Chapter 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 will learn to build various types of plots and to customize them to make them more visually appealing and interpretable.

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

In [ ]:
```
year = [x for x in range(1950, 2101)]
print(year)
```

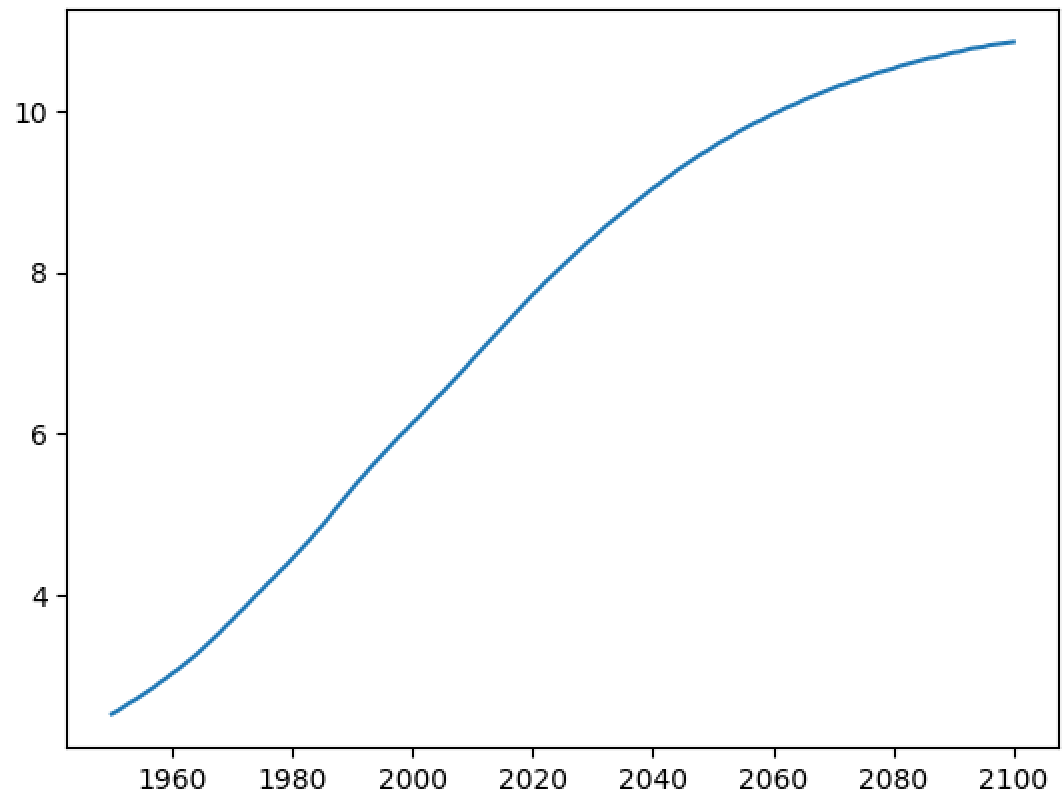
```
[1950, 1951, 1952, 1953, 1954, 1955, 1956, 1957, 1958, 1959, 1960, 1961, 1962, 1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970, 19
71, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988, 1989, 1990, 1991, 1992,
1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 201
4, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023, 2024, 2025, 2026, 2027, 2028, 2029, 2030, 2031, 2032, 2033, 2034, 2035,
2036, 2037, 2038, 2039, 2040, 2041, 2042, 2043, 2044, 2045, 2046, 2047, 2048, 2049, 2050, 2051, 2052, 2053, 2054, 2055, 2056, 205
7, 2058, 2059, 2060, 2061, 2062, 2063, 2064, 2065, 2066, 2067, 2068, 2069, 2070, 2071, 2072, 2073, 2074, 2075, 2076, 2077, 2078,
2079, 2080, 2081, 2082, 2083, 2084, 2085, 2086, 2087, 2088, 2089, 2090, 2091, 2092, 2093, 2094, 2095, 2096, 2097, 2098, 2099, 210
0]
```

In [ ]:
```
pop = [2.53, 2.57, 2.62, 2.67, 2.71, 2.76, 2.81, 2.86, 2.92, 2.97, 3.03, 3.08, 3.14, 3.2, 3.26, 3.33, 3.4, 3.47, 3.54, 3.62, 3.69,
```

In [ ]:
```
# 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)
```

```python
# Display the plot with plt.show()
plt.show() # no need to use plt.show() in jupyter notebook
```

```
2100
10.85
```



Line plot (2)

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.

In [ ]:
```python
gdp_cap = [974.5803384, 5937.029525999998, 6223.367465, 4797.231267, 12779.37964, 34435.367439999995, 36126.4927, 29796.04834, 13
```

In [ ]:
```python
life_exp = [43.828, 76.423, 72.301, 42.731, 75.32, 81.235, 79.829, 75.635, 64.062, 79.441, 56.728, 65.554, 74.852, 50.728, 72.39,
```

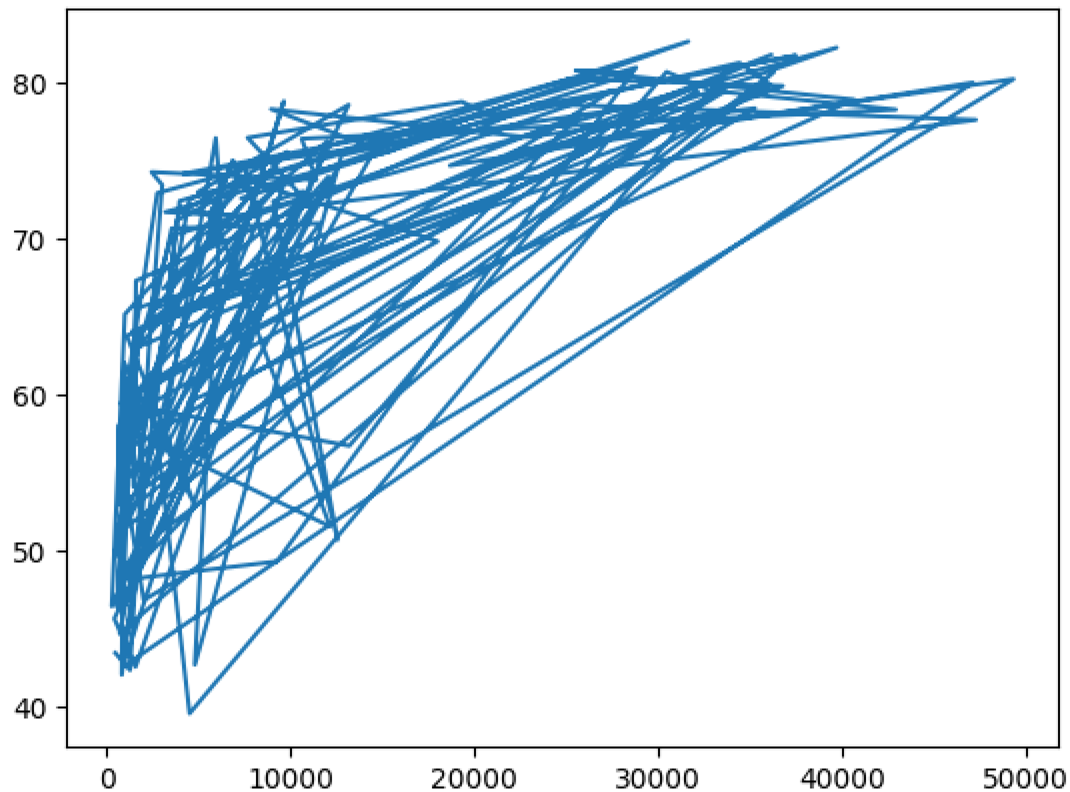
In [ ]:

```python
# 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)
```

```
469.70929810000007
43.487
```

Out[ ]:

```
[<matplotlib.lines.Line2D at 0x1b755ce3830>]
```



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

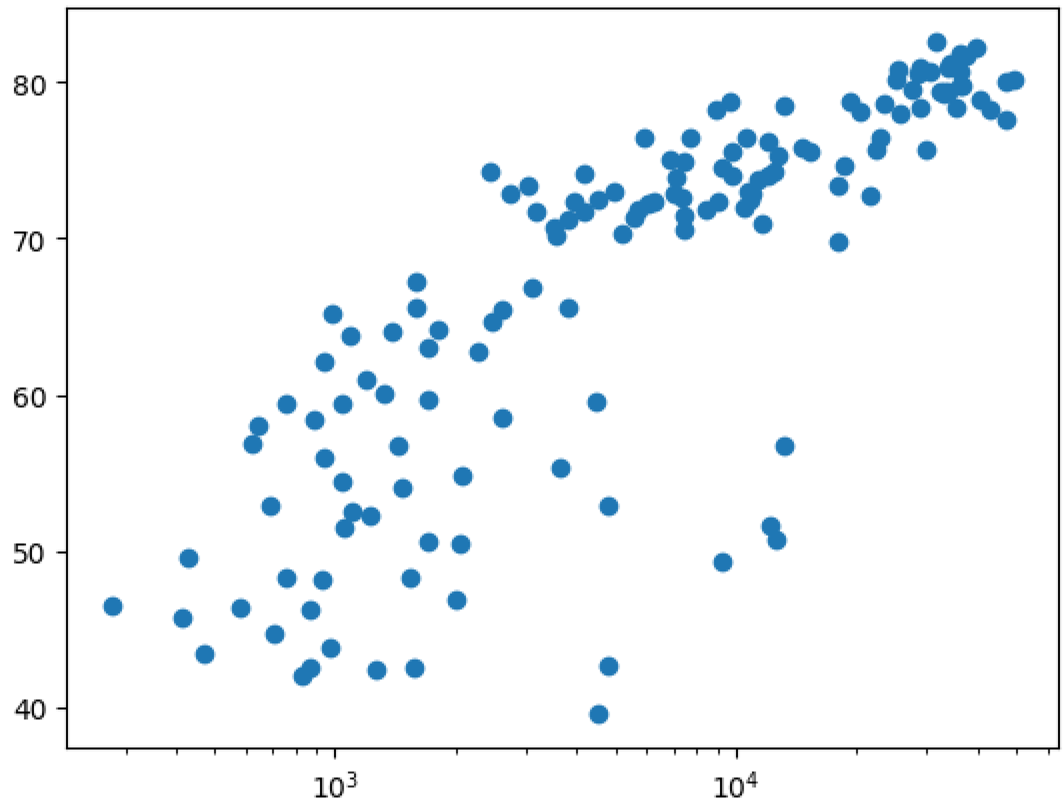
Scatter Plot (1)

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.

```python
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?

In [ ]:
```
# 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')
```



Great! That looks much better!

Scatter plot (2)

In the previous exercise, you saw that 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_2007 is available, listing the corresponding populations for the countries in 2007. The populations are in millions of people.

In [ ]:
```
len(life_exp)
```

Out[ ]:
```
142
```

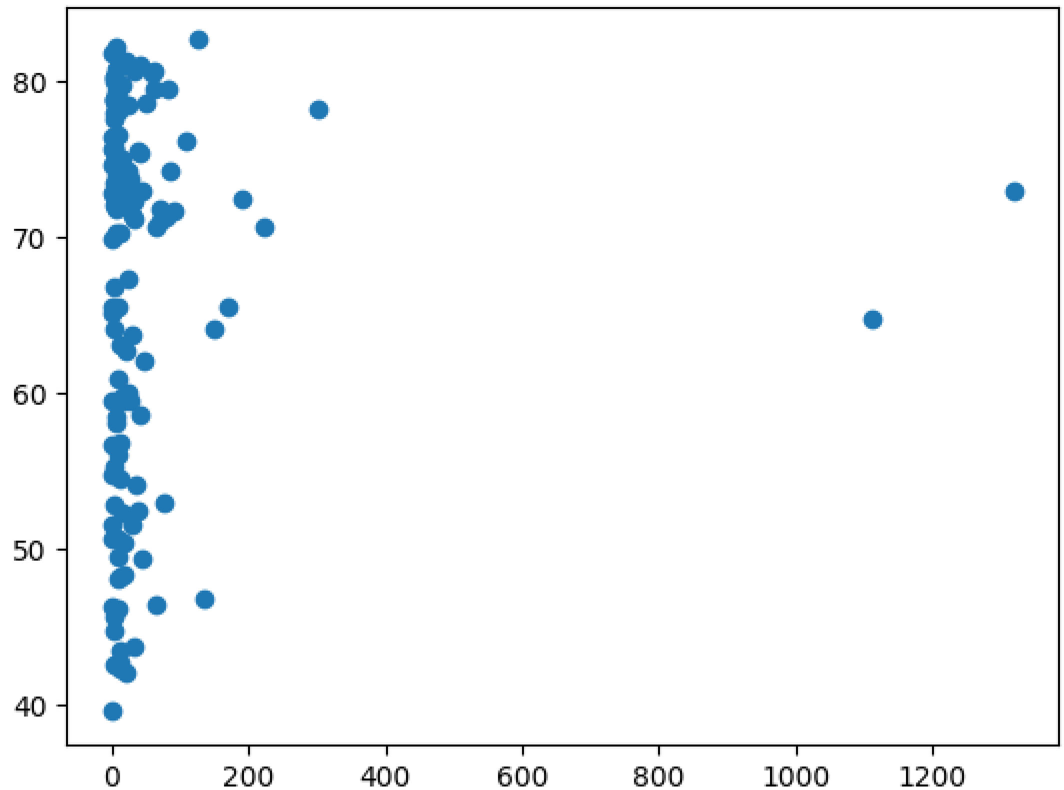
In [ ]:
```
pop_2007 = [31.889923, 3.600523, 33.333216, 12.420476, 40.301927, 20.434176, 8.199783, 0.708573, 150.448339, 10.392226, 8.078314,
```

In [ ]:
```
len(pop_2007)
```

Out[ ]:
```
142
```

In [ ]:
```
# Build Scatter plot
plt.scatter(pop_2007, life_exp)
```

Out[ ]:
```
<matplotlib.collections.PathCollection at 0x1b759069220>
```



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

Build a histogram (1)

life_exp, the list containing data on the life expectancy for different countries in 2007, is available

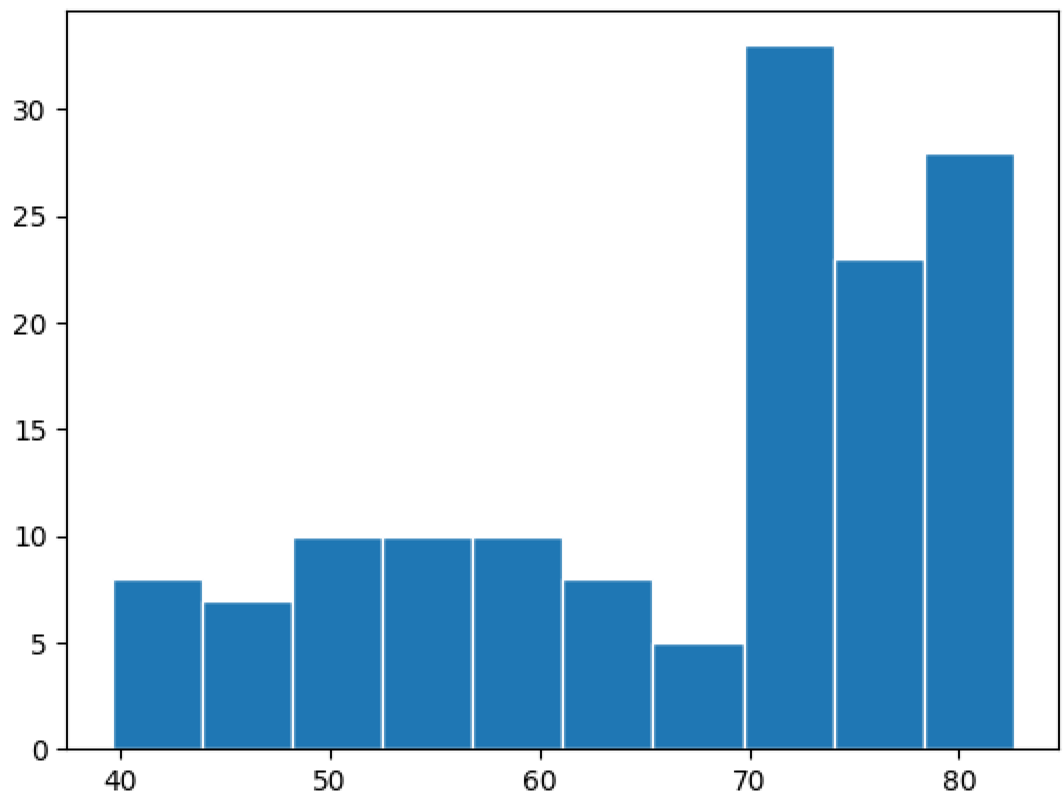
To see how life expectancy in different countries is distributed, let's create a histogram of life_exp.

In [ ]:

```
# Create histogram of life_exp data
plt.hist(life_exp, ec='white')
```

Out[ ]:

```
(array([ 8.,  7., 10., 10., 10.,  8.,  5., 33., 23., 28.]),
 array([39.613, 43.912, 48.211, 52.51 , 56.809, 61.108, 65.407, 69.706,
        74.005, 78.304, 82.603]),
 <BarContainer object of 10 artists>)
```

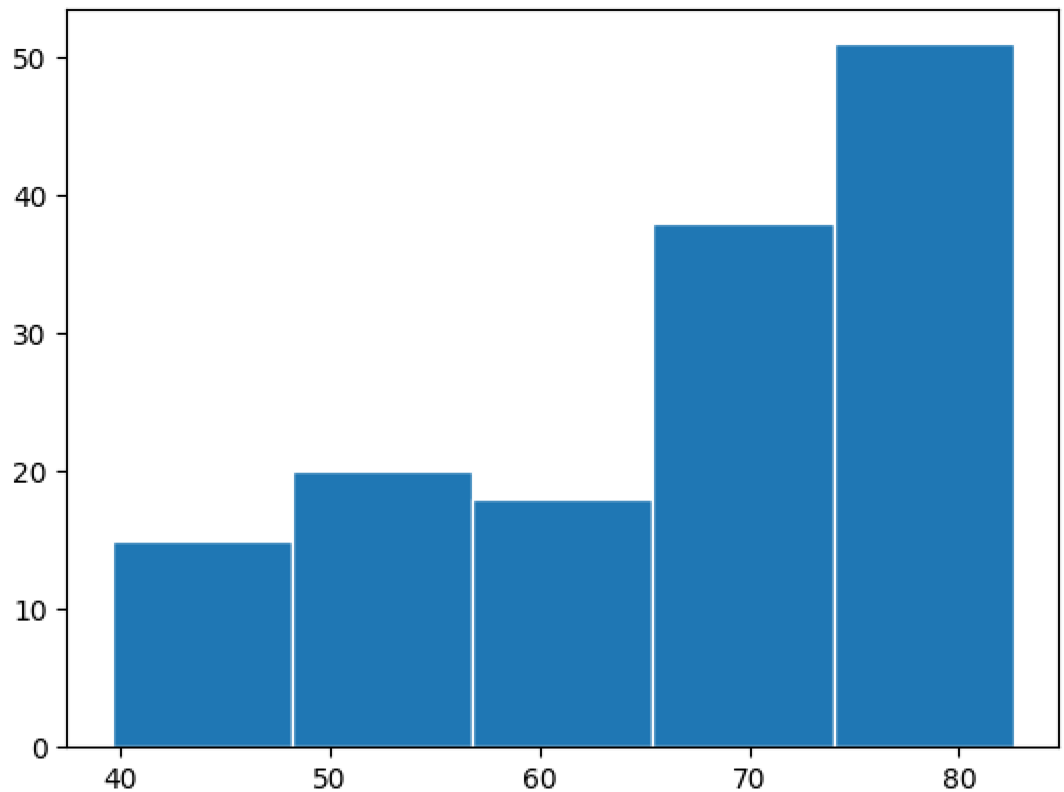


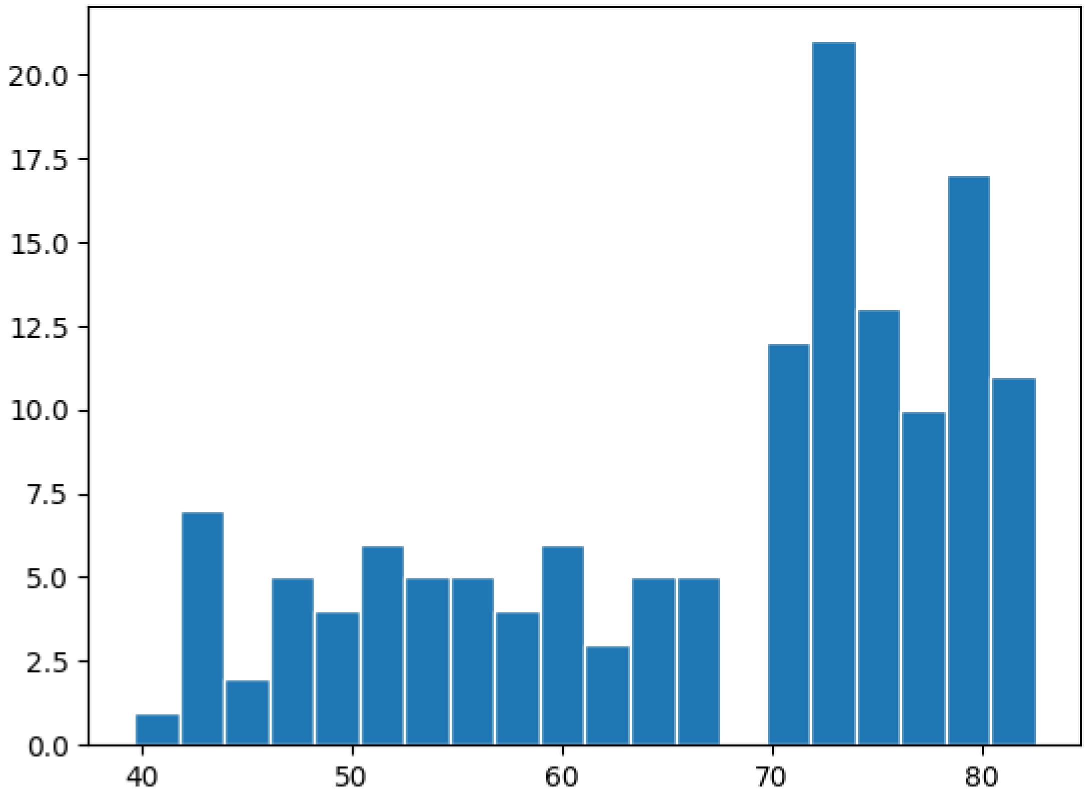
Build a histogram (2): bins

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.

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

# Build histogram with 20 bins
plt.hist(life_exp, bins = 20, ec='white')
plt.show()
```





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.

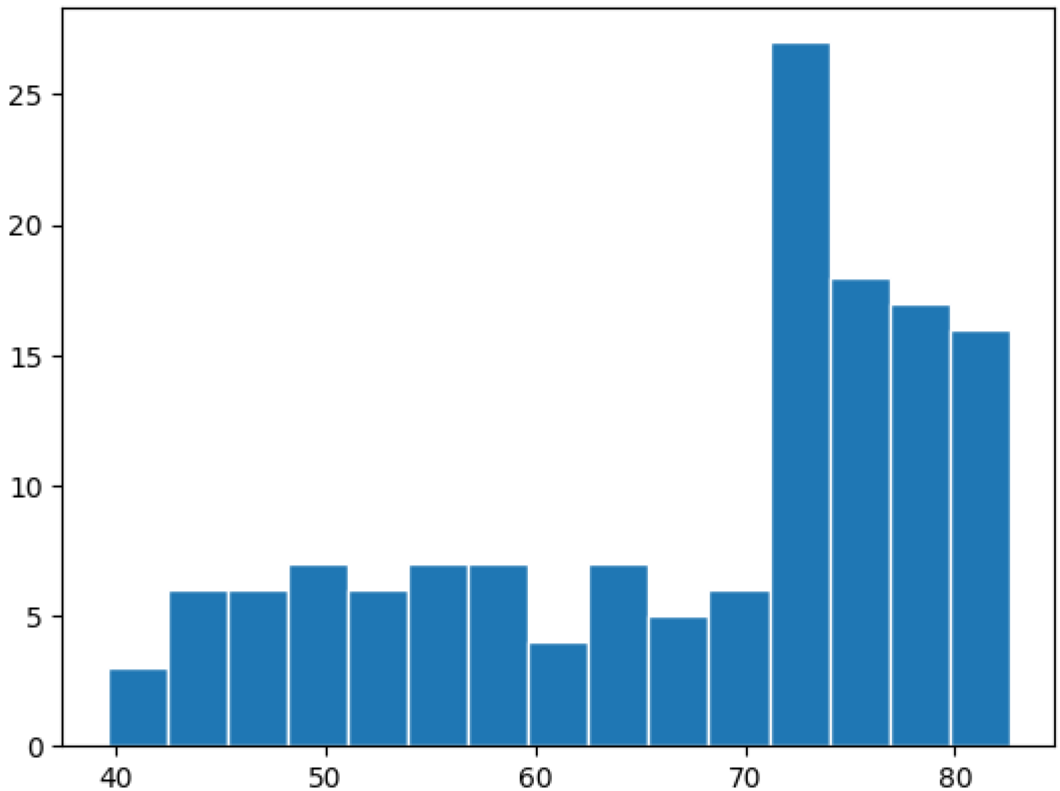
Let's do a similar comparison. life_exp contains life expectancy data for different countries in 2007. You also have access to a second list now, life_exp1950, containing similar data for 1950. Can you make a histogram for both datasets?

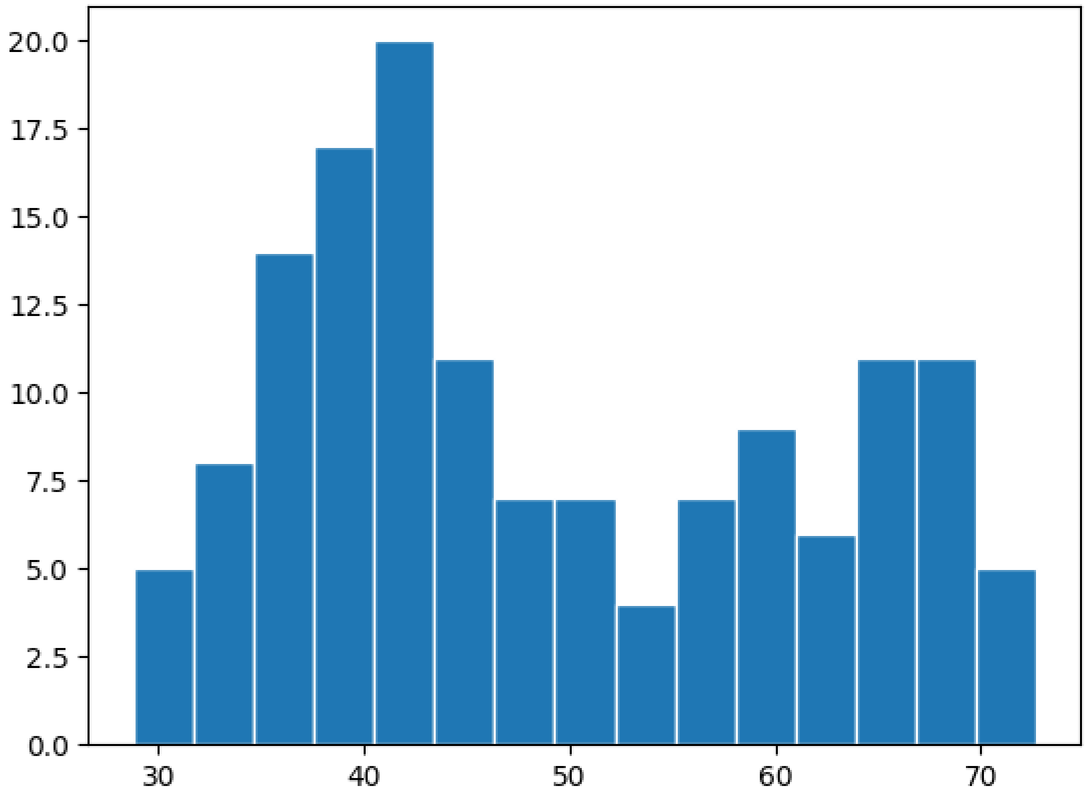
You'll again be making two plots. The plt.show() and plt.clf() commands to render everything nicely are already included.

In [ ]:
```
life_exp1950 = [28.8, 55.23, 43.08, 30.02, 62.48, 69.12, 66.8, 50.94, 37.48, 68.0, 38.22, 40.41, 53.82, 47.62, 50.92, 59.6, 31.98,
```

In [ ]:
```
# Histogram of life_exp, 15 bins
plt.hist(life_exp, bins = 15, ec='white')

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

# Histogram of life_exp1950, 15 bins
```

```
plt.hist(life_exp1950, bins = 15, ec='white')
# Show and clear plot again
plt.show()
plt.clf()
```






```
<Figure size 640x480 with 0 Axes>
```

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.

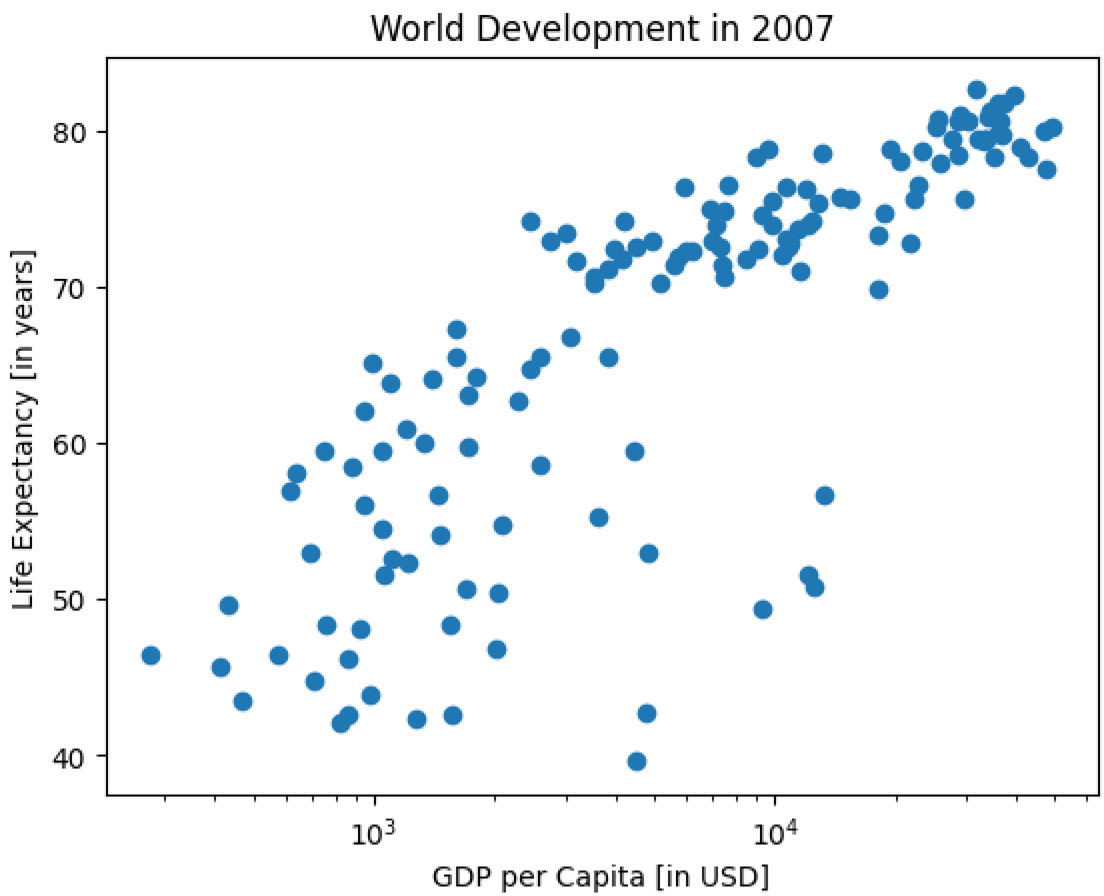
In [ ]:

```
# 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)
```

Out[ ]:

```
Text(0.5, 1.0, 'World Development in 2007')
```



Ticks

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.

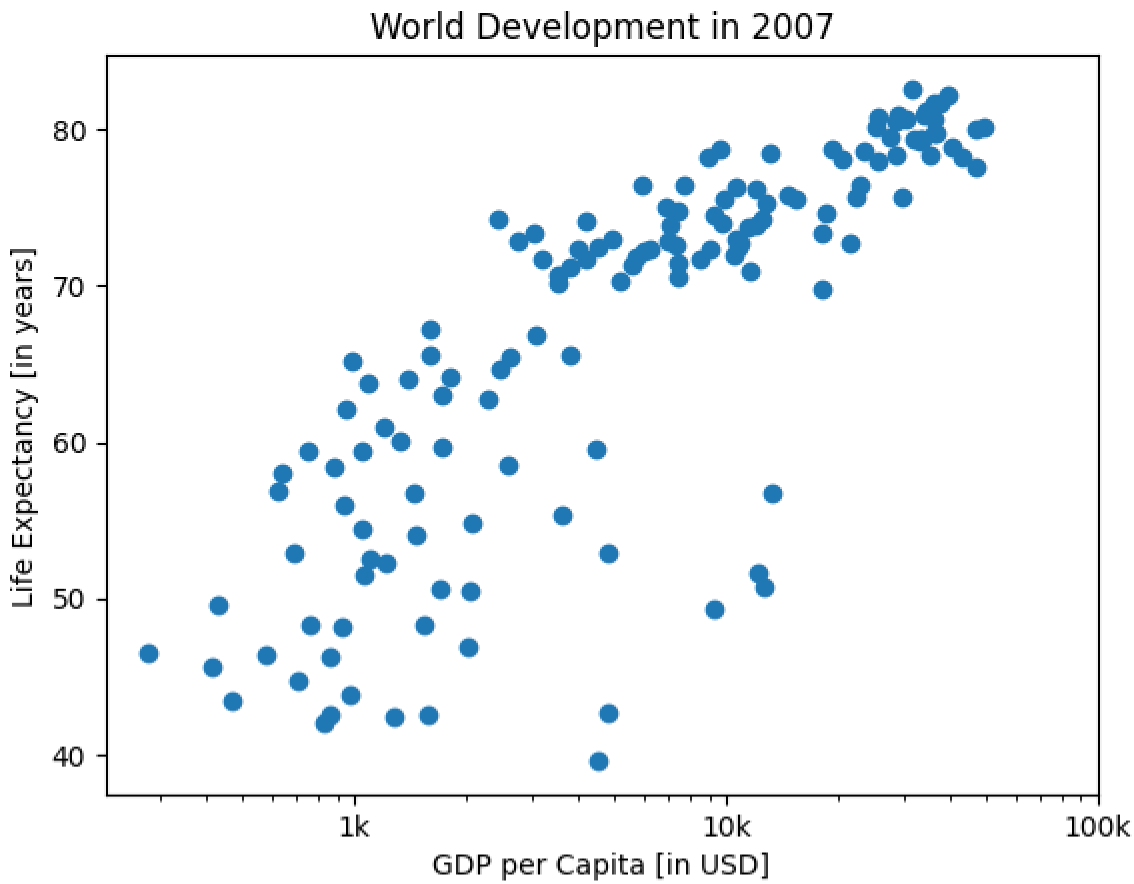
In [ ]:

```
# Scatter plot
plt.scatter(gdp_cap, life_exp)
```

```python
# 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)
```

Out[ ]:

```
([<matplotlib.axis.XTick at 0x1b7591128a0>,
  <matplotlib.axis.XTick at 0x1b75947b740>,
  <matplotlib.axis.XTick at 0x1b75947a960>],
 [Text(1000, 0, '1k'), Text(10000, 0, '10k'), Text(100000, 0, '100k')])
```

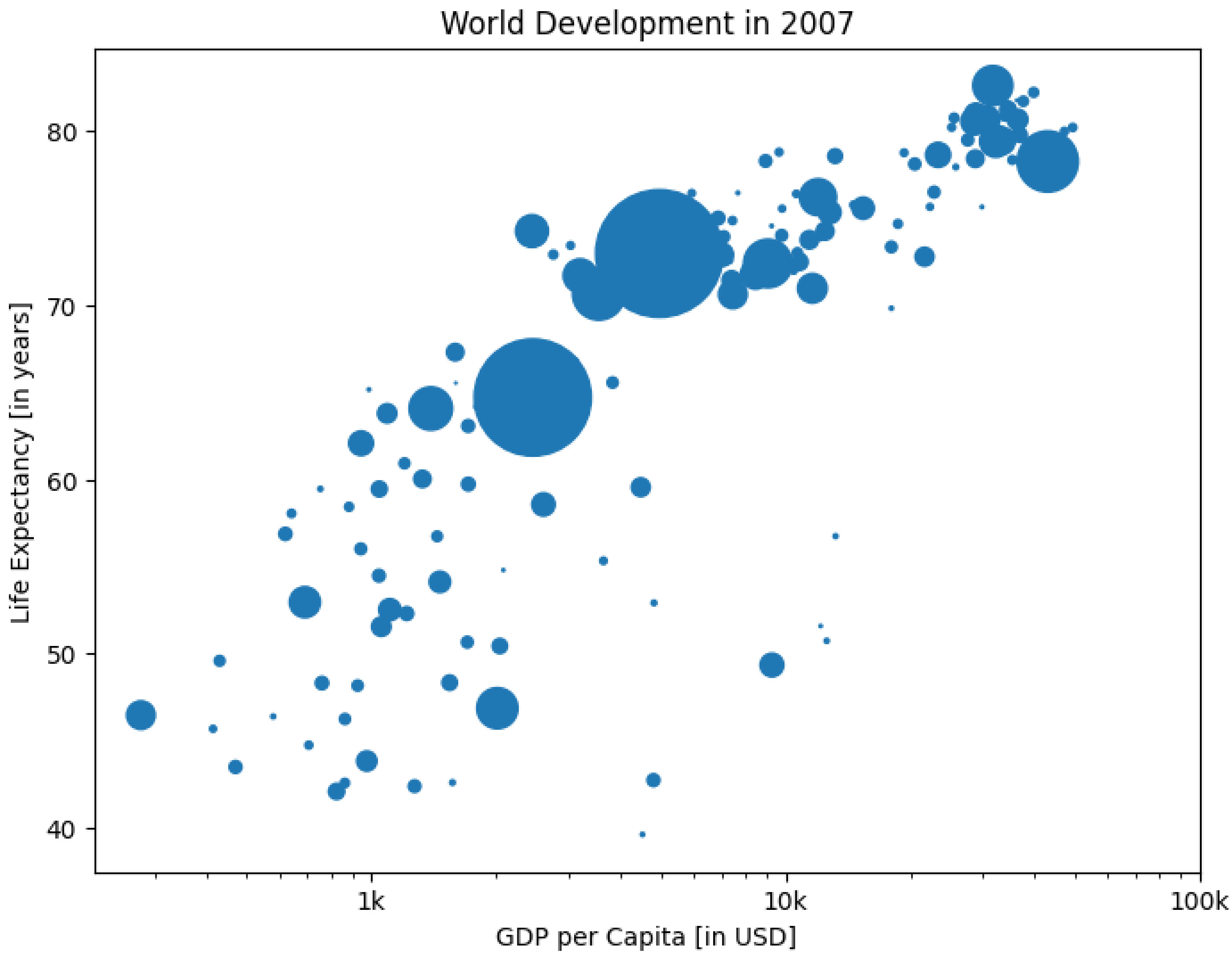


Great! Your plot is shaping up nicely!

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?

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

# Store pop_2017 as a numpy array: np_pop
np_pop = np.array(pop_2007)

# Double np_pop
np_pop = np_pop * 2

# Update: set s argument to np_pop
plt.figure(figsize = (8, 6))
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()
```



Bellissimo! Can you already tell which bubbles correspond to which countries?

Colors

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.

In [ ]:

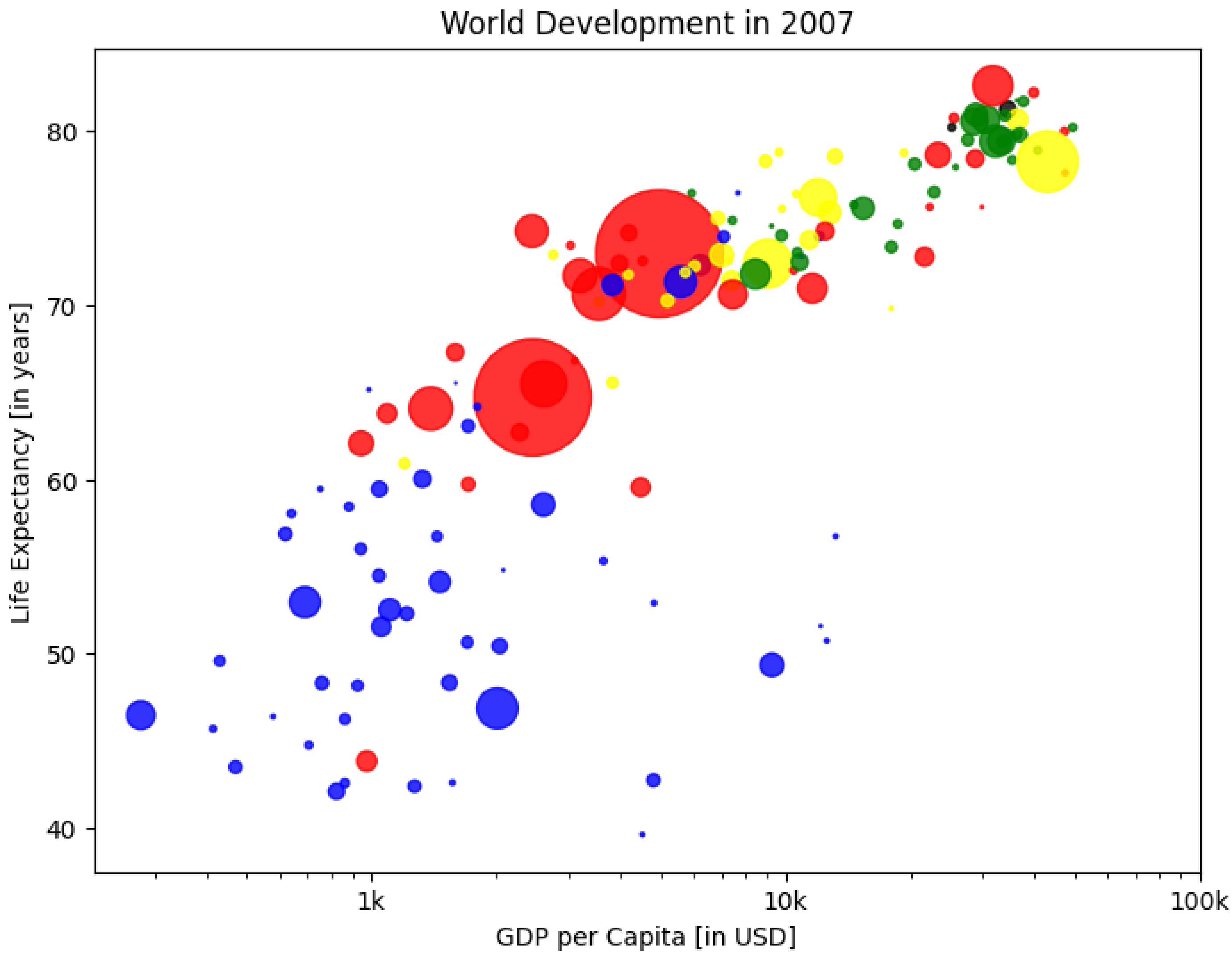
```
col = ['red', 'green', 'blue', 'blue', 'yellow', 'black', 'green', 'red', 'red', 'green', 'blue', 'yellow', 'green', 'blue', 'yell
```

In [ ]:

```
# Specify c and alpha inside plt.scatter()
plt.figure(figsize = (8, 6))
plt.scatter(x = gdp_cap, y = life_exp, s = np.array(pop_2007) * 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'])
```

Out[ ]:

```
([<matplotlib.axis.XTick at 0x1b7594328a0>,
  <matplotlib.axis.XTick at 0x1b758fa3dd0>,
  <matplotlib.axis.XTick at 0x1b755da54f0>],
 [Text(1000, 0, '1k'), Text(10000, 0, '10k'), Text(100000, 0, '100k')])
```



Nice! This is looking more and more like Hans Rosling's plot!

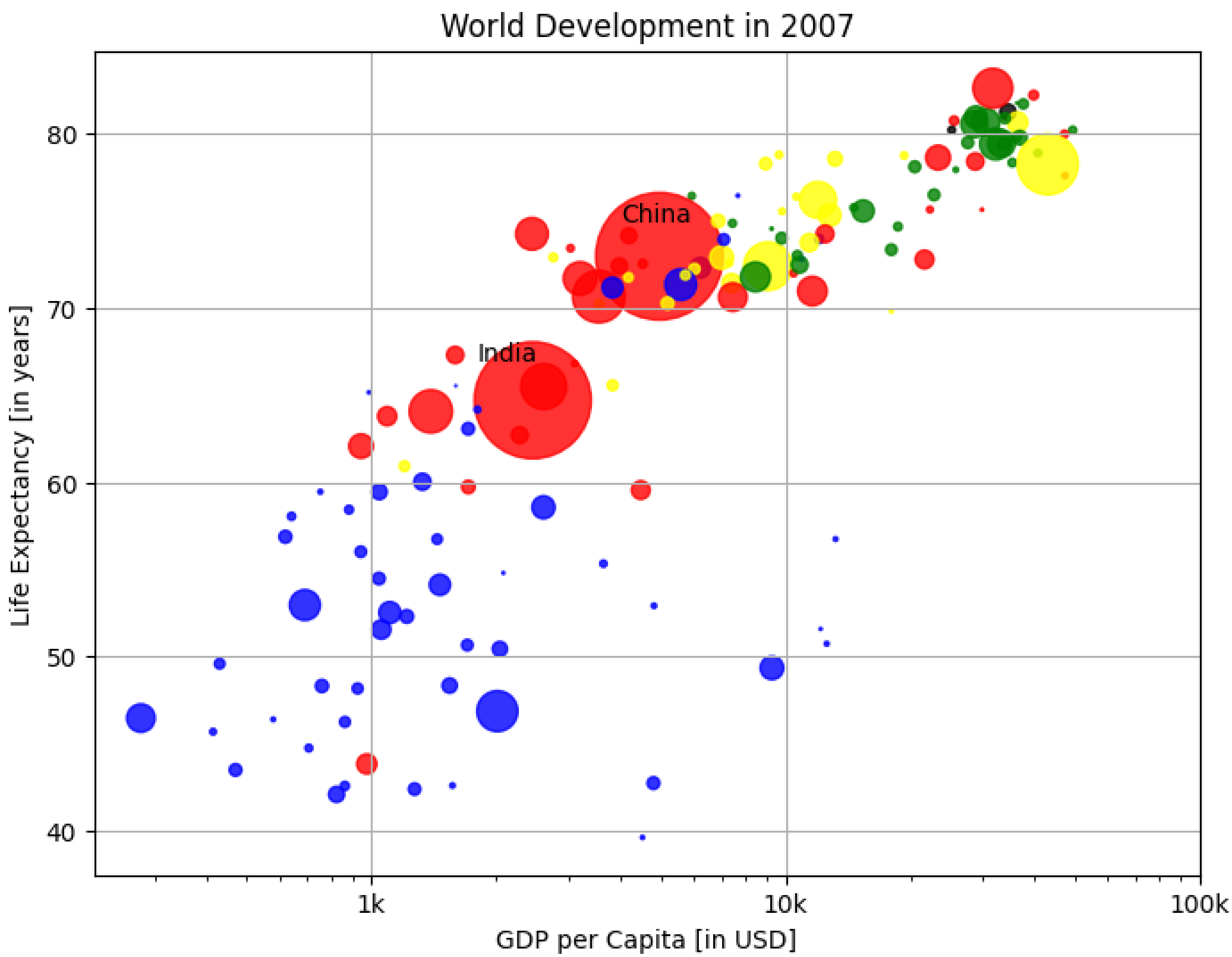
Additional Customizations

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

In [ ]:

```
# Scatter plot
plt.figure(figsize = (8, 6))
plt.scatter(x = gdp_cap, y = life_exp, s = np.array(pop_2007) * 2, c = col, alpha = 0.8)

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

```python
plt.ylabel('Life Expectancy [in years]')
plt.title('World Development in 2007')
plt.xticks([1000,10000,100000], ['1k','10k','100k'])

# Additional customizations
plt.text(1800, 67, 'India')
plt.text(4000, 75, 'China')

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



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

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?

The countries in blue, corresponding to Africa, have both low life expectancy and a low GDP per capita.

Correct! Up to the next chapter, on dictionaries!

Chapter 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, manipulating and accessing the information you need from these data structures.

Motivation for dictionaries

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

In [ ]:

```
# 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
```

it's not very convenient. Head over to the next exercise to create a dictionary of this data.

Create dictionary

The countries and capitals lists are again available. 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.

In [ ]:
```
# From string in countries and capitals, create dictionary europe
europe = { 'spain':'madrid', 'france':'paris', 'germany':'berlin', 'norway':'oslo' }

# Print europe
print(europe)
```

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

Great! Now that you've built your first dictionaries, let's get serious!

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.

In [ ]:
```
# 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
```

Good job, now you're warmed up for some more.

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'

In [ ]:
```
# 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.

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

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

In [ ]:

```
# 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'}
```

Great job! That's much better!

Dictionariception

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']

```
In [ ]: # 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': 5
9.83}}
```

Great! It's time to learn about a new data structure!

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.

In [ ]:
```
# 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
cars
```

Out[ ]:

	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!

Dictionary to DataFrame (2)

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

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

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

# Specify row labels of cars
cars.index = row_labels
```

```
# Print cars
cars
```

Out[ ]:

	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

Nice! That looks much better already!

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'.

In [ ]:

```
# Import the cars.csv data: cars
cars = pd.read_csv('cars.csv')

# Print out cars
cars
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
Cell In[35], line 2
      1 # Import the cars.csv data: cars
----> 2 cars = pd.read_csv('cars.csv')
      4 # Print out cars
      5 cars

File c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\io\parsers\readers.py:948, in read_csv(filep
ath_or_buffer, sep, delimiter, header, names, index_col, usecols, dtype, engine, converters, true_values, false_values, skipiniti
alspace, skiprows, skipfooter, nrows, na_values, keep_default_na, na_filter, verbose, skip_blank_lines, parse_dates, infer_dateti
me_format, keep_date_col, date_parser, date_format, dayfirst, cache_dates, iterator, chunksize, compression, thousands, decimal,
lineterminator, quotechar, quoting, doublequote, escapechar, comment, encoding, encoding_errors, dialect, on_bad_lines, delim_whi
tespace, low_memory, memory_map, float_precision, storage_options, dtype_backend)
    935 kwds_defaults = _refine_defaults_read(
    936     dialect,
    937     delimiter,
   (...)
    944     dtype_backend=dtype_backend,
    945 )
    946 kwds.update(kwds_defaults)
--> 948 return _read(filepath_or_buffer, kwds)

File c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\io\parsers\readers.py:611, in _read(filepath
_or_buffer, kwds)
    608 _validate_names(kwds.get("names", None))
    610 # Create the parser.
--> 611 parser = TextFileReader(filepath_or_buffer, **kwds)
    613 if chunksize or iterator:
    614     return parser

File c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\io\parsers\readers.py:1448, in TextFileReade
r.__init__(self, f, engine, **kwds)
   1445     self.options["has_index_names"] = kwds["has_index_names"]
   1447 self.handles: IOHandles | None = None
-> 1448 self._engine = self._make_engine(f, self.engine)

File c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\io\parsers\readers.py:1705, in TextFileReade
r._make_engine(self, f, engine)
   1703     if "b" not in mode:
   1704         mode += "b"
-> 1705 self.handles = get_handle(
   1706     f,
   1707     mode,
   1708     encoding=self.options.get("encoding", None),
   1709     compression=self.options.get("compression", None),
   1710     memory_map=self.options.get("memory_map", False),
   1711     is_text=is_text,
   1712     errors=self.options.get("encoding_errors", "strict"),
```

```
   1713     storage_options=self.options.get("storage_options", None),
   1714 )
   1715 assert self.handles is not None
   1716 f = self.handles.handle

File c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\io\common.py:863, in get_handle(path_or_buf, mode, encoding, compression, memory_map, is_text, errors, storage_options)
    858 elif isinstance(handle, str):
    859     # Check whether the filename is to be opened in binary mode.
    860     # Binary mode does not support 'encoding' and 'newline'.
    861     if ioargs.encoding and "b" not in ioargs.mode:
    862         # Encoding
--> 863         handle = open(
    864             handle,
    865             ioargs.mode,
    866             encoding=ioargs.encoding,
    867             errors=errors,
    868             newline="",
    869         )
    870     else:
    871         # Binary mode
    872         handle = open(handle, ioargs.mode)

FileNotFoundError: [Errno 2] No such file or directory: 'cars.csv'
```

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

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!

In [ ]:

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

# Print out cars
cars
```

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

That's much better!

Square Brackets (1)

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.

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.

In [ ]:
```
from IPython import InteractiveShell
InteractiveShell.ast_node_interactivity = 'all'
```

In [ ]:
```
# Print out country column as Pandas Series
cars['country']

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

# Print out DataFrame with country and drives_right columns
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

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!

In [ ]:
```
# Print out first 3 observations
cars[0:3]

# Print out fourth, fifth and sixth observation
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.

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]]
```

In [ ]:
```
cars
```

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

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

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

```
cars_per_cap      588
country         Japan
drives_right    False
Name: JAP, dtype: object
```

	cars_per_cap	country	drives_right
AUS	731	Australia	False
EG	45	Egypt	True

You aced selecting observations from DataFrames; over to selecting both rows and columns!

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. 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]]
```

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

# Print sub-DataFrame
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.

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]]
```

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

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

# Print out cars_per_cap and drives_right as DataFrame
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

What a drill on indexing and selecting data from Pandas DataFrames! You've done great! It's time to head over to chapter 3 to learn all about logic, control flow, and filtering!

Chapter 3 - Logic, Control Flow and Filtering

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

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.

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.

In [ ]:
```
# 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

Greater and less than

You know 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

In [ ]:
```
# 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
```

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 the previous course? 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.

In [ ]:
```
# 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.

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 [ ]:
```
# 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
```

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.

False

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.

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(your_house > 13,
               your_house < 15)
```

In [ ]:

```
# 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]
```

Correcto perfecto!

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.

In [ ]:
```
# 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.

Add else

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?

In [ ]:
```
# 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.
```

Nice! Again, feel free to play around with different values of room and area some more. After, head over to the next exercise where you'll take this customization one step further!

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.

In [ ]:
```
# 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 kitchen.
medium size, nice!
```

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)?

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.

In [ ]:
```
# Import cars data
import pandas as pd
cars = pd.read_csv('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)
```

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

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

In [ ]:
```
cars[cars['drives_right']]
```

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

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!

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

car_maniac = cars[cars['cars_per_cap'] > 500]

# Print car_maniac
car_maniac
```

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

In [ ]:
```
cars[cars['cars_per_cap'] > 500]
```

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

Good job! The output shows that the US, Australia and Japan have a cars_per_cap of over 500.

In [ ]:
```
cars
```

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

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]
```

In [ ]:
```
# 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
medium
```

	cars_per_cap	country	drives_right
RU	200	Russia	True

Great work!

Chapter 4 - Loops

There are several techniques to repeatedly execute Python code. While loops are like repeated if statements; the for loop is there to iterate over all kinds of data structures. Learn all about them in this chapter.

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
```

Answer: 3

Correct! After 3 runs, x will be equal to 4, causing x < 4 to evaluate to False. This means that the while loop is executed 3 times, giving three printouts.

Basic while loop

Below you can find the example 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

In [ ]:

```
# Initialize offset
offset = 8

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

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

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
```

```
# 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.

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

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

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

Good work! The while loop is not that often used in Data Science, so let's head over to the for loop.

Loop over a list

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

# Code the for loop
for elements in areas:
    print(elements)
```

```
11.25
18.0
20.0
10.75
9.5
```

Great! That wasn't too hard, was it?

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:

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

In [ ]:

```
# areas list
areas = [11.25, 18.0, 20.0, 10.75, 9.50]

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

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

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?

In [ ]:

```
# 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
```

Much better!

Loop over list of lists

Remember the house variable from the Intro to Python course? . 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!

In [ ]:
```
# 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 name in house:
    print("the ",name[0]," is", name[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
```

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 europe 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!

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

# Iterate over europe
for key, value in europe.items():
    print("the capital of ",key,"is", value)
```

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

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

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.

In [ ]:
```
import numpy as np
```

In [ ]:
```
np_height = np.array([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,
,74,70,73,75,76,76,78,74,74,76,77,81,78,75,77,75,76,74,72,72,75,73,73,73,70,70,70,76,68,71,72,75,75,75,75,68,74,78,71,73,76,74,74,
,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,
,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,
,76,75,74,76,75,73,71,76])
```

In [ ]:
```
np_baseball = np.array([[74,180,74,215,72,210,72,210,73,188,69,176,69,209,71,200,76,231
,71,180,73,188,73,180,74,185,74,160,69,180,70,185,73,189,75,185
,78,219,79,230,76,205,74,230,76,195,72,180,71,192,75,225,77,203
,74,195,73,182,74,188,78,200,73,180,75,200,73,200,75,245,75,240
,74,215,69,185,71,175,74,199,73,200,73,215,76,200,74,205,74,206
,70,186,72,188,77,220,74,210,70,195,73,200,75,200,76,212,76,224
,78,210,74,205,74,220,76,195,77,200,81,260,78,228,75,270,77,200
,75,210,76,190,74,220,72,180,72,205,75,210,73,220,73,211,73,200
,70,180,70,190,70,170,76,230,68,155,71,185,72,185,75,200,75,225
,75,225,75,220,68,160,74,205,78,235,71,250,73,210,76,190,74,160
,74,200,79,205,75,222,73,195,76,205,74,220,74,220,73,170,72,185
,74,195,73,220,74,230,72,180,73,220,69,180,72,180,73,170,75,210
```

```
,75,215,73,200,72,213,72,180,76,192,74,235,72,185,77,235,74,210
,77,222,75,210,76,230,80,220,74,180,74,190,75,200,78,210,73,194
,73,180,74,190,75,240,76,200,71,198,73,200,74,195,76,210,76,220
,74,190,73,210,74,225,70,180,72,185,73,170,73,185,73,185,73,180
,71,178,74,175,74,200,72,204,74,211,71,190,74,210,73,190,75,190
,75,185,79,290,73,175,75,185,76,200,74,220,76,170,78,220,74,190
,76,220,72,205,74,200,76,250,74,225,75,215,78,210,75,215,72,195
,74,200,72,194,74,220,70,180,71,180,70,170,75,195,71,180,71,170
,73,206,72,205,71,200,73,225,72,201,75,225,74,233,74,180,75,225
,73,180,77,220,73,180,76,237,75,215,74,190,76,235,75,190,73,180
,71,165,76,195]]).reshape(200, 2)
```

In [ ]:

```
# Import numpy as np
import numpy as np

# For loop over np_height
for x in np_height:
    print(x,"inches")

# For loop over np_baseball
for x in np.nditer(np_baseball):
    print(x)
```

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

74
180
74
215
72
210
72
210
73
188
69
176
69
209
71
200
76
231
71
180
73
188
73
180
74
185
74
160
69
180
70
185
73
189
75
185
78
219
79
230
76
205
74
230
76
195
72
180
71
192

75
225
77
203
74
195
73
182
74
188
78
200
73
180
75
200
73
200
75
245
75
240
74
215
69
185
71
175
74
199
73
200
73
215
76
200
74
205
74
206
70
186
72
188
77
220
74
210
70
195

73
200
75
200
76
212
76
224
78
210
74
205
74
220
76
195
77
200
81
260
78
228
75
270
77
200
75
210
76
190
74
220
72
180
72
205
75
210
73
220
73
211
73
200
70
180
70
190
70
170

76
230
68
155
71
185
72
185
75
200
75
225
75
225
75
220
68
160
74
205
78
235
71
250
73
210
76
190
74
160
74
200
79
205
75
222
73
195
76
205
74
220
74
220
73
170
72
185
74
195

73
220
74
230
72
180
73
220
69
180
72
180
73
170
75
210
75
215
73
200
72
213
72
180
76
192
74
235
72
185
77
235
74
210
77
222
75
210
76
230
80
220
74
180
74
190
75
200
78
210

73
194
73
180
74
190
75
240
76
200
71
198
73
200
74
195
76
210
76
220
74
190
73
210
74
225
70
180
72
185
73
170
73
185
73
185
73
180
71
178
74
175
74
200
72
204
74
211
71
190

74
210
73
190
75
190
75
185
79
290
73
175
75
185
76
200
74
220
76
170
78
220
74
190
76
220
72
205
74
200
76
250
74
225
75
215
78
210
75
215
72
195
74
200
72
194
74
220
70
180

71
180
70
170
75
195
71
180
71
170
73
206
72
205
71
200
73
225
72
201
75
225
74
233
74
180
75
225
73
180
77
220
73
180
76
237
75
215
74
190
76
235
75
190
73
180
71
165
76
195

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.

In [ ]:

```
# Import cars data
import pandas as pd
cars = pd.read_csv('cars.csv', index_col = 0)

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

```
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
```

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'])
```

In [ ]:

```
# Import cars data
import pandas as pd
cars = pd.read_csv('cars.csv', index_col = 0)
```

```
# Adapt for loop
for lab, row in cars.iterrows() :
    print(lab,": ",row['cars_per_cap'],sep = "")
```

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

Add column (1)

You can 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.

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

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


# Print cars
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

Great, but you might remember that there is also an easier way to do this.

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 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:

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

# Use .apply(str.upper)
cars['COUNTRY'] = cars['country'].apply(str.upper)
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

Great job! It's time to blend everything you've learned together in a case-study. Head over to the next chapter!

Chapter 5 - Case Study: Hacker Statistics

This chapter blends together everything you've learned up to now. You will use hacker statistics to calculate your chances of winning a bet. Use random number generators, loops and matplotlib to get the competitive edge!

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 the 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.

In [ ]:

```
# 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
```

Great! Now let's simulate a dice.

Roll the dice

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

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.

np.random.randint(4, 8)

In [ ]:

```
np.random.seed(123)

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

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

6
3

Alright! Time to actually start coding things up!

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?

In [ ]:
```
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 < 6:
    step += 1
else :
    step = step + np.random.randint(1,7)

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

6 53

Cool! You threw a 6, so the code for the else statement was executed. You threw again, and apparently you threw 3, causing you to take three steps up: you're currently at step 53.

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.

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

# Complete the ___
```

```python
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, 4
6, 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!

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)

In [ ]:

```python
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, 4
7, 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!

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.

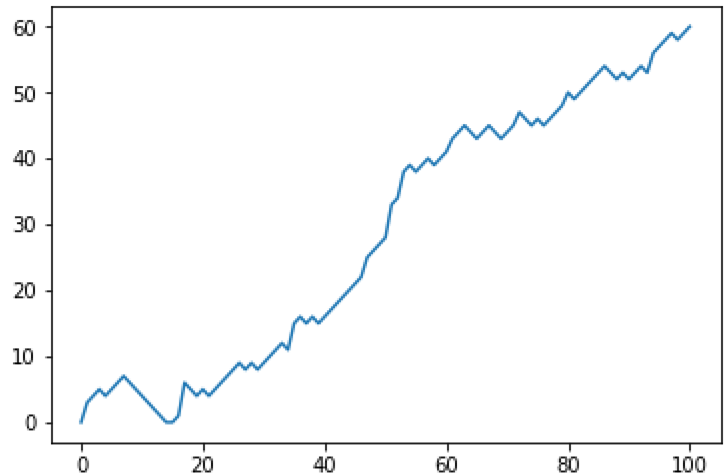
If you pass only one argument, Python will know what to do and will use the index of the list to map onto the x axis, and the values in the list onto the y axis.

In [ ]:

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

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

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

    random_walk.append(step)

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

```
# Plot random_walk
plt.plot(random_walk)
```

```
[<matplotlib.lines.Line2D at 0xc879d10>]
```



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

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.

In [ ]:

```
np.random.seed(123)
# Initialize all_walks (don't change this line)
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)
```

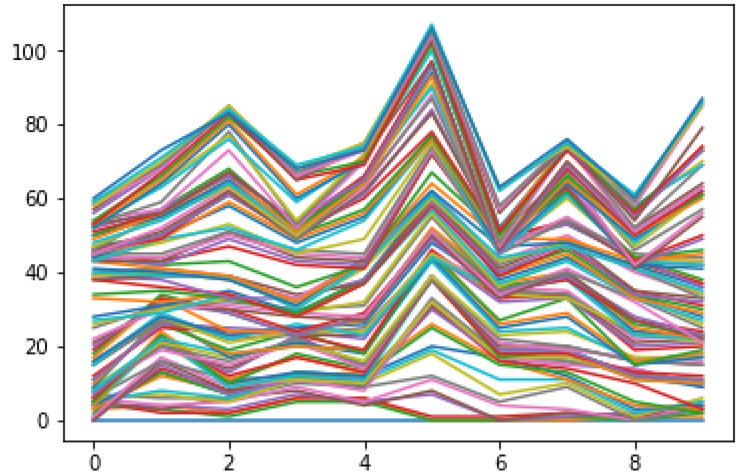
```
[[0, 3, 4, 5, 4, 5, 6, 7, 6, 5, 4, 3, 2, 1, 0, 0, 1, 6, 5, 4, 5, 4, 5, 6, 7, 8, 9, 8, 9, 8, 9, 10, 11, 12, 11, 15, 16, 15, 16, 1
5, 16, 17, 18, 19, 20, 21, 22, 25, 26, 27, 28, 33, 34, 38, 39, 38, 39, 40, 39, 40, 41, 43, 44, 45, 44, 43, 44, 45, 44, 43, 44, 4
5, 47, 46, 45, 46, 45, 46, 47, 48, 50, 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, 4
9, 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, 2
3, 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, 5
3, 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, 8
3], [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, 4
4, 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, 6
7, 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, 4
1, 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, 7
4, 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, 3
8, 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, 6
1, 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, 10
4, 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, 3
9, 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, 4
6, 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, 4
4, 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, 6
6, 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, 1
4, 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, 3
4, 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, 4
9, 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, 6
1, 60, 61, 62, 63, 64, 63, 69, 70, 69, 73, 74, 73, 74, 73, 79, 85, 86, 85, 86, 87]]
```

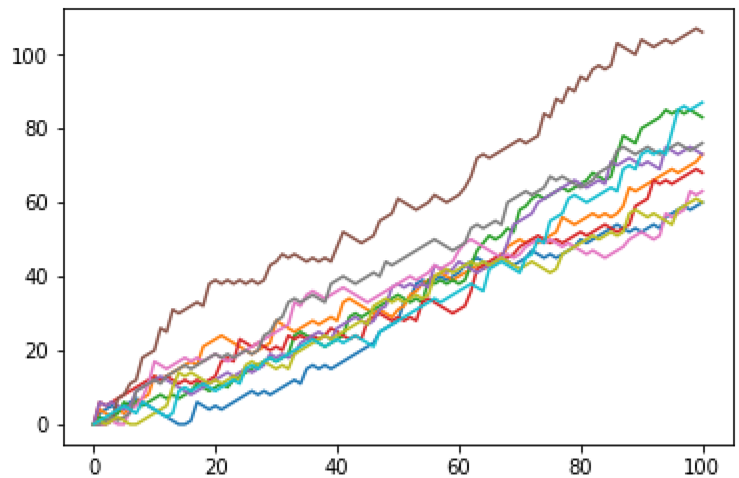
Visualize all walks

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!

In [ ]:

```
# 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_aw.transpose()

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





Good job! You can clearly see how the different simulations of the random walk went. Transposing the 2D Numpy array was crucial; otherwise Python misunderstood.

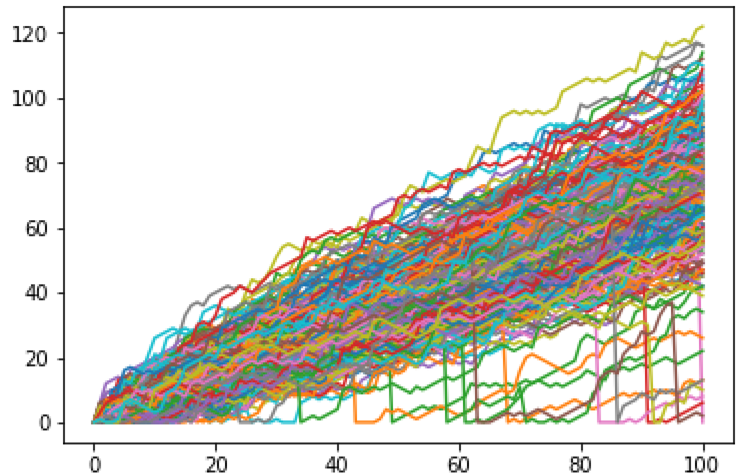
Implement clumsiness

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.

In [ ]:

```
np.random.seed(123)
# Simulate random walk 250 times
all_walks = []
for i in range(250) :
    random_walk = [0]
    for x in range(100) :
        step = random_walk[-1]
        dice = np.random.randint(1,7)
        if dice <= 2:
            step = max(0, step - 1)
        elif dice <= 5:
            step = step + 1
        else:
            step = step + np.random.randint(1,7)

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

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

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



Superb! Look at the plot. In some of the 250 simulations you're indeed taking a deep dive down!

Plot the distribution

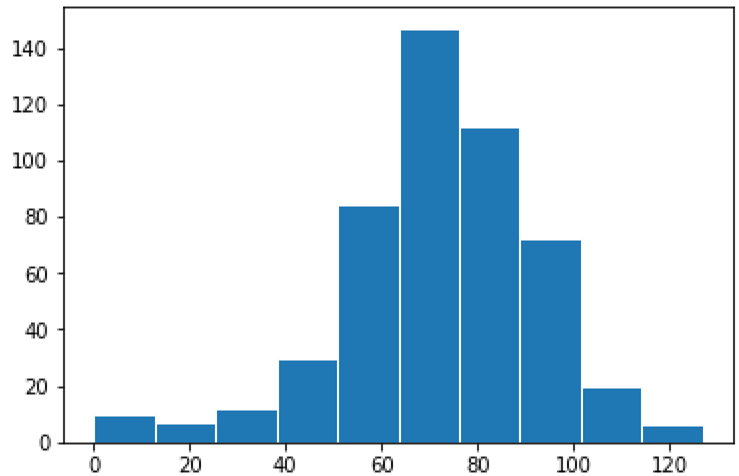
All these fancy visualizations have put us on a sidetrack. We still have to solve the million-dollar problem: What are the odds that you'll reach 60 steps high on the Empire State Building?

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.

In [ ]:

```
np.random.seed(123)
# Simulate random walk 500 times
all_walks = []
for i in range(500) :
    random_walk = [0]
    for x in range(100) :
        step = random_walk[-1]
        dice = np.random.randint(1,7)
        if dice <= 2:
            step = max(0, step - 1)
        elif dice <= 5:
            step = step + 1
        else:
            step = step + np.random.randint(1,7)
        if np.random.rand() <= 0.001 :
            step = 0
        random_walk.append(step)
    all_walks.append(random_walk)

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

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

# Plot histogram of ends, display plot
plt.hist(ends, ec='white')
plt.show()
```



Great job! Have a look at a histogram; what do you think your chances are?

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.

In [ ]:
```
chance = np.sum(ends >= 60)/500 * 100
chance
```

```
78.4
```

Correct! Seems like you have a pretty high chance of winning the bet!