**Dates in Python**

Hi! My name is Max Shron, I will be your instructor for this course on working with dates and times in Python. Dates are everywhere in data science. Stock prices go up and down, experiments begin and end, people are born, politicians take votes, and on and on. All these events happen at a particular point in time. Knowing how to analyze data over time is a core data science skill.

This course is divided into four chapters. The first chapter will be about working with dates and calendars. In chapter two, we will add time into the mix, and combine dates and times. In chapter three, we'll tackle one of the toughest parts of working with time: time zones and Daylight Saving. And finally, in chapter four, we'll connect what we've learned about working with dates and times to explore how Pandas can make answering even complex questions about dates much easier.

**Dates in Python**

Let's begin. Python has a special date class, called "date", which you will use to represent dates. A date, like a string, or a number, or a numpy array, has special rules for creating it and methods for working with it. In this lesson, we're going to discuss creating dates and extracting some basic information out of them.

**Why do we need a date class in Python?**

Why do we need a special date class? Let's have a look. To understand how dates work, in this chapter you're going to be exploring 67 years of Hurricane landfalls in the US state of Florida. two\_hurricanes is a list with the dates of two hurricanes represented as strings: the last 2016 hurricane (on October 7th, 2016) and the first 2017 hurricane (on June 21st, 2017). The dates are represented in the US style, with the month, then the day, then the year. Suppose you want to do something interesting with these dates. How would you figure out how many days had elapsed between them? How would you check that they were ordered from earliest to latest? How would you know which day of the week each was? Doing these things manually would be challenging, but Python makes all of them easy. By the end of this chapter, you'll know how to do each of these things yourself.



**Creating date objects**

To create a date object, we start by importing the date class. The collection of date and time-related classes are stored in the "datetime" package. We create a date using the date() function. Here we've created dates corresponding to the two hurricanes, now as Python date objects. The inputs to date() are the year, month, and day. The first date is October 7, 2016, and the second date is June 21, 2017. The order is easy to remember: it goes from the biggest to smallest. Year, month, day. Later in this chapter, you'll create dates directly from lists of strings, but in this lesson, you're going to stick to creating dates by hand or using lists of already created dates.

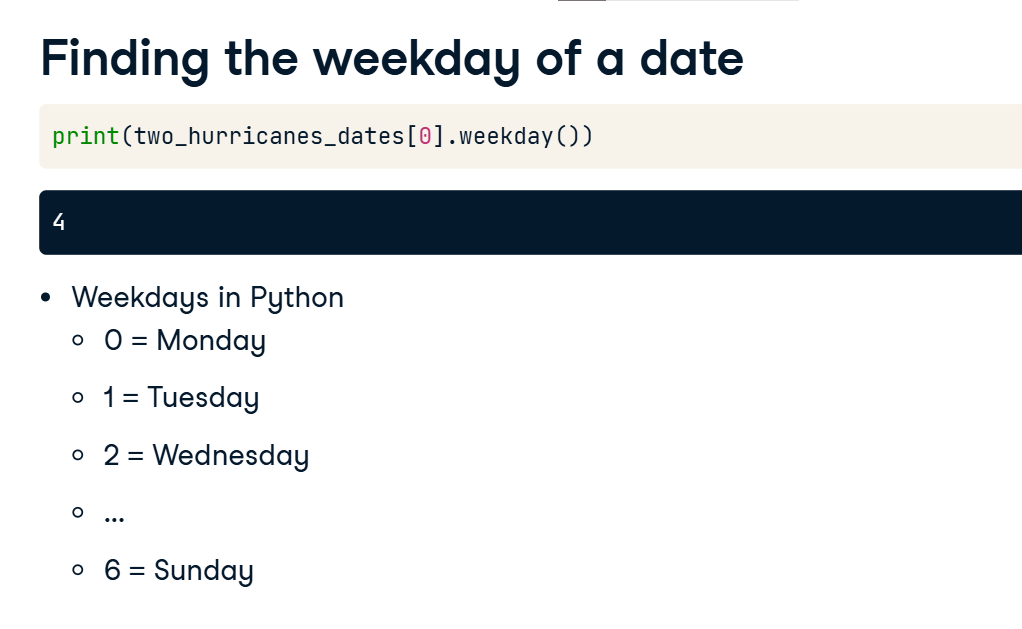
**Attributes of a date**

You can access individual components of a date using the date's attributes. You can access the year of the date using the year attribute, like so, and the result is 2016. Similarly, you can access the month and day using the month and day attributes like so.



**Finding the weekday of a date**

You can also ask Python to do more complicated work. Here we call the weekday() method on the date, and see that the weekday is 4. What does 4 mean here? Python counts weekdays from 0, starting on Monday. 1 is Tuesday, 2 is Wednesday, and so on, up to 6 being a Sunday. This date was a Friday.



**Math with Dates**

In the last lesson, we discussed how to create date objects and access their attributes. In this lesson, we're going to talk about how to do math with dates: counting days between events, moving forward or backward by a number of days, putting them in order, and so on.

**Math with dates**

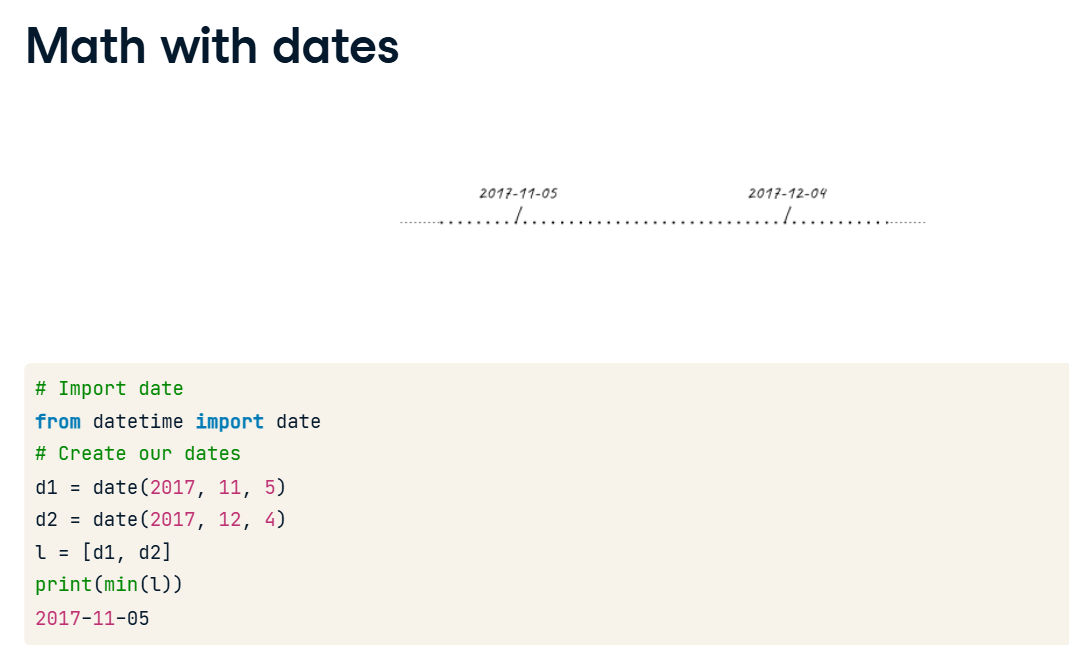
Let's take a step back. Think back to when you first learned arithmetic. You probably started with something like this: a number line. This one has the numbers 10 through 16 on it. A number line tells you what order the numbers go in, and how far apart numbers are from each other. Let's pick two numbers, 11 and 14, and represent them in Python as the variables a and b, respectively. We'll put them into a list, l. Python can tell us which number in this list is the least, using the min() function. min stands for the minimum. In this case, 11 is the lowest number in the list, so we get 11.

**Math with dates**

We can also subtract numbers. When you subtract two numbers, in this case subtracting 11 from 14, the result is 3. Said another way, if we took three steps from 11, we would get 14.

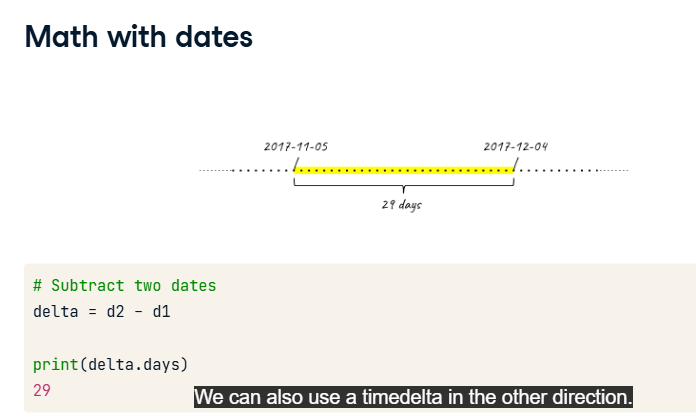
Now let's think about how this applies to dates. Let's call this line a calendar line, instead of a number line. Each dot on this calendar line corresponds to a particular day.

Let's put two dates onto this calendar line: November 5th, 2017, and December 4th, 2017. Let's represent this in Python. We start by importing the date class from the datetime package. We create two date objects: d1 is November 5th, 2017, and d2 is December 4th, 2017. As before, we put them into a list, l. What Python is doing under the hood, so to speak, is not that different from putting the dates onto a calendar line. For example, if we call min of l, we again get the "least" date, which means the earliest one. In this case, that's November 5th, 2017.

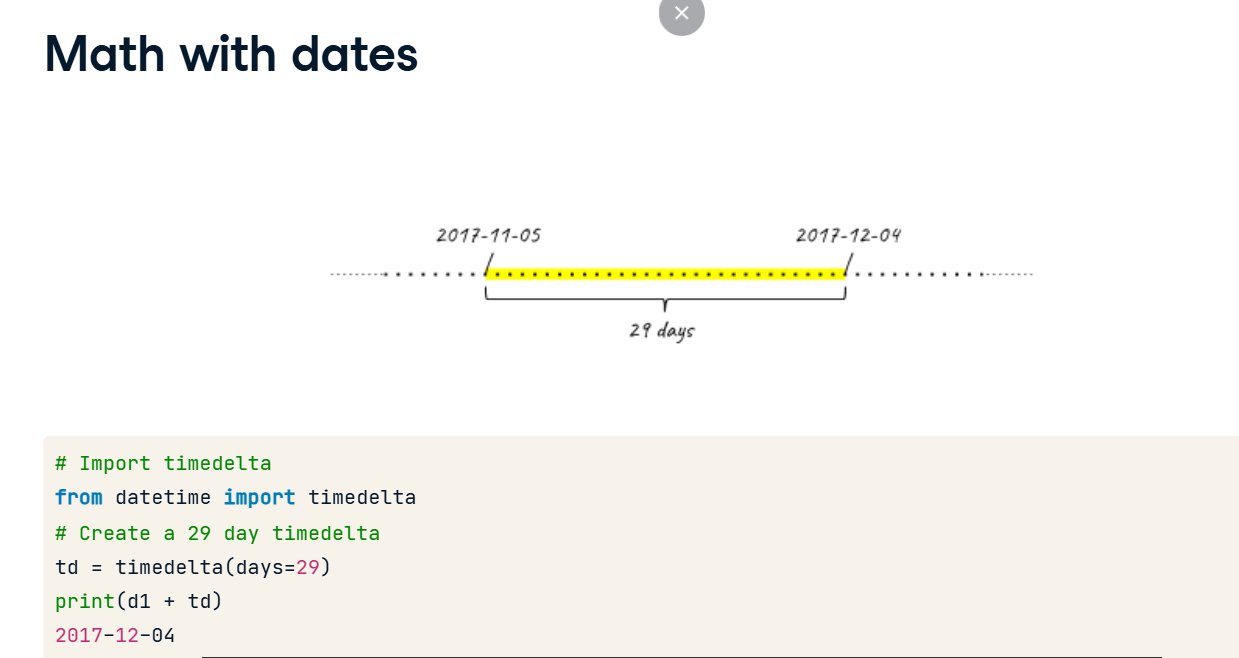


**Math with dates**

And just like numbers, we can subtract two dates. When we do this, we get an object of type "timedelta". Timedeltas give us the elapsed time between events. If you access the days attribute of this object, you can obtain the number of days between the two dates.



We can also use a timedelta in the other direction. First, let's import timedelta from datetime. Next, we create a 29-day timedelta, by passing days=29 to timedelta(). Now when we add td to our original date we get back December 4th, 2017. Python handled the fact that November has 30 days in it for us, without us having to remember which months are 30 day months, 31 day months, or 28 day months.



**Incrementing variables with +=**

Finally a quick side note: we will use the "plus-equals" operation a number of times in the rest of the course, so we should discuss it. If you aren't familiar with it, you can see how it works here. On the left-hand side, we create a variable x, set it to zero. If we set x equal to x + 1, we increment x by 1. Similarly, on the right-hand side, we set x = 0, and then we increment it with x += 1. It has the same effect, and we'll use it all the time for counting.

**Turning dates into strings**

Python has a very flexible set of tools for turning dates back into strings to be easily read. We want to put dates back into strings when, for example, we want to print results, but also if we want to put dates into filenames, or if we want to write dates out to CSV or Excel files.

**ISO 8601 format**

For example, let's create a date and see how Python prints it by default. As before, we import date from datetime and let's again create an object for November 5th, 2017. When we ask Python to print the date, it prints the year, day and then the month, separated by dashes, and always with two digits for the day and month. In the comment, you can see I've noted this as YYYY-MM-DD; four digit year, two digit month, and two digit day of the month. This default format is also known as ISO format, or ISO 8601 format, after the international standard ISO 8601 that it is based on. ISO 8601 strings are always the same length since month and day are written with 0s when they are less than 10. We'll talk about another advantage of ISO 8601 in a moment. If we want the ISO representation of a date as a string, say to write it to a CSV file instead of just printing it, you can call the isoformat() method. In this example, we put it inside a list so you can see that it creates a string.

**ISO 8601 format**

The ISO 8601 format has another nice advantage. To demonstrate, we've created a variable called some\_dates and represented two dates here as strings: January 1, 2000, and December 31, 1999. Dates formatted as ISO 8601 strings sort correctly. When we print the sorted version of this list, the earlier day is first, and the later date is second. For example, if we use ISO 8601 dates in filenames, they can be correctly sorted from earliest to latest. If we had month or day first, the strings would not sort in chronological order.

**Every other format**

If you don't want to put dates in ISO 8601 format, Python has a flexible set of options for representing dates in other ways, using the strftime() method.

**Every other format: strftime**

strftime() works by letting you pass a "format string" which Python uses to format your date. Let's see an example. We again create an example date of January 5th, 2017. We then call strftime() on d, with the format string of % capital Y. Strftime reads the % capital Y and fills in the year in this string for us. strftime() though is very flexible: we can give it arbitrary strings with % capital Y in them for the format string, and it will stick the year in. For example, we can use the format string of "Year is %Y".

**Every other format: strftime**

Strftime has other placeholders besides %Y: % lowercase m gives the month, and % lowercase d gives the day of the month. Using these, we can represent dates in arbitrary formats for whatever our needs are.

**Turning dates into strings**

In this lesson, we discussed how Python can represent a date as a string. We emphasized the importance and utility of ISO 8601 format, but also introduced strftime(), which lets you turn dates in a wide variety of strings depending on your needs.