**Intermediate Python for Data Science: Matplotlib**

Learn to visualize real data with matplotlib's functions.

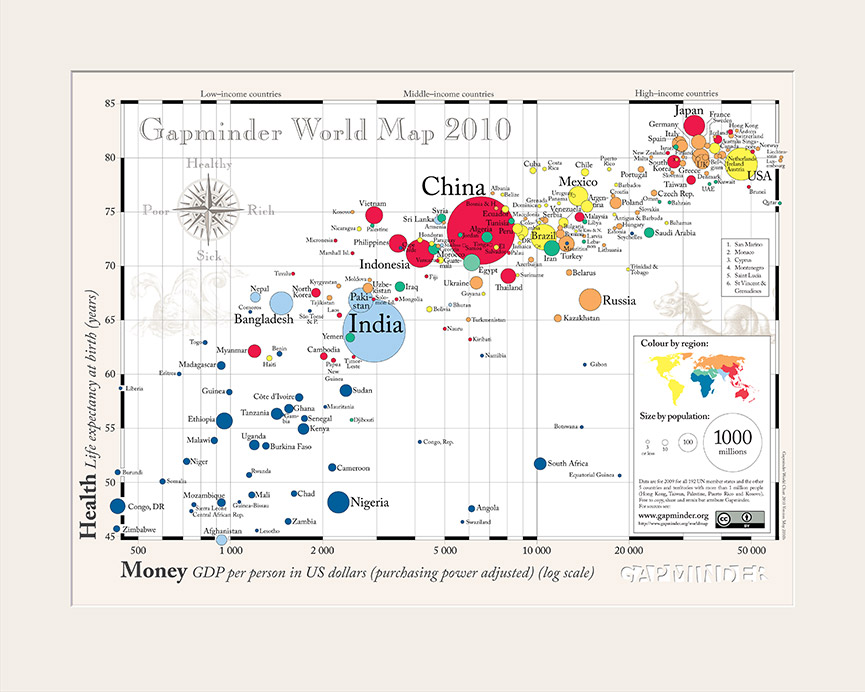
**Course Excerpt: Intermediate Python for Data Science - Matplotlib**

*Below is an excerpt--video and transcript--from the first chapter of the* [*Intermediate Python for Data Science course*](https://www.datacamp.com/courses/intermediate-python-for-data-science/)*.* [*Here*](https://www.datacamp.com/courses/intermediate-python-for-data-science/) *is the full chapter, including interactive exercises.*

**Basic plots with matplotlib**

Hi, my name is Filip, and I'm a data scientist at DataCamp. In this intermediate Python course, you will further enhance your Python skills for data science. You will learn how to visualize data and store data in new data structures. Along the way, you will master control structures, which you will need to customize the flow of your scripts and algorithms. We'll finish this chapter with a case study, where you'll blend together everything you've learned to solve a cool problem.

This first chapter is about data visualization, which is a very important part of data analysis. First of all, you will use it continuously to explore your dataset. The better you understand your data, the better you'll be able to extract insights. And once you've found those insights, again, you'll need visualization to be able to share your precious insights with other people. As an example, have a look at this beautfiul plot.



It's made by the Swedish professor Hans Rosling. His talks about global development have been viewed millions of times. And what makes them so intriguing, is that by making beautiful plots, he allows the data to tell their own story. Here we see a bubble chart, where each bubble represents a country. The bigger the bubble, the bigger the country's population, so the two biggest bubbles here are China and India.

There are 2 axes. The horizontal axis shows the GDP per capita, in US dollars. The vertical axis shows life expectancy. We clearly see that people live longer in countries with a higher GDP per capita. Still, there is a huge difference in life expectancy between countries on the same income level.

Now why do I tell you all of this? Well, because by the end of this chapter, you'll be able to build this beautiful plot yourself.

There are many visualization packages in Python, but the mother of them all, is matplotlib. You will need its subpackage pyplot. By convention, this subpackage is imported as plt, like this.

For our first example, let's try to gain some insights in the evolution of the world population. I have a list with years here, year, and a list with corresponding populations, expressed in billions, pop. In the year 1970, for example, 3.7 billion people lived on planet Earth.

To plot this data as a line chart, we call plt.plot() and use our two lists as arguments. The first argument corresponds to the horizontal axis, and the second one to the vertical axis. You might think that a plot will pop up right now, but Python's pretty lazy. It will wait for the **show()** function to actually display the plot. This is because you might want to add some extra ingredients to your plot before actually displaying it, such as titles and label customizations. I'll talk about that some more later on. Just remember this: the **plot()** function tells Python what to plot and how to plot it. **show()** actually displays the plot.

When we look at our plot, we see that the years are indeed shown on the horizontal axis, and the populations on the vertical axis. There are four data points, and Python draws a line between them. In 1950, the world population was around 2 point 5 billion. In 2010, it was around 7 billion. So the world population has almost tripled in sixty years, that's pretty scary. What if the population keeps on growing like that? Will the world become over populated? You'll find out in the exercises.

Let me first introduce you to another type of plot: the scatter plot. To create it, we can start from the code from before. This time, though, you change the plot function to scatter. The resulting scatter plot simply plots all the individual data points; Python doesn't connect the dots with a line. For many applications, the scatter plot is often a better choice than the line plot, so remember this scatter function well. You could also say that this is a more -honest- way of plotting your data, because you can clearly see that the plot is based on just four data points.

Now that we've got the basics of matplotlib covered, it's your turn to build some awesome plots!

# define a simple function

import matplotlib.pyplot as plt

year = [1950, 1970, 1990, 2010]

pop = [2.519, 3.692, 5.263, 6.972]

plt.plot(year, pop)

plt.show()

import matplotlib.pyplot as plt

year = [1950, 1970, 1990, 2010]

pop = [2.519, 3.692, 5.263, 6.972]

plt.scatter(year, pop)

plt.show()

**The Histogram**

In this video, I'll introduce the histogram. The histogram is a type of visualization that's very useful to explore your data. It can help you to get an idea about the distribution of your variables. To see how it works, imagine 12 values between 0 and 6. I've put them along a number line here. To build a histogram for these values, you can divide the line into equal chunks, called bins. Suppose you go for 3 bins, that each have a width of 2. Next, you count how many data points sit inside each bin. There's 4 data points in the first bin, 6 in the second bin and 2 in the third bin. Finally, you draw a bar for each bin. The height of the bar corresponds to the number of data points that fall in this bin. The result is a histogram, which gives us a nice overview on how the 12 values are distributed. Most values are in the middle, but there are more values below 2 than there are values above 4.

Of course, also matplotlib is able to build histograms. As before, you should start by importing the pyplot package that's inside matplotlib. Next, you can use the **hist()** function. Let's open up its documentation. There's a bunch of arguments you can specify, but the first two here are the most important ones. x should be a list of values you want to build a histogram for. You can use the second argument, bins, to tell Python in how many bins the data should be divided. Based on this number, **hist()** will automatically find appropriate boundaries for all bins, and calculate how may values are in each one. If you don't specify the bins argument, it will by 10 by default.

So to generate the histogram that you've seen before, let's start by building a list with the 12 values. Next, you simply call **hist()** and pass this list as an input, so it's matched to the argument x. I also specified the bins argument to be 3, so that the values are divided in three bins. If you finally call the show function, a nice histogram results. Histograms are really useful to give a bigger picture. As an example, have a look at this so-called population pyramid. The age distribution is shown, for both males and females, in the European union. Notice that the histograms are flipped 90 degrees; the bins are horizontal now. The bins are largest for the ages 40 to 44, where there are 20 million males and 20 million females. They are the so-called baby boomers.

These are figures of the year 2010. What do you think will have changed in 2050? Let's have a look. The distribution is flatter, and the baby boom generation has gotten older. With the blink of an eye, you can easily see how demographics will be changing over time. That's the true power of histograms at work here! Now head over to the exercises to experiment with histograms yourself!

values = [0,0.6,1.4,1.6,2.2,2.5,2.6,3.2,3.5,3.9,4.2,6]

import matplotlib.pyplot as plt

plt.hist(values,bins=3)

plt.show()

**Customization**

Creating a plot is one thing. Making the correct plot, that makes the message very clear, is the real challenge. For each visualization, you have many options. First of all, there are the different plot types. And for each plot, you can do an infinite number of customizations. You can change colors, shapes, labels, axes, and so on. The choice depends on, one, the data, and two, the story you want to tell with this data. Since there are a so many possible customizations, the best way to learn this, is by example.

Let's start with the code in this script to build a simple line plot. It's similar to the line plot we've created in the first video, but this time the year and pop lists contain more data, including projections until the year 2100, forecasted by the United Nations. If we run this script, we already get a pretty nice plot: it shows that the population explosion that's going on, will have slowed down by the end of the century.

But some things can be improved. First, it should be clearer which data we are displaying, especially to people who are seeing the graph for the first time. And second, the plot really needs to draw the attention to the population explosion. The first thing you always need to do is label your axes. Let's do this by adding the xlabel() and ylabel() functions. As inputs, we pass strings that should be placed alongside the axes. Make sure to call these functions before calling the **show()** method, otherwise your customizations will not be displayed. If we run the script again, this time the axes are annotated. We're also going to add a title to our plot, with the title function. We pass the actual title, 'World Population Projections', as an argument. And there's the title!

So, using xlabel, ylabel and title, we can give the reader more information about the data on the plot: now they can at least tell what the plot is about. To put the population growth in perspective, I want to have the y-axis start from zero. You can do this with the **yticks()** function. The first input is a list, in this example with the numbers zero up to ten, with intervals of 2. If we run this, the plot will change: the curve shifts up. Now it's clear that already in 1950, there were already about 2.5 billion people on this planet.

Next, to make it clear we're talking about billions, we can add a second argument to the yticks() function, which is a list with the display names of the ticks. This list should have the same length as the first list. The tick 0 gets the name 0, the tick 2 gets the name 2B, the tick 4 gets the name 4B and so on. By the way, B stands for Billions here. If we run this version of the script, the labels will change accordingly, great.

Finally, let's add some more historical data to accentuate the population explosion in the last 60 years. On Wikipedia, I found the world population data for the years 1800, 1850 and 1900. I can write them in list form and append them to the pop and year lists with the plus sign. If I now run the script once more, three data points are added to the graph, giving a more complete picture. Now that's how you turn an average line plot into a visual that has a clear story to tell! Over to you now. Head over to the exercises, gradually customize the world development chart and become the next Hans Rosling!

import matplotlib.pyplot as plt

import pandas as pd

year = list(range(1950, 2101))

pop = [2.53,2.57,2.62,2.67,2.71,2.76,2.81,2.86,2.92,2.97,3.03,3.08,3.14,3.2,3.26,3.33,3.4,3.47,3.54,3.62,3.69,3.77,3.84,3.92,4.,4.07,4.15,4.22,4.3,4.37,4.45,4.53,4.61,4.69,4.78,4.86,4.95,5.05,5.14,5.23,5.32,5.41,5.49,5.58,5.66,5.74,5.82,5.9,5.98,6.05,6.13,6.2,6.28,6.36,6.44,6.51,6.59,6.67,6.75,6.83,6.92,7.,7.08,7.16,7.24,7.32,7.4,7.48,7.56,7.64,7.72,7.79,7.87,7.94,8.01,8.08,8.15,8.22,8.29,8.36,8.42,8.49,8.56,8.62,8.68,8.74,8.8,8.86,8.92,8.98,9.04,9.09,9.15,9.2,9.26,9.31,9.36,9.41,9.46,9.5,9.55,9.6,9.64,9.68,9.73,9.77,9.81,9.85,9.88,9.92,9.96,9.99,10.03,10.06,10.09,10.13,10.16,10.19,10.22,10.25,10.28,10.31,10.33,10.36,10.38,10.41,10.43,10.46,10.48,10.5,10.52,10.55,10.57,10.59,10.61,10.63,10.65,10.66,10.68,10.7,10.72,10.73,10.75,10.77,10.78,10.79,10.81,10.82,10.83,10.84,10.85]

pop = [1,1.262,1.650] + pop

year = [1800,1850,1900] + year

plt.plot(year, pop)

plt.xlabel('Year')

plt.ylabel('Population')

plt.title('World Population Projections')

plt.yticks([0,2,4,6,8,10],['0','2B','4B','6B','8B','10B'])

plt.show()