece of code in the IPython Shell?

#### Instructions

Possible Answers - small - **medium** - large - The syntax is incorrect; this code will produce an error.

### 1.1.8 3.8 if

It's time to take a closer look around in your house.

Two variables are defined in the sample code: `room`, a string that tells you which room of the house we're looking at, and `area`, the `area` of that room.

#### Instructions

- Examine the `if` statement that prints out "Looking around in the kitchen." if `room` equals "kit".
- Write another `if` statement that prints out "big place!" if `area` is greater than 15.

```
[35]: # Define variables
room = "kit"
area = 14.0

if statement for room
if room == "kit" :
 print("looking around in the kitchen.")

if statement for area
if area > 15 :
 print("big place!")
```

looking around in the kitchen.

*Great! big place! wasn't printed, because `area > 15` is not `True`. Experiment with other values of `room` and `area` to see how the printouts change.*

### 1.1.9 3.9 Add else

On the right, the `if` construct for `room` has been extended with an `else` statement so that "looking around elsewhere." is printed if the condition `room == "kit"` evaluates to `False`.

Can you do a similar thing to add more functionality to the `if` construct for `area`?

#### Instructions

Add an `else` statement to the second control structure so that "pretty small." is printed out if `area > 15` evaluates to `False`.

```
[36]: # Define variables
room = "kit"
area = 14.0

if-else construct for room
if room == "kit" :
 print("looking around in the kitchen.")
```

```

else :
 print("looking around elsewhere.")

if-else construct for area
if area > 15 :
 print("big place!")
else :
 print("pretty small.")

```

looking around in the kitchen.  
pretty small.

### 1.1.10 3.10 Customize further: elif

It's also possible to have a look around in the bedroom. The sample code contains an `elif` part that checks if `room` equals "bed". In that case, "looking around in the bedroom." is printed out.

It's up to you now! Make a similar addition to the second control structure to further customize the messages for different values of `area`.

#### Instructions

Add an `elif` to the second control structure such that "medium size, nice!" is printed out if `area` is greater than 10.

```

[37]: # 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!

### 1.1.11 3.11 Driving right (1)

Remember that `cars` dataset, containing the cars per 1000 people (`cars_per_cap`) and whether people drive right (`drives_right`) for different countries (`country`)? The code that imports this data in CSV format into Python as a DataFrame is available on the right.

In the video, you saw a step-by-step approach to filter observations from a DataFrame based on boolean arrays. Let's start simple and try to find all observations in `cars` where `drives_right` is `True`.

`drives_right` is a boolean column, so you'll have to extract it as a Series and then use this boolean Series to select observations from `cars`.

#### Instructions

- Extract the `drives_right` column as a Pandas Series and store it as `dr`.
- Use `dr`, a boolean Series, to subset the `cars` DataFrame. Store the resulting selection in `sel`.
- Print `sel`, and assert that `drives_right` is `True` for all observations.

```
[38]: # Import cars data
import pandas as pd
cars = pd.read_csv('./data/cars.csv', index_col = 0)

Extract drives_right column as Series: dr
dr = cars["drives_right"]

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

Print sel
print(sel)
```

|     | <code>cars_per_cap</code> | <code>country</code> | <code>drives_right</code> |
|-----|---------------------------|----------------------|---------------------------|
| US  | 809                       | United States        | True                      |
| RU  | 200                       | Russia               | True                      |
| MOR | 70                        | Morocco              | True                      |
| EG  | 45                        | Egypt                | True                      |

### 1.1.12 3.12 Driving right (2)

The code in the previous example worked fine, but you actually unnecessarily created a new variable `dr`. You can achieve the same result without this intermediate variable. Put the code that computes `dr` straight into the square brackets that select observations from `cars`.

#### Instructions

Convert the code on the right to a one-liner that calculates the variable `sel` as before.

```
[39]: # Import cars data
import pandas as pd
cars = pd.read_csv('./data/cars.csv', index_col = 0)
```

```

Convert code to a one-liner
dr = cars['drives_right']
sel = cars[dr]

One-liner
sel = cars[cars['drives_right']]

Print sel
print(sel)

```

|     | <code>cars_per_cap</code> | <code>country</code> | <code>drives_right</code> |
|-----|---------------------------|----------------------|---------------------------|
| US  | 809                       | United States        | True                      |
| RU  | 200                       | Russia               | True                      |
| MOR | 70                        | Morocco              | True                      |
| EG  | 45                        | Egypt                | True                      |

### 1.1.13 3.13 Cars per capita (1)

Let's stick to the `cars` data some more. This time you want to find out which countries have a high cars per capita figure. In other words, in which countries do many people have a car, or maybe multiple cars.

Similar to the previous example, you'll want to build up a boolean Series, that you can then use to subset the `cars` DataFrame to select certain observations. If you want to do this in a one-liner, that's perfectly fine!

#### Instructions

- Select the `cars_per_cap` column from `cars` as a Pandas Series and store it as `cpc`.
- Use `cpc` in combination with a comparison operator and 500. You want to end up with a boolean Series that's `True` if the corresponding country has a `cars_per_cap` of more than 500 and `False` otherwise. Store this boolean Series as `many_cars`.
- Use `many_cars` to subset `cars`, similar to what you did before. Store the result as `car_maniac`.
- Print out `car_maniac` to see if you got it right.

```

[40]: # Import cars data
import pandas as pd
cars = pd.read_csv('./data/cars.csv', index_col = 0)

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)

```

|    | <code>cars_per_cap</code> | <code>country</code> | <code>drives_right</code> |
|----|---------------------------|----------------------|---------------------------|
| US | 809                       | United States        | True                      |

|     |     |           |       |
|-----|-----|-----------|-------|
| AUS | 731 | Australia | False |
| JAP | 588 | Japan     | False |

### 1.1.14 3.14 Cars per capita (2)

Remember about `np.logical_and()`, `np.logical_or()` and `np.logical_not()`, the Numpy variants of the `and`, `or` and `not` operators? You can also use them on Pandas Series to do more advanced filtering operations.

Take this example that selects the observations that have a `cars_per_cap` between 10 and 80. Try out these lines of code step by step to see what's happening. `~~~ cpc = cars['cars_per_cap'] between = np.logical_and(cpc > 10, cpc < 80) medium = cars[between] ~~~` **Instructions**

- Use the code sample above to create a DataFrame `medium`, that includes all the observations of `cars` that have a `cars_per_cap` between 100 and 500.
- Print out `medium`.

```
[41]: # Import cars data
import pandas as pd
cars = pd.read_csv('./data/cars.csv', index_col = 0)

Import numpy, you'll need this
import numpy as np

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         |

## 1.2 4. Loops

There are several techniques you can use to repeatedly execute Python code. While loops are like repeated if statements, the for loop iterates over all kinds of data structures. Learn all about them in this chapter.

### 1.2.1 4.1 while: warming up

The while loop is like a repeated if statement. The code is executed over and over again, as long as the condition is `True`. Have another look at its recipe. `~~~ while condition : expression ~~~` Can you tell how many printouts the following while loop will do? `~~~ x = 1 while x < 4 : print(x) x = x + 1 ~~~`

**Instructions**

**Possible Answers - 0 - 1 - 2 - 3 - 4**

### 1.2.2 4.2 Basic while loop

Below you can find the example from the video where the `error` variable, initially equal to 50.0, is divided by 4 and printed out on every run: `~~~ error = 50.0 while error > 1 : error = error / 4 print(error) ~~~` This example will come in handy, because it's time to build a `while` loop yourself! We're going to code a `while` loop that implements a very basic control system for an [inverted pendulum](#). If there's an offset from standing perfectly straight, the `while` loop will incrementally fix this offset.

Note that if your `while` loop takes too long to run, you might have made a mistake. In particular, remember to **indent** the contents of the loop!

#### Instructions

- Create the variable `offset` with an initial value of 8.
- Code a `while` loop that keeps running as long as `offset` is not equal to 0. Inside the `while` loop:
- Print out the sentence "correcting...".
- Next, decrease the value of `offset` by 1. You can do this with `offset = offset - 1`.
- Finally, still within your loop, print out `offset` so you can see how it changes.

```
[42]: # 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
```



### 1.2.3 4.3 Add conditionals

The `while` loop that corrects the `offset` is a good start, but what if `offset` is negative? You can try to run the following code where `offset` is initialized to `-6`: `~~~ # Initialize offset offset = -6`

## 2 Code the while loop

`while offset != 0 : print("correcting...") offset = offset - 1 print(offset) ~~~` but your session will be disconnected. The `while` loop will never stop running, because `offset` will be further decreased on every run. `offset != 0` will never become `False` and the `while` loop continues forever.

Fix things by putting an `if-else` statement inside the `while` loop. If your code is still taking too long to run, you probably made a mistake!

### Instructions

- Inside the `while` loop, complete the `if-else` statement:
  - If `offset` is greater than zero, you should decrease `offset` by 1.
  - Else, you should increase `offset` by 1.
- If you've coded things correctly, hitting *Submit Answer* should work this time.

*If your code is still taking too long to run (or your session is expiring), you probably made a mistake. Check your code and make sure that the statement `offset != 0` will eventually evaluate to `FALSE`!*

```
[43]: # Initialize offset
offset = -6

Code the while loop
while offset != 0 :
 print("correcting...")
 if offset > 0 :
 offset = offset - 1
 else :
 offset = offset + 1
 print(offset)
```

```
correcting...
-5
correcting...
-4
correcting...
-3
correcting...
-2
correcting...
-1
correcting...
0
```

### 2.0.1 4.4 Loop over a list

Have another look at the for loop that Filip showed in the video: `~~~ fam = [1.73, 1.68, 1.71, 1.89]`  
`for height in fam : print(height) ~~~`

As usual, you simply have to indent the code with 4 spaces to tell Python which code should be executed in the for loop.

The `areas` variable, containing the area of different rooms in your house, is already defined.

#### Instructions

Write a for loop that iterates over all elements of the `areas` list and prints out every element separately.

```
[44]: # areas list
areas = ['hallway', 11.25, 'kitchen', 18.0, 'living room', 20.0, '
 bedroom', 10.75, 'bathroom', 9.50]

Code the for loop
for i in areas :
 print(i)
```

```
11.25
18.0
20.0
10.75
9.5
```

### 2.0.2 4.5 Indexes and values (1)

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) + ":" + str(height))`

#### Instructions

- Adapt the for loop in the sample code to use `enumerate()` and use two iterator variables.
- Update the `print()` statement so that on each run, a line of the form "room x: y" should be printed, where x is the index of the list element and y is the actual list element, i.e. the area. Make sure to print out this exact string, with the correct spacing.

```
[45]: # areas list
areas = ['hallway', 11.25, 'kitchen', 18.0, 'living room', 20.0, '
 bedroom', 10.75, 'bathroom', 9.50]

Change for loop to use enumerate()
for x, y in enumerate(areas) :
 print("room " + str(x) + ": " + str(y))
```

```
room 0: 11.25
room 1: 18.0
room 2: 20.0
```

```
room 3: 10.75
room 4: 9.5
```

### 2.0.3 4.6 Indexes and values (2)

For non-programmer folks, room 0: 11.25 is strange. Wouldn't it be better if the count started at 1?

#### Instructions

- Adapt the `print()` function in the `for` loop on the right so that the first printout becomes "room 1: 11.25", the second one "room 2: 18.0" and so on.

```
[46]: # areas list
areas = ['hallway', 11.25, 'kitchen', 18.0, 'living room', 20.0, '
 bedroom', 10.75, 'bathroom', 9.50]

Code the for loop
for index, area in enumerate(areas) :
 print("room " + str(index + 1) + ": " + str(area))
```

```
room 1: 11.25
room 2: 18.0
room 3: 20.0
room 4: 10.75
room 5: 9.5
```

### 2.0.4 4.7 Loop over list of lists

Remember the `house` variable from the Intro to Python course? Have a look at its definition on the right. It's basically a list of lists, where each sublist contains the name and area of a room in your house.

It's up to you to build a `for` loop from scratch this time!

#### Instructions

Write a `for` loop that goes through each sublist of `house` and prints out the `x is y sqm`, where `x` is the name of the room and `y` is the area of the room.

```
[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 i in house :
 print("the " + i[0] + " is " + str(i[1]) + " sqm")
```

```
the hallway is 11.25 sqm
the kitchen is 18.0 sqm
the living room is 20.0 sqm
the bedroom is 10.75 sqm
the bathroom is 9.5 sqm
```

### 2.0.5 4.8 Loop over dictionary

In Python 3, you need the `items()` method to loop over a dictionary: `~~~ world = { "afghanistan":30.55, "albania":2.77, "algeria":39.21 }`

for key, value in world.items() : print(key + " - " + str(value)) `~~~` Remember the `europa` dictionary that contained the names of some European countries as key and their capitals as corresponding value? Go ahead and write a loop to iterate over it!

#### Instructions

Write a `for` loop that goes through each key:value pair of `europa`. On each iteration, "the capital of x is y" should be printed out, where x is the key and y is the value of the pair.

```
[48]: # Definition of dictionary
europa = {'spain':'madrid', 'france':'paris', 'germany':'berlin',
 'norway':'oslo', 'italy':'rome', 'poland':'warsaw', 'austria':
 'vienna' }

Iterate over europa
for key, value in europa.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
```

*Great! Notice that the order of the printouts doesn't necessarily correspond with the order used when defining `europa`. Remember: dictionaries are inherently unordered!*

### 2.0.6 4.9 Loop over Numpy array

If you're dealing with a 1D Numpy array, looping over all elements can be as simple as: `~~~ for x in my_array : ...` `~~~` If you're dealing with a 2D Numpy array, it's more complicated. A 2D array is built up of multiple 1D arrays. To explicitly iterate over all separate elements of a multi-dimensional array, you'll need this syntax: `~~~ for x in np.nditer(my_array) : ...` `~~~` Two Numpy arrays that you might recognize from the intro course are available in your Python session: `np_height`, a Numpy array containing the heights of Major League Baseball players, and `np_baseball`, a 2D Numpy array that contains both the heights (first column) and weights (second column) of those players.

## Instructions

- Import the numpy package under the local alias np.
- Write a for loop that iterates over all elements in np\_height and prints out "x inches" for each element, where x is the value in the array.
- Write a for loop that visits every element of the np\_baseball array and prints it out.

```
[56]: import pandas as pd
df = pd.read_csv('./Data/baseball.csv')
df_baseball = df[['Height', 'Weight']]
np_baseball = np.array(df_baseball)
np_height = np.array(df_baseball.Height)
```

```
[57]: # Import numpy as np
import numpy as np

For loop over np_height
for i in np_height :
 print(str(i) + " inches")

For loop over np_baseball
for i in np.nditer(np_baseball) :
 print(i, end='\n')
```

```
74 inches
74 inches
72 inches
72 inches
73 inches
69 inches
69 inches
71 inches
76 inches
71 inches
73 inches
73 inches
74 inches
74 inches
69 inches
70 inches
73 inches
75 inches
78 inches
79 inches
76 inches
74 inches
76 inches
72 inches
```

71 inches  
75 inches  
77 inches  
74 inches  
73 inches  
74 inches  
78 inches  
73 inches  
75 inches  
73 inches  
75 inches  
75 inches  
74 inches  
69 inches  
71 inches  
74 inches  
73 inches  
73 inches  
76 inches  
74 inches  
74 inches  
70 inches  
72 inches  
77 inches  
74 inches  
70 inches  
73 inches  
75 inches  
76 inches  
76 inches  
78 inches  
74 inches  
74 inches  
76 inches  
77 inches  
81 inches  
78 inches  
75 inches  
77 inches  
75 inches  
76 inches  
74 inches  
72 inches  
72 inches  
75 inches  
73 inches  
73 inches  
73 inches

70 inches  
70 inches  
70 inches  
76 inches  
68 inches  
71 inches  
72 inches  
75 inches  
75 inches  
75 inches  
75 inches  
68 inches  
74 inches  
78 inches  
71 inches  
73 inches  
76 inches  
74 inches  
74 inches  
79 inches  
75 inches  
73 inches  
76 inches  
74 inches  
74 inches  
73 inches  
72 inches  
74 inches  
73 inches  
74 inches  
72 inches  
73 inches  
69 inches  
72 inches  
73 inches  
75 inches  
75 inches  
73 inches  
72 inches  
72 inches  
76 inches  
74 inches  
72 inches  
77 inches  
74 inches  
77 inches  
75 inches  
76 inches

80 inches  
74 inches  
74 inches  
75 inches  
78 inches  
73 inches  
73 inches  
74 inches  
75 inches  
76 inches  
71 inches  
73 inches  
74 inches  
76 inches  
76 inches  
74 inches  
73 inches  
74 inches  
70 inches  
72 inches  
73 inches  
73 inches  
73 inches  
73 inches  
71 inches  
74 inches  
74 inches  
72 inches  
74 inches  
71 inches  
74 inches  
73 inches  
75 inches  
75 inches  
79 inches  
73 inches  
75 inches  
76 inches  
74 inches  
76 inches  
78 inches  
74 inches  
76 inches  
72 inches  
74 inches  
76 inches  
74 inches  
75 inches



78 inches  
75 inches  
72 inches  
74 inches  
72 inches  
74 inches  
70 inches  
71 inches  
70 inches  
75 inches  
71 inches  
71 inches  
73 inches  
72 inches  
71 inches  
73 inches  
72 inches  
75 inches  
74 inches  
74 inches  
75 inches  
73 inches  
77 inches  
73 inches  
76 inches  
75 inches  
74 inches  
76 inches  
75 inches  
73 inches  
71 inches  
76 inches  
75 inches  
72 inches  
71 inches  
77 inches  
73 inches  
74 inches  
71 inches  
72 inches  
74 inches  
75 inches  
73 inches  
72 inches  
75 inches  
75 inches  
74 inches  
72 inches

74 inches  
71 inches  
70 inches  
74 inches  
77 inches  
77 inches  
75 inches  
75 inches  
78 inches  
75 inches  
76 inches  
73 inches  
75 inches  
75 inches  
79 inches  
77 inches  
76 inches  
71 inches  
75 inches  
74 inches  
69 inches  
71 inches  
76 inches  
72 inches  
72 inches  
70 inches  
72 inches  
73 inches  
71 inches  
72 inches  
71 inches  
73 inches  
72 inches  
73 inches  
74 inches  
74 inches  
72 inches  
75 inches  
74 inches  
74 inches  
77 inches  
75 inches  
73 inches  
72 inches  
71 inches  
74 inches  
77 inches  
75 inches

75 inches  
75 inches  
78 inches  
78 inches  
74 inches  
76 inches  
78 inches  
76 inches  
70 inches  
72 inches  
80 inches  
74 inches  
74 inches  
71 inches  
70 inches  
72 inches  
71 inches  
74 inches  
71 inches  
72 inches  
71 inches  
74 inches  
69 inches  
76 inches  
75 inches  
75 inches  
76 inches  
73 inches  
76 inches  
73 inches  
77 inches  
73 inches  
72 inches  
72 inches  
77 inches  
77 inches  
71 inches  
74 inches  
74 inches  
73 inches  
78 inches  
75 inches  
73 inches  
70 inches  
74 inches  
72 inches  
73 inches  
73 inches

75 inches  
75 inches  
74 inches  
76 inches  
73 inches  
74 inches  
75 inches  
75 inches  
72 inches  
73 inches  
73 inches  
72 inches  
74 inches  
78 inches  
76 inches  
73 inches  
74 inches  
75 inches  
70 inches  
75 inches  
71 inches  
72 inches  
78 inches  
75 inches  
73 inches  
73 inches  
71 inches  
75 inches  
77 inches  
72 inches  
69 inches  
73 inches  
74 inches  
72 inches  
70 inches  
75 inches  
70 inches  
72 inches  
72 inches  
74 inches  
73 inches  
74 inches  
76 inches  
75 inches  
80 inches  
72 inches  
75 inches  
73 inches

74 inches  
74 inches  
73 inches  
75 inches  
75 inches  
71 inches  
73 inches  
75 inches  
74 inches  
74 inches  
72 inches  
74 inches  
74 inches  
74 inches  
73 inches  
76 inches  
75 inches  
72 inches  
73 inches  
73 inches  
73 inches  
72 inches  
72 inches  
72 inches  
72 inches  
71 inches  
75 inches  
75 inches  
74 inches  
73 inches  
75 inches  
79 inches  
74 inches  
76 inches  
73 inches  
74 inches  
74 inches  
72 inches  
74 inches  
74 inches  
75 inches  
78 inches  
74 inches  
74 inches  
74 inches  
77 inches  
70 inches  
73 inches

74 inches  
73 inches  
71 inches  
75 inches  
71 inches  
72 inches  
77 inches  
74 inches  
70 inches  
77 inches  
73 inches  
72 inches  
76 inches  
71 inches  
76 inches  
78 inches  
75 inches  
73 inches  
78 inches  
74 inches  
79 inches  
75 inches  
76 inches  
72 inches  
75 inches  
75 inches  
70 inches  
72 inches  
70 inches  
74 inches  
71 inches  
76 inches  
73 inches  
76 inches  
71 inches  
69 inches  
72 inches  
72 inches  
69 inches  
73 inches  
69 inches  
73 inches  
74 inches  
74 inches  
72 inches  
71 inches  
72 inches  
72 inches

76 inches  
76 inches  
76 inches  
74 inches  
76 inches  
75 inches  
71 inches  
72 inches  
71 inches  
73 inches  
75 inches  
76 inches  
75 inches  
71 inches  
75 inches  
74 inches  
72 inches  
73 inches  
73 inches  
73 inches  
73 inches  
76 inches  
72 inches  
76 inches  
73 inches  
73 inches  
73 inches  
75 inches  
75 inches  
77 inches  
73 inches  
72 inches  
75 inches  
70 inches  
74 inches  
72 inches  
80 inches  
71 inches  
71 inches  
74 inches  
74 inches  
73 inches  
75 inches  
76 inches  
73 inches  
77 inches  
72 inches  
73 inches

77 inches  
76 inches  
71 inches  
75 inches  
73 inches  
74 inches  
77 inches  
71 inches  
72 inches  
73 inches  
69 inches  
73 inches  
70 inches  
74 inches  
76 inches  
73 inches  
73 inches  
75 inches  
73 inches  
79 inches  
74 inches  
73 inches  
74 inches  
77 inches  
75 inches  
74 inches  
73 inches  
77 inches  
73 inches  
77 inches  
74 inches  
74 inches  
73 inches  
77 inches  
74 inches  
77 inches  
75 inches  
77 inches  
75 inches  
71 inches  
74 inches  
70 inches  
79 inches  
72 inches  
72 inches  
70 inches  
74 inches  
74 inches



72 inches  
73 inches  
72 inches  
74 inches  
74 inches  
76 inches  
82 inches  
74 inches  
74 inches  
70 inches  
73 inches  
73 inches  
74 inches  
77 inches  
72 inches  
76 inches  
73 inches  
73 inches  
72 inches  
74 inches  
74 inches  
71 inches  
72 inches  
75 inches  
74 inches  
74 inches  
77 inches  
70 inches  
71 inches  
73 inches  
76 inches  
71 inches  
75 inches  
74 inches  
72 inches  
76 inches  
79 inches  
76 inches  
73 inches  
76 inches  
78 inches  
75 inches  
76 inches  
72 inches  
72 inches  
73 inches  
73 inches  
75 inches

71 inches  
76 inches  
70 inches  
75 inches  
74 inches  
75 inches  
73 inches  
71 inches  
71 inches  
72 inches  
73 inches  
73 inches  
72 inches  
69 inches  
73 inches  
78 inches  
71 inches  
73 inches  
75 inches  
76 inches  
70 inches  
74 inches  
77 inches  
75 inches  
79 inches  
72 inches  
77 inches  
73 inches  
75 inches  
75 inches  
75 inches  
73 inches  
73 inches  
76 inches  
77 inches  
75 inches  
70 inches  
71 inches  
71 inches  
75 inches  
74 inches  
69 inches  
70 inches  
75 inches  
72 inches  
75 inches  
73 inches  
72 inches

72 inches  
72 inches  
76 inches  
75 inches  
74 inches  
69 inches  
73 inches  
72 inches  
72 inches  
75 inches  
77 inches  
76 inches  
80 inches  
77 inches  
76 inches  
79 inches  
71 inches  
75 inches  
73 inches  
76 inches  
77 inches  
73 inches  
76 inches  
70 inches  
75 inches  
73 inches  
75 inches  
70 inches  
69 inches  
71 inches  
72 inches  
72 inches  
73 inches  
70 inches  
70 inches  
73 inches  
76 inches  
75 inches  
72 inches  
73 inches  
79 inches  
71 inches  
72 inches  
74 inches  
74 inches  
74 inches  
72 inches  
76 inches

76 inches  
72 inches  
72 inches  
71 inches  
72 inches  
72 inches  
70 inches  
77 inches  
74 inches  
72 inches  
76 inches  
71 inches  
76 inches  
71 inches  
73 inches  
70 inches  
73 inches  
73 inches  
72 inches  
71 inches  
71 inches  
71 inches  
72 inches  
72 inches  
74 inches  
74 inches  
74 inches  
71 inches  
72 inches  
75 inches  
72 inches  
71 inches  
72 inches  
72 inches  
72 inches  
72 inches  
74 inches  
74 inches  
77 inches  
75 inches  
73 inches  
75 inches  
73 inches  
76 inches  
72 inches  
77 inches  
75 inches  
72 inches

71 inches  
71 inches  
75 inches  
72 inches  
73 inches  
73 inches  
71 inches  
70 inches  
75 inches  
71 inches  
76 inches  
73 inches  
68 inches  
71 inches  
72 inches  
74 inches  
77 inches  
72 inches  
76 inches  
78 inches  
81 inches  
72 inches  
73 inches  
76 inches  
72 inches  
72 inches  
74 inches  
76 inches  
73 inches  
76 inches  
75 inches  
70 inches  
71 inches  
74 inches  
72 inches  
73 inches  
76 inches  
76 inches  
73 inches  
71 inches  
68 inches  
71 inches  
71 inches  
74 inches  
77 inches  
69 inches  
72 inches  
76 inches

75 inches  
76 inches  
75 inches  
76 inches  
72 inches  
74 inches  
76 inches  
74 inches  
72 inches  
75 inches  
78 inches  
77 inches  
70 inches  
72 inches  
79 inches  
74 inches  
71 inches  
68 inches  
77 inches  
75 inches  
71 inches  
72 inches  
70 inches  
72 inches  
72 inches  
73 inches  
72 inches  
74 inches  
72 inches  
72 inches  
75 inches  
72 inches  
73 inches  
74 inches  
72 inches  
78 inches  
75 inches  
72 inches  
74 inches  
75 inches  
75 inches  
76 inches  
74 inches  
74 inches  
73 inches  
74 inches  
71 inches  
74 inches

75 inches  
76 inches  
74 inches  
76 inches  
76 inches  
73 inches  
75 inches  
75 inches  
74 inches  
68 inches  
72 inches  
75 inches  
71 inches  
70 inches  
72 inches  
73 inches  
72 inches  
75 inches  
74 inches  
70 inches  
76 inches  
71 inches  
82 inches  
72 inches  
73 inches  
74 inches  
71 inches  
75 inches  
77 inches  
72 inches  
74 inches  
72 inches  
73 inches  
78 inches  
77 inches  
73 inches  
73 inches  
73 inches  
73 inches  
73 inches  
76 inches  
75 inches  
70 inches  
73 inches  
72 inches  
73 inches  
75 inches  
74 inches

73 inches  
73 inches  
76 inches  
73 inches  
75 inches  
70 inches  
77 inches  
72 inches  
77 inches  
74 inches  
75 inches  
75 inches  
75 inches  
75 inches  
72 inches  
74 inches  
71 inches  
76 inches  
71 inches  
75 inches  
76 inches  
83 inches  
75 inches  
74 inches  
76 inches  
72 inches  
72 inches  
75 inches  
75 inches  
72 inches  
77 inches  
73 inches  
72 inches  
70 inches  
74 inches  
72 inches  
74 inches  
72 inches  
71 inches  
70 inches  
71 inches  
76 inches  
74 inches  
76 inches  
74 inches  
74 inches  
74 inches  
75 inches



75 inches  
71 inches  
71 inches  
74 inches  
77 inches  
71 inches  
74 inches  
75 inches  
77 inches  
76 inches  
74 inches  
76 inches  
72 inches  
71 inches  
72 inches  
75 inches  
73 inches  
68 inches  
72 inches  
69 inches  
73 inches  
73 inches  
75 inches  
70 inches  
70 inches  
74 inches  
75 inches  
74 inches  
74 inches  
73 inches  
74 inches  
75 inches  
77 inches  
73 inches  
74 inches  
76 inches  
74 inches  
75 inches  
73 inches  
76 inches  
78 inches  
75 inches  
73 inches  
77 inches  
74 inches  
72 inches  
74 inches  
72 inches

71 inches  
73 inches  
75 inches  
73 inches  
67 inches  
67 inches  
76 inches  
74 inches  
73 inches  
70 inches  
75 inches  
70 inches  
72 inches  
77 inches  
79 inches  
78 inches  
74 inches  
75 inches  
75 inches  
78 inches  
76 inches  
75 inches  
69 inches  
75 inches  
72 inches  
75 inches  
73 inches  
74 inches  
75 inches  
75 inches  
73 inches  
74  
74  
72  
72  
73  
69  
69  
71  
76  
71  
73  
73  
74  
74  
69  
70  
73

75  
78  
79  
76  
74  
76  
72  
71  
75  
77  
74  
73  
74  
78  
73  
75  
73  
75  
75  
74  
69  
71  
74  
73  
73  
76  
74  
74  
70  
72  
77  
74  
70  
73  
75  
76  
76  
78  
74  
74  
76  
77  
81  
78  
75  
77  
75  
76

74  
72  
72  
75  
73  
73  
73  
70  
70  
70  
76  
68  
71  
72  
75  
75  
75  
75  
68  
74  
78  
71  
73  
76  
74  
74  
79  
75  
73  
76  
74  
74  
73  
72  
74  
73  
74  
72  
73  
69  
72  
73  
75  
75  
73  
72  
72  
76

74  
72  
77  
74  
77  
75  
76  
80  
74  
74  
75  
78  
73  
73  
74  
75  
76  
71  
73  
74  
76  
76  
74  
73  
74  
70  
72  
73  
73  
73  
73  
71  
74  
74  
72  
74  
71  
74  
73  
75  
75  
79  
73  
75  
76  
74  
76  
78

74  
76  
72  
74  
76  
74  
75  
78  
75  
72  
74  
72  
74  
70  
71  
70  
75  
71  
71  
73  
72  
71  
73  
72  
75  
74  
74  
75  
73  
77  
73  
76  
75  
74  
76  
75  
73  
71  
76  
75  
72  
71  
77  
73  
74  
71  
72  
74

75  
73  
72  
75  
75  
74  
72  
74  
71  
70  
74  
77  
77  
75  
75  
78  
75  
76  
73  
75  
75  
79  
77  
76  
71  
75  
74  
69  
71  
76  
72  
72  
70  
72  
73  
71  
72  
71  
73  
72  
73  
74  
74  
72  
75  
74  
74  
77

75  
73  
72  
71  
74  
77  
75  
75  
75  
78  
78  
74  
76  
78  
76  
70  
72  
80  
74  
74  
71  
70  
72  
71  
74  
71  
72  
71  
74  
69  
76  
75  
75  
76  
73  
76  
73  
77  
73  
72  
72  
77  
77  
71  
74  
74  
73  
78



75  
73  
70  
74  
72  
73  
73  
75  
75  
74  
76  
73  
74  
75  
75  
72  
73  
73  
72  
74  
78  
76  
73  
74  
75  
70  
75  
71  
72  
78  
75  
73  
73  
71  
75  
77  
72  
69  
73  
74  
72  
70  
75  
70  
72  
72  
74  
73

74  
76  
75  
80  
72  
75  
73  
74  
74  
73  
75  
75  
71  
73  
75  
74  
74  
72  
74  
74  
74  
73  
76  
75  
72  
73  
73  
73  
72  
72  
72  
72  
71  
75  
75  
74  
73  
75  
79  
74  
76  
73  
74  
74  
72  
74  
74  
75

78  
74  
74  
74  
77  
70  
73  
74  
73  
71  
75  
71  
72  
77  
74  
70  
77  
73  
72  
76  
71  
76  
78  
75  
73  
78  
74  
79  
75  
76  
72  
75  
75  
70  
72  
70  
74  
71  
76  
73  
76  
71  
69  
72  
72  
69  
73  
69

73  
74  
74  
72  
71  
72  
72  
76  
76  
76  
74  
76  
75  
71  
72  
71  
73  
75  
76  
75  
71  
75  
74  
72  
73  
73  
73  
73  
76  
72  
76  
73  
73  
73  
75  
75  
77  
73  
72  
75  
70  
74  
72  
80  
71  
71  
74  
74

73  
75  
76  
73  
77  
72  
73  
77  
76  
71  
75  
73  
74  
77  
71  
72  
73  
69  
73  
70  
74  
76  
73  
73  
75  
73  
79  
74  
73  
74  
77  
75  
74  
73  
77  
73  
77  
74  
74  
73  
77  
74  
77  
75  
77  
75  
71  
74

70  
79  
72  
72  
70  
74  
74  
72  
73  
72  
74  
74  
76  
82  
74  
74  
70  
73  
73  
74  
77  
72  
76  
73  
73  
72  
74  
74  
71  
72  
75  
74  
74  
77  
70  
71  
73  
76  
71  
75  
74  
72  
76  
79  
76  
73  
76  
78

75  
76  
72  
72  
73  
73  
75  
71  
76  
70  
75  
74  
75  
73  
71  
71  
72  
73  
73  
72  
69  
73  
78  
71  
73  
75  
76  
70  
74  
77  
75  
79  
72  
77  
73  
75  
75  
75  
73  
73  
76  
77  
75  
70  
71  
71  
75  
74

69  
70  
75  
72  
75  
73  
72  
72  
72  
76  
75  
74  
69  
73  
72  
72  
75  
77  
76  
80  
77  
76  
79  
71  
75  
73  
76  
77  
73  
76  
70  
75  
73  
75  
70  
69  
71  
72  
72  
73  
70  
70  
73  
76  
75  
72  
73  
79



71  
72  
74  
74  
74  
72  
76  
76  
72  
72  
71  
72  
72  
70  
77  
74  
72  
76  
71  
76  
71  
73  
70  
73  
73  
72  
71  
71  
71  
72  
72  
74  
74  
74  
71  
72  
75  
72  
71  
72  
72  
72  
72  
72  
74  
74  
77  
75  
73

75  
73  
76  
72  
77  
75  
72  
71  
71  
75  
72  
73  
73  
71  
70  
75  
71  
76  
73  
68  
71  
72  
74  
77  
72  
76  
78  
81  
72  
73  
76  
72  
72  
74  
76  
73  
76  
75  
70  
71  
74  
72  
73  
76  
76  
73  
71  
68

71  
71  
74  
77  
69  
72  
76  
75  
76  
75  
76  
72  
74  
76  
74  
72  
75  
78  
77  
70  
72  
79  
74  
71  
68  
77  
75  
71  
72  
70  
72  
72  
73  
72  
74  
72  
72  
75  
72  
73  
74  
72  
78  
75  
72  
74  
75  
75

76  
74  
74  
73  
74  
71  
74  
75  
76  
74  
76  
76  
73  
75  
75  
74  
68  
72  
75  
71  
70  
72  
73  
72  
75  
74  
70  
76  
71  
82  
72  
73  
74  
71  
75  
77  
72  
74  
72  
73  
78  
77  
73  
73  
73  
73  
73  
76

75  
70  
73  
72  
73  
75  
74  
73  
73  
76  
73  
75  
70  
77  
72  
77  
74  
75  
75  
75  
75  
72  
74  
71  
76  
71  
75  
76  
83  
75  
74  
76  
72  
72  
75  
75  
72  
77  
73  
72  
70  
74  
72  
74  
72  
71  
70  
71

76  
74  
76  
74  
74  
74  
75  
75  
71  
71  
74  
77  
71  
74  
75  
77  
76  
74  
76  
72  
71  
72  
75  
73  
68  
72  
69  
73  
73  
75  
70  
70  
74  
75  
74  
74  
73  
74  
75  
77  
73  
74  
76  
74  
75  
73  
76  
78

75  
73  
77  
74  
72  
74  
72  
71  
73  
75  
73  
67  
67  
76  
74  
73  
70  
75  
70  
72  
77  
79  
78  
74  
75  
75  
78  
76  
75  
69  
75  
72  
75  
73  
74  
75  
75  
73  
180  
215  
210  
210  
188  
176  
209  
200  
231  
180

188  
180  
185  
160  
180  
185  
189  
185  
219  
230  
205  
230  
195  
180  
192  
225  
203  
195  
182  
188  
200  
180  
200  
200  
245  
240  
215  
185  
175  
199  
200  
215  
200  
205  
206  
186  
188  
220  
210  
195  
200  
200  
212  
224  
210  
205  
220  
195



200  
260  
228  
270  
200  
210  
190  
220  
180  
205  
210  
220  
211  
200  
180  
190  
170  
230  
155  
185  
185  
200  
225  
225  
220  
160  
205  
235  
250  
210  
190  
160  
200  
205  
222  
195  
205  
220  
220  
170  
185  
195  
220  
230  
180  
220  
180  
180

170  
210  
215  
200  
213  
180  
192  
235  
185  
235  
210  
222  
210  
230  
220  
180  
190  
200  
210  
194  
180  
190  
240  
200  
198  
200  
195  
210  
220  
190  
210  
225  
180  
185  
170  
185  
185  
180  
178  
175  
200  
204  
211  
190  
210  
190  
190  
185

290  
175  
185  
200  
220  
170  
220  
190  
220  
205  
200  
250  
225  
215  
210  
215  
195  
200  
194  
220  
180  
180  
170  
195  
180  
170  
206  
205  
200  
225  
201  
225  
233  
180  
225  
180  
220  
180  
237  
215  
190  
235  
190  
180  
165  
195  
200  
190

190  
185  
185  
205  
190  
205  
206  
220  
208  
170  
195  
210  
190  
211  
230  
170  
185  
185  
241  
225  
210  
175  
230  
200  
215  
198  
226  
278  
215  
230  
240  
184  
219  
170  
218  
190  
225  
220  
176  
190  
197  
204  
167  
180  
195  
220  
215  
185

190  
205  
205  
200  
210  
215  
200  
205  
211  
190  
208  
200  
210  
232  
230  
210  
220  
210  
202  
212  
225  
170  
190  
200  
237  
220  
170  
193  
190  
150  
220  
200  
190  
185  
185  
200  
172  
220  
225  
190  
195  
219  
190  
197  
200  
195  
210  
177

220  
235  
180  
195  
195  
190  
230  
190  
200  
190  
190  
200  
200  
184  
200  
180  
219  
187  
200  
220  
205  
190  
170  
160  
215  
175  
205  
200  
214  
200  
190  
180  
205  
220  
190  
215  
235  
191  
200  
181  
200  
210  
240  
185  
165  
190  
185  
175

155  
210  
170  
175  
220  
210  
205  
200  
205  
195  
240  
150  
200  
215  
202  
200  
190  
205  
190  
160  
215  
185  
200  
190  
210  
185  
220  
190  
202  
205  
220  
175  
160  
190  
200  
229  
206  
220  
180  
195  
175  
188  
230  
190  
200  
190  
219  
235

180  
180  
180  
200  
234  
185  
220  
223  
200  
210  
200  
210  
190  
177  
227  
180  
195  
199  
175  
185  
240  
210  
180  
194  
225  
180  
205  
193  
230  
230  
220  
200  
249  
190  
208  
245  
250  
160  
192  
220  
170  
197  
155  
190  
200  
220  
210  
228



190  
160  
184  
180  
180  
200  
176  
160  
222  
211  
195  
200  
175  
206  
240  
185  
260  
185  
221  
205  
200  
170  
201  
205  
185  
205  
245  
220  
210  
220  
185  
175  
170  
180  
200  
210  
175  
220  
206  
180  
210  
195  
200  
200  
164  
180  
220  
195

205  
170  
240  
210  
195  
200  
205  
192  
190  
170  
240  
200  
205  
175  
250  
220  
224  
210  
195  
180  
245  
175  
180  
215  
175  
180  
195  
230  
230  
205  
215  
195  
180  
205  
180  
190  
180  
190  
190  
220  
210  
255  
190  
230  
200  
205  
210  
225

215  
220  
205  
200  
220  
197  
225  
187  
245  
185  
185  
175  
200  
180  
188  
225  
200  
210  
245  
213  
231  
165  
228  
210  
250  
191  
190  
200  
215  
254  
232  
180  
215  
220  
180  
200  
170  
195  
210  
200  
220  
165  
180  
200  
200  
170  
224  
220

180  
198  
240  
239  
185  
210  
220  
200  
195  
220  
230  
170  
220  
230  
165  
205  
192  
210  
205  
200  
210  
185  
195  
202  
205  
195  
180  
200  
185  
240  
185  
220  
205  
205  
180  
201  
190  
208  
240  
180  
230  
195  
215  
190  
195  
215  
215  
220

220  
230  
195  
190  
195  
209  
204  
170  
185  
205  
175  
210  
190  
180  
180  
160  
235  
200  
210  
180  
190  
197  
203  
205  
170  
200  
250  
200  
220  
200  
190  
170  
190  
220  
215  
206  
215  
185  
235  
188  
230  
195  
168  
190  
160  
200  
200  
189

180  
190  
200  
220  
187  
240  
190  
180  
185  
210  
220  
219  
190  
193  
175  
180  
215  
210  
200  
190  
185  
220  
170  
195  
205  
195  
210  
190  
190  
180  
220  
190  
186  
185  
190  
180  
190  
170  
210  
240  
220  
180  
210  
210  
195  
160  
180  
205

200  
185  
245  
190  
210  
200  
200  
222  
215  
240  
170  
220  
156  
190  
202  
221  
200  
190  
210  
190  
200  
165  
190  
185  
230  
208  
209  
175  
180  
200  
205  
200  
250  
210  
230  
244  
202  
240  
200  
215  
177  
210  
170  
215  
217  
198  
200  
220

170  
200  
230  
231  
183  
192  
167  
190  
180  
180  
215  
160  
205  
223  
175  
170  
190  
240  
175  
230  
223  
196  
167  
195  
190  
250  
190  
190  
190  
170  
160  
150  
225  
220  
209  
210  
176  
260  
195  
190  
184  
180  
195  
195  
219  
225  
212  
202



185  
200  
209  
200  
195  
228  
210  
190  
212  
190  
218  
220  
190  
235  
210  
200  
188  
210  
235  
188  
215  
216  
220  
180  
185  
200  
210  
220  
185  
231  
210  
195  
200  
205  
200  
190  
250  
185  
180  
170  
180  
208  
235  
215  
244  
220  
185  
230

190  
200  
180  
190  
196  
180  
230  
224  
160  
178  
205  
185  
210  
180  
190  
200  
257  
190  
220  
165  
205  
200  
208  
185  
215  
170  
235  
210  
170  
180  
170  
190  
150  
230  
203  
260  
246  
186  
210  
198  
210  
215  
180  
200  
245  
200  
192  
192

200  
192  
205  
190  
186  
170  
197  
219  
200  
220  
207  
225  
207  
212  
225  
170  
190  
210  
230  
210  
200  
238  
234  
222  
200  
190  
170  
220  
223  
210  
215  
196  
175  
175  
189  
205  
210  
180  
180  
197  
220  
228  
190  
204  
165  
216  
220  
208

210  
215  
195  
200  
215  
229  
240  
207  
205  
208  
185  
190  
170  
208  
225  
190  
225  
185  
180  
165  
240  
220  
212  
163  
215  
175  
205  
210  
205  
208  
215  
180  
200  
230  
211  
230  
190  
220  
180  
205  
190  
180  
205  
190  
195

### 2.0.7 4.10 Loop over DataFrame (1)

Iterating over a Pandas DataFrame is typically done with the `iterrows()` method. Used in a `for` loop, every observation is iterated over and on every iteration the row label and actual row contents are available: `~~~ for lab, row in brics.iterrows() : ... ~~~` In this and the following exercises you will be working on the `cars` DataFrame. It contains information on the `cars` per capita and whether people drive right or left for seven countries in the world.

#### Instructions

- Write a `for` loop that iterates over the rows of `cars` and on each iteration perform two `print()` calls: one to print out the row label and one to print out all of the rows contents.

```
[60]: # Import cars data
import pandas as pd
cars = pd.read_csv('./data/cars.csv', index_col = 0)

Iterate over rows of cars
for lab, row in cars.iterrows() :
 print(lab)
 print(row)
```

```
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
```

## 2.0.8 4.11 Loop over DataFrame (2)

The row data that's generated by `iterrows()` on every run is a Pandas Series. This format is not very convenient to print out. Luckily, you can easily select variables from the Pandas Series using square brackets: `~~~ for lab, row in brics.iterrows(): print(row['country']) ~~~` **Instructions**

- Using the iterators `lab` and `row`, adapt the code in the for loop such that the first iteration prints out "US: 809", the second iteration "AUS: 731", and so on.
- The output should be in the form "country: cars\_per\_cap". Make sure to print out this exact string (with the correct spacing).
  - You can use `str()` to convert your integer data to a string so that you can print it in conjunction with the country label.

```
[61]: # Import cars data
import pandas as pd
cars = pd.read_csv('./data/cars.csv', index_col = 0)

Adapt for loop
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
```

## 2.0.9 4.12 Add column (1)

In the video, Filip showed you how to add the length of the country names of the brics DataFrame in a new column: `~~~ for lab, row in brics.iterrows(): brics.loc[lab, "name_length"] = len(row["country"]) ~~~` You can do similar things on the cars DataFrame.

### Instructions

- Use a `for` loop to add a new column, named `COUNTRY`, that contains a uppercase version of the country names in the "country" column. You can use the string method `upper()` for this.

- To see if your code worked, print out `cars`. Don't indent this code, so that it's not part of the `for` loop.

```
[62]: # Import cars data
import pandas as pd
cars = pd.read_csv('./data/cars.csv', index_col = 0)

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

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         |

#### 2.0.10 4.13 Add column (2)

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` (<http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.apply.html>) 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) ~~~` We can do a similar thing to call the `upper()` method on every name in the `country` column. However, `upper()` is a method, so we'll need a slightly different approach:

#### Instructions

- Replace the `for` loop with a one-liner that uses `.apply(str.upper)`. The call should give the same result: a column `COUNTRY` should be added to `cars`, containing an uppercase version of the country names.
- As usual, print out `cars` to see the fruits of your hard labor

```
[63]: # Import cars data
import pandas as pd
cars = pd.read_csv('./data/cars.csv', index_col = 0)
```

```
Use .apply(str.upper)
cars["COUNTRY"] = cars["country"].apply(str.upper)
print(cars)
```

|     | <code>cars_per_cap</code> | <code>country</code> | <code>drives_right</code> | <code>COUNTRY</code> |
|-----|---------------------------|----------------------|---------------------------|----------------------|
| 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                |

## 2.1 5. Case Study: Hacker Statistics

This chapter will allow you to apply all the concepts you’ve learned in this course. You will use hacker statistics to calculate your chances of winning a bet. Use random number generators, loops, and Matplotlib to gain a competitive edge!

### 2.1.1 5.1 Random float

Randomness has many uses in science, art, statistics, cryptography, gaming, gambling, and other fields. You’re going to use randomness to simulate a game.

All the functionality you need is contained in the `random` package, a sub-package of `numpy`. In this exercise, you’ll be using two functions from this package:

- `seed()`: sets the random seed, so that your results are reproducible between simulations. As an argument, it takes an integer of your choosing. If you call the function, no output will be generated.
- `rand()`: if you don’t specify any arguments, it generates a random float between zero and one.

#### Instructions

- Import `numpy` as `np`.
- Use `seed()` to set the seed; as an argument, pass 123.
- Generate your first random float with `rand()` and print it out.

```
[64]: # Import numpy as np
import numpy as np

Set the seed
np.random.seed(123)

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

0.6964691855978616



### 2.1.2 5.2 Roll the dice

In the previous exercise, you used `rand()`, that generates a random float between 0 and 1.

As Filip explained in the video you can just as well use `randint()`, also a function of the random package, to generate integers randomly. The following call generates the integer 4, 5, 6 or 7 randomly. **8 is not included**. ~~~ `import numpy as np` `np.random.randint(4, 8)` ~~~

Numpy has already been imported as `np` and a seed has been set. Can you roll some dice?

#### Instructions

- Use `randint()` with the appropriate arguments to randomly generate the integer 1, 2, 3, 4, 5 or 6. This simulates a dice. Print it out.
- Repeat the outcome to see if the second throw is different. Again, print out the result.

```
[65]: # Import numpy and set seed
import numpy as np
np.random.seed(123)

Use randint() to simulate a dice
dice = np.random.randint(1, 7)
print(dice)
Use randint() again
dice = np.random.randint(1, 7)
print(dice)
```

6  
3

### 2.1.3 5.3 Determine your next move

In the Empire State Building bet, your next move depends on the number of eyes you throw with the dice. We can perfectly code this with an `if-elif-else` construct!

The sample code assumes that you're currently at step 50. Can you fill in the missing pieces to finish the script? `numpy` is already imported as `np` and the seed has been set to 123, so you don't have to worry about that anymore.

#### Instructions

- Roll the dice. Use `randint()` to create the variable `dice`.
- Finish the `if-elif-else` construct by replacing `__`:
- If `dice` is 1 or 2, you go one step down.
- If `dice` is 3, 4 or 5, you go one step up.
- Else, you throw the `dice` again. The number of eyes is the number of `steps` you go up.
- Print out `dice` and `step`. Given the value of `dice`, was `step` updated correctly?

```
[66]: # Import numpy and set seed
import numpy as np
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 <= 5 :
 step = step + 1
else :
 step = step + np.random.randint(1,7)

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

```

6  
53

#### 2.1.4 5.4 The next step

Before, you have already written Python code that determines the next step based on the previous step. Now it's time to put this code inside a `for` loop so that we can simulate a random walk.

Instructions

- Make a list `random_walk` that contains the first step, which is the integer 0.
- Finish the `for` loop:
- The loop should run 100 times.
- On each iteration, set `step` equal to the last element in the `random_walk` list. You can use the index `-1` for this.
- Next, let the `if-elif-else` construct update `step` for you.
- The code that appends `step` to `random_walk` is already coded.
- Print out `random_walk`.

```

[67]: # Import numpy and set seed
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, 9, 10, 11, 10, 14, 15, 14, 15, 14, 15, 16, 17, 18, 19, 20, 21,
24, 25, 26, 27, 32, 33, 37, 38, 37, 38, 39, 38, 39, 40, 42, 43, 44, 43, 42, 43,
44, 43, 42, 43, 44, 46, 45, 44, 45, 44, 45, 46, 47, 49, 48, 49, 50, 51, 52, 53,
52, 51, 52, 51, 52, 53, 52, 55, 56, 57, 58, 57, 58, 59]

```

*Good job! There's still something wrong: the level at index 15 is negative!*

### 2.1.5 5.5 How low can you go?

Things are shaping up nicely! You already have code that calculates your location in the Empire State Building after 100 dice throws. However, there's something we haven't thought about - you can't go below 0!

A typical way to solve problems like this is by using `max()`. If you pass `max()` two arguments, the biggest one gets returned. For example, to make sure that a variable `x` never goes below 10 when you decrease it, you can use: `~~~ x = max(10, x - 1) ~~~` **Instructions**

- Use `max()` in a similar way to make sure that `step` doesn't go below zero if `dice <= 2`.
- Hit Submit Answer and check the contents of `random_walk`.

```

[68]: # Import numpy and set seed
import numpy as np
np.random.seed(123)

Initialize random_walk
random_walk = [0]

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

```

```

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)

random_walk.append(step)

print(random_walk)

```

```

[0, 3, 4, 5, 4, 5, 6, 7, 6, 5, 4, 3, 2, 1, 0, 0, 1, 6, 5, 4, 5, 4, 5, 6, 7, 8,
9, 8, 9, 8, 9, 10, 11, 12, 11, 15, 16, 15, 16, 15, 16, 17, 18, 19, 20, 21, 22,
25, 26, 27, 28, 33, 34, 38, 39, 38, 39, 40, 39, 40, 41, 43, 44, 45, 44, 43, 44,
45, 44, 43, 44, 45, 47, 46, 45, 46, 45, 46, 47, 48, 50, 49, 50, 51, 52, 53, 54,
53, 52, 53, 52, 53, 54, 53, 56, 57, 58, 59, 58, 59, 60]

```

If you look closely at the output, you'll see that around index 15 the step stays at 0. You're not going below zero anymore. Great!

### 2.1.6 5.6 Visualize the walk

Let's visualize this random walk! Remember how you could use `matplotlib` to build a line plot? ~~~ `import matplotlib.pyplot as plt` `plt.plot(x, y)` `plt.show()` ~~~ The first list you pass is mapped onto the `x` axis and the second list is mapped onto the `y` axis.

If you pass only one argument, Python will know what to do and will use the index of the list to map onto the `x` axis, and the values in the list onto the `y` axis.

#### Instructions

Add some lines of code after the `for` loop: - Import `matplotlib.pyplot` as `plt`. - Use `plt.plot()` to plot `random_walk`. - Finish off with `plt.show()` to actually display the plot.

```

[69]: # Initialization
import numpy as np
np.random.seed(123)
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)

```

```

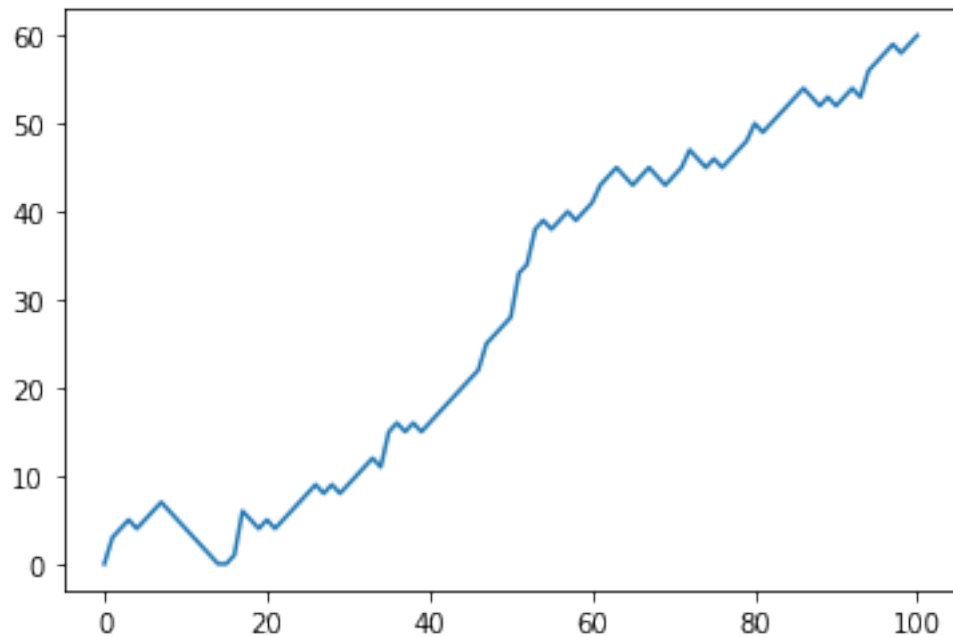
 random_walk.append(step)

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

Plot random_walk
plt.plot(random_walk)

Show the plot
plt.show()

```



*This is pretty cool! You can clearly see how your random walk progressed.*

### 2.1.7 5.7 Simulate multiple walks

A single random walk is one thing, but that doesn't tell you if you have a good chance at winning the bet.

To get an idea about how big your chances are of reaching 60 steps, you can repeatedly simulate the random walk and collect the results. That's exactly what you'll do in this exercise.

The sample code already sets you off in the right direction. Another `for` loop is wrapped around the code you already wrote. It's up to you to add some bits and pieces to make sure all of the results are recorded correctly.

**Note:** Don't change anything about the initialization of `all_walks` that is given. Setting any number inside the list will cause the exercise to crash!

## Instructions

- Fill in the specification of the `for` loop so that the random walk is simulated 10 times.
- After the `random_walk` array is entirely populated, append the array to the `all_walks` list.
- Finally, after the top-level `for` loop, print out `all_walks`.

```
[70]: # Initialization
import numpy as np
np.random.seed(123)

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, 3, 4, 5, 4, 5, 6, 7, 6, 5, 4, 3, 2, 1, 0, 0, 1, 6, 5, 4, 5, 4, 5, 6, 7, 8,
9, 8, 9, 8, 9, 10, 11, 12, 11, 15, 16, 15, 16, 15, 16, 17, 18, 19, 20, 21, 22,
25, 26, 27, 28, 33, 34, 38, 39, 38, 39, 40, 39, 40, 41, 43, 44, 45, 44, 43, 44,
45, 44, 43, 44, 45, 47, 46, 45, 46, 45, 46, 47, 48, 50, 49, 50, 51, 52, 53, 54,
53, 52, 53, 52, 53, 54, 53, 56, 57, 58, 59, 58, 59, 60], [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, 19, 20,
21, 22, 28, 27, 26, 25, 26, 27, 28, 27, 28, 29, 28, 33, 34, 33, 32, 31, 30, 31,
30, 29, 31, 32, 35, 36, 38, 39, 40, 41, 40, 39, 40, 41, 42, 43, 42, 43, 44, 45,
48, 49, 50, 49, 50, 49, 50, 51, 52, 56, 55, 54, 55, 56, 57, 56, 57, 56, 57, 59,
64, 63, 64, 65, 66, 67, 68, 69, 68, 69, 70, 71, 73], [0, 2, 1, 2, 3, 6, 5, 6, 5,
6, 7, 8, 7, 8, 7, 8, 9, 11, 10, 9, 10, 11, 10, 12, 13, 14, 15, 16, 17, 18, 17,
18, 19, 24, 25, 24, 23, 22, 21, 22, 23, 24, 29, 30, 29, 30, 31, 32, 33, 34, 35,
```

```

34, 33, 34, 33, 39, 38, 39, 38, 39, 38, 39, 43, 47, 49, 51, 50, 51, 53, 52, 58,
59, 61, 62, 61, 62, 63, 64, 63, 64, 65, 66, 68, 67, 66, 67, 73, 78, 77, 76, 80,
81, 82, 83, 85, 84, 85, 84, 85, 84, 83], [0, 6, 5, 6, 7, 8, 9, 10, 11, 12, 13,
12, 13, 12, 11, 12, 11, 12, 11, 12, 13, 17, 18, 17, 23, 22, 21, 22, 21, 20, 21,
20, 24, 23, 24, 23, 24, 23, 24, 26, 25, 24, 23, 24, 23, 28, 29, 30, 29, 28, 29,
28, 29, 28, 33, 34, 33, 32, 31, 30, 31, 32, 36, 42, 43, 44, 45, 46, 45, 46, 48,
49, 50, 51, 50, 49, 50, 49, 50, 51, 52, 51, 52, 53, 54, 53, 52, 53, 54, 59, 60,
61, 66, 65, 66, 65, 66, 67, 68, 69, 68], [0, 6, 5, 6, 5, 4, 5, 9, 10, 11, 12,
13, 12, 11, 10, 9, 8, 9, 10, 11, 12, 13, 14, 13, 14, 15, 14, 15, 16, 19, 18, 19,
18, 19, 22, 23, 24, 25, 24, 23, 26, 27, 28, 29, 28, 27, 28, 31, 32, 37, 38, 37,
38, 37, 38, 37, 43, 42, 41, 42, 44, 43, 42, 41, 42, 43, 44, 45, 49, 54, 55, 56,
57, 60, 61, 62, 63, 64, 65, 66, 65, 64, 65, 66, 65, 71, 70, 71, 72, 71, 70, 71,
70, 69, 75, 74, 73, 74, 75, 74, 73], [0, 0, 0, 1, 7, 8, 11, 12, 18, 19, 20, 26,
25, 31, 30, 31, 32, 33, 32, 38, 39, 38, 39, 38, 39, 38, 39, 38, 39, 43, 44, 46,
45, 46, 45, 44, 45, 44, 45, 44, 48, 52, 51, 50, 49, 50, 51, 55, 56, 57, 61, 60,
59, 58, 59, 60, 62, 61, 60, 61, 62, 64, 67, 72, 73, 72, 73, 74, 75, 76, 77, 76,
77, 78, 84, 83, 88, 87, 91, 90, 94, 93, 96, 97, 96, 97, 103, 102, 101, 100, 104,
103, 102, 103, 104, 103, 104, 105, 106, 107, 106], [0, 0, 0, 1, 0, 0, 4, 5, 7,
11, 17, 16, 15, 16, 17, 18, 17, 18, 17, 18, 19, 18, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 33, 32, 35, 36, 35, 34, 35, 36, 37, 36, 35, 34, 33, 34, 35, 36,
37, 38, 39, 40, 39, 40, 41, 43, 42, 43, 44, 47, 49, 50, 49, 48, 47, 46, 45, 46,
45, 46, 48, 49, 50, 49, 50, 49, 48, 49, 48, 47, 46, 47, 46, 45, 46, 47, 48, 50,
51, 52, 51, 50, 51, 57, 56, 57, 58, 63, 62, 63], [0, 0, 1, 2, 1, 2, 3, 9, 10,
11, 12, 11, 13, 14, 15, 16, 15, 16, 17, 18, 19, 18, 19, 18, 19, 20, 19, 20, 24,
25, 28, 29, 33, 34, 33, 34, 35, 34, 33, 38, 39, 40, 39, 38, 39, 40, 41, 40, 44,
43, 44, 45, 46, 47, 48, 49, 50, 49, 48, 47, 48, 49, 53, 54, 53, 54, 55, 54, 60,
61, 62, 63, 62, 63, 64, 67, 66, 67, 66, 65, 64, 65, 66, 68, 69, 70, 74, 75, 74,
73, 74, 75, 74, 73, 74, 75, 76, 75, 74, 75, 76], [0, 1, 0, 1, 2, 1, 0, 0, 1, 2,
3, 4, 5, 10, 14, 13, 14, 13, 12, 11, 12, 11, 12, 13, 12, 16, 17, 16, 17, 16, 15,
16, 15, 19, 20, 21, 22, 23, 24, 23, 24, 25, 26, 27, 28, 27, 32, 33, 34, 33, 34,
33, 34, 35, 34, 35, 40, 41, 42, 41, 42, 43, 44, 43, 44, 43, 44, 45, 44, 43, 42,
43, 44, 43, 42, 41, 42, 46, 47, 48, 49, 50, 51, 50, 51, 52, 51, 52, 57, 58, 57,
56, 57, 56, 55, 54, 58, 59, 60, 61, 60], [0, 1, 2, 3, 4, 5, 4, 3, 6, 5, 4, 3, 2,
3, 9, 10, 9, 10, 11, 10, 9, 10, 11, 12, 11, 15, 16, 15, 17, 18, 17, 18, 19, 20,
21, 22, 23, 22, 21, 22, 23, 22, 23, 24, 23, 22, 21, 25, 26, 27, 28, 29, 30, 31,
32, 33, 34, 33, 34, 35, 36, 37, 38, 37, 36, 42, 43, 44, 43, 42, 41, 45, 46, 50,
49, 55, 56, 57, 61, 62, 61, 60, 61, 62, 63, 64, 63, 69, 70, 69, 73, 74, 73, 74,
73, 79, 85, 86, 85, 86, 87]]

```

### 2.1.8 5.8 Visualize all walks

`all_walks` is a list of lists: every sub-list represents a single random walk. If you convert this list of lists to a Numpy array, you can start making interesting plots! `matplotlib.pyplot` is already imported as `plt`.

The nested `for` loop is already coded for you - don't worry about it. For now, focus on the code that comes after this `for` loop.

### Instructions

- Use `np.array()` to convert `all_walks` to a Numpy array, `np_aw`.
- Try to use `plt.plot()` on `np_aw`. Also include `plt.show()`. Does it work out of the box?
- Transpose `np_aw` by calling `np.transpose()` on `np_aw`. Call the result `np_aw_t`. Now every row in `np_all_walks` represents the position after 1 throw for the 10 random walks.
- Use `plt.plot()` to plot `np_aw_t`; also include a `plt.show()`. Does it look better this time?

```
[71]: import matplotlib.pyplot as plt
import numpy as np
np.random.seed(123)
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)

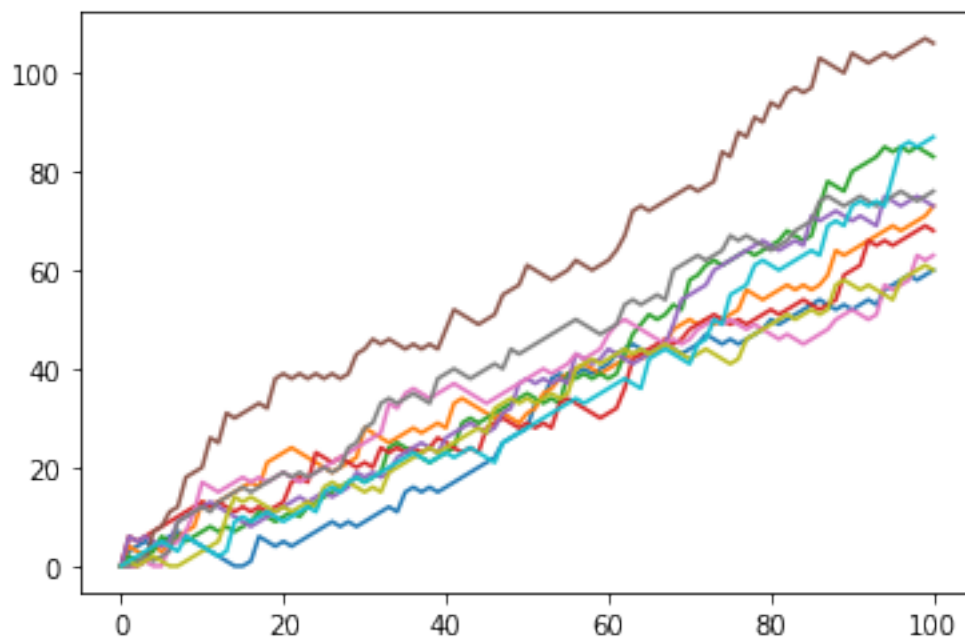
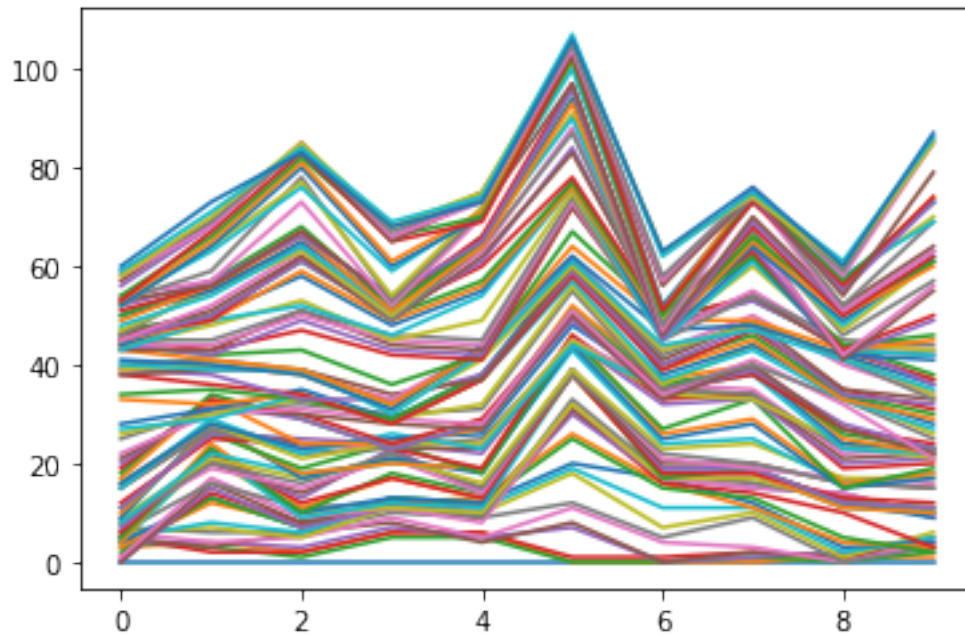
Plot np_aw and show
plt.plot(np_aw)
plt.show()

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()
```





### 2.1.9 5.9 Implement clumsiness

With this neatly written code of yours, changing the number of times the random walk should be simulated is super-easy. You simply update the `range()` function in the top-level `for` loop.

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.

### Instructions

- Change the `range()` function so that the simulation is performed 250 times.
- Finish the `if` condition so that `step` is set to 0 if a random float is less or equal to 0.001. Use `np.random.rand()`.

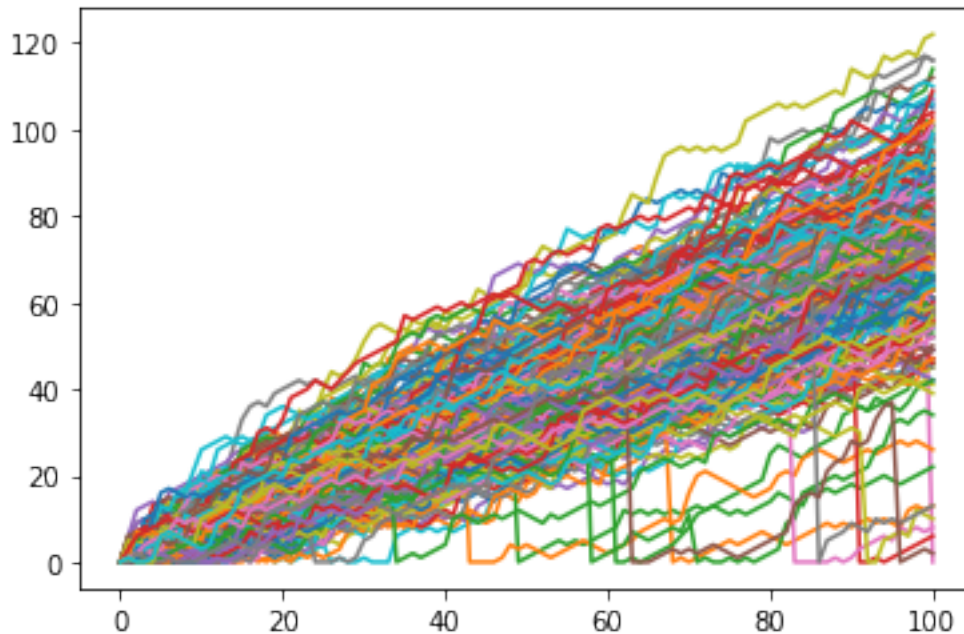
```
[72]: import matplotlib.pyplot as plt
import numpy as np
np.random.seed(123)
all_walks = []

Simulate random walk 250 times
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()
```



### 2.1.10 5.10 Plot the distribution

All these fancy visualizations have put us on a sidetrack. We still have to solve the million-dollar problem: What are the odds that you'll reach 60 steps high on the Empire State Building?

Basically, you want to know about the end points of all the random walks you've simulated. These end points have a certain distribution that you can visualize with a histogram.

Note that if your code is taking too long to run, you might be plotting a histogram of the wrong data!

#### Instructions

- To make sure we've got enough simulations, go crazy. Simulate the random walk 500 times.
- From `np_aw_t`, select the last row. This contains the endpoint of all 500 random walks you've simulated. Store this Numpy array as `ends`.
- Use `plt.hist()` to build a histogram of `ends`. Don't forget `plt.show()` to display the plot.

```
[73]: import matplotlib.pyplot as plt
import numpy as np
np.random.seed(123)
all_walks = []

Simulate random walk 500 times
for i in range(500):
 random_walk = [0]
 for x in range(100):
 step = random_walk[-1]
```

```

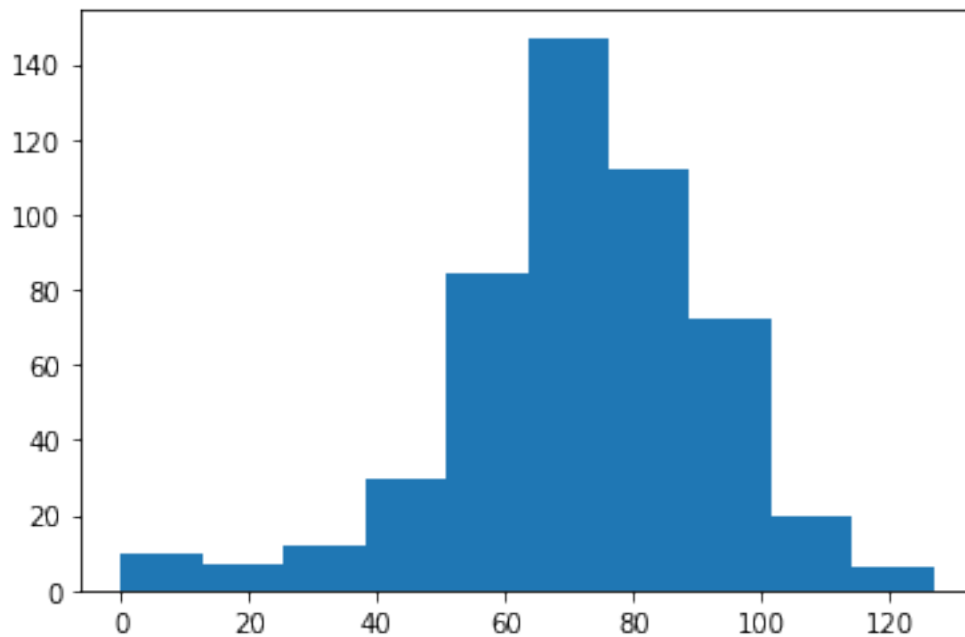
dice = np.random.randint(1,7)
if dice <= 2:
 step = max(0, step - 1)
elif dice <= 5:
 step = step + 1
else:
 step = step + np.random.randint(1,7)
if np.random.rand() <= 0.001 :
 step = 0
random_walk.append(step)
all_walks.append(random_walk)

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()

```



### 2.1.11 5.11 Calculate the odds

The histogram of the previous exercise 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.

Well then, what's the estimated chance that you'll reach 60 steps high if you play this Empire State Building game? The `ends` array is everything you need; it's available in your Python session so you can make calculations in the IPython Shell.

### Instructions

**Possible Answers** - 48.8% - 73.9% - **78.4%** - 95.9%