**Uniformity**

Stellar work on chapter 2! You're now an expert at handling categorical and text variables.

**In this chapter**

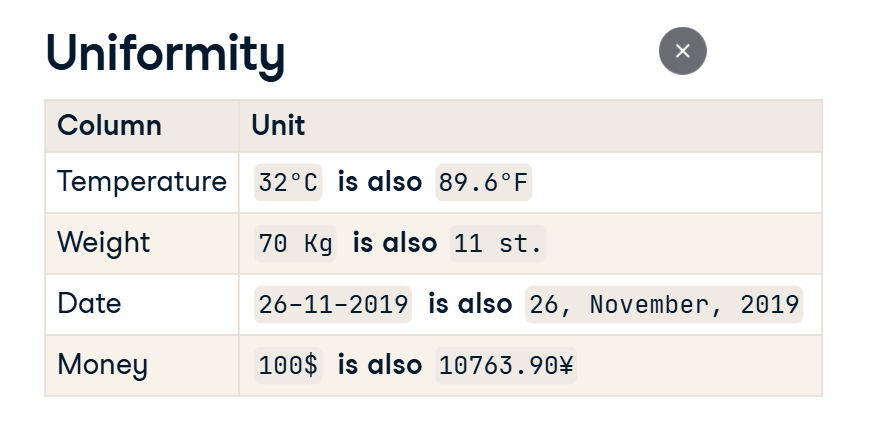
In this chapter, we're looking at more advanced data cleaning problems, such as uniformity, cross field validation and dealing with missing data.

**Data range constraints**

In chapter 1, we saw how out of range values are a common problem when cleaning data, and that when left untouched, can skew your analysis.

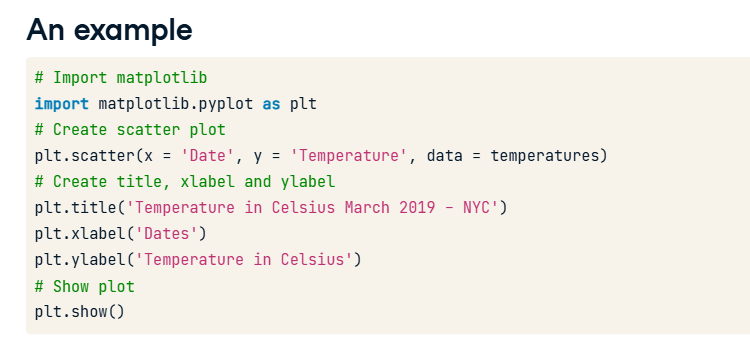
**Uniformity**

In this lesson, we're going to tackle a problem that could similarly skew our data, which is unit uniformity. For example, we can have temperature data that has values in both Fahrenheit and Celsius, weight data in Kilograms and in stones, dates in multiple formats, and so on. Verifying unit uniformity is imperative to having accurate analysis.



**An example**

Here's a dataset with average temperature data throughout the month of March in New York City. The dataset was collected from different sources with temperature data in Celsius and Fahrenheit merged together. We can see that unless a major climate event occurred,this value here is most likely Fahrenheit, not Celsius. Let's confirm the presence of these values visually.

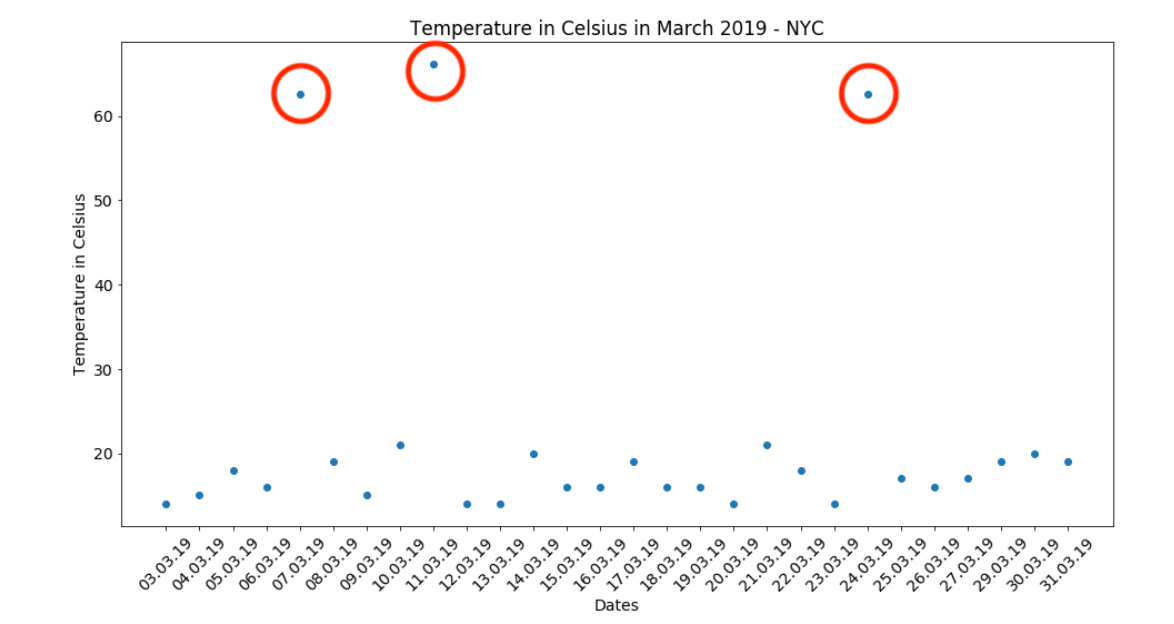


**An example**

We can do so by plotting a scatter plot of our data. We can do this using matplotlib.pyplot, which was imported as plt. We use the plt dot scatter function, which takes in what to plot on the x axis, the y axis, and which data source to use. We set the title, axis labels with the helper functions seen here, show the plot with plt dot show,

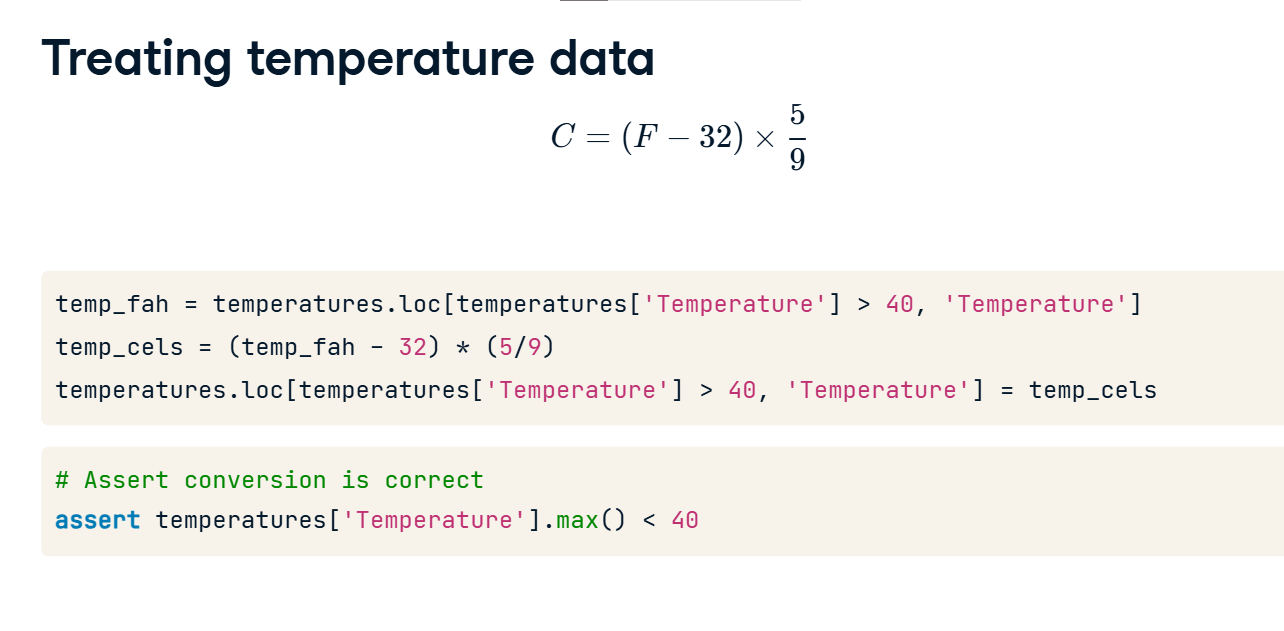
and voila.

Notice these values here? They all must be fahrenheit.



**Treating temperature data**

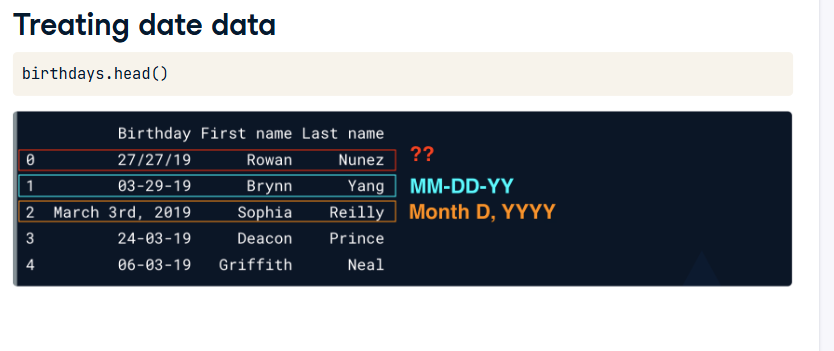
A simple web search returns the formula for converting Fahrenheit to Celsius. To convert our temperature data, we isolate all rows of temperature column where it is above 40 using the loc method. We chose 40 because it's a common sense maximum for Celsius temperatures in New York City. We then convert these values to Celsius using the formula above, and reassign them to their respective Fahrenheit values in temperatures. We can make sure that our conversion was correct with an assert statement, by making sure the maximum value of temperature is less than 40.



**Treating date data**

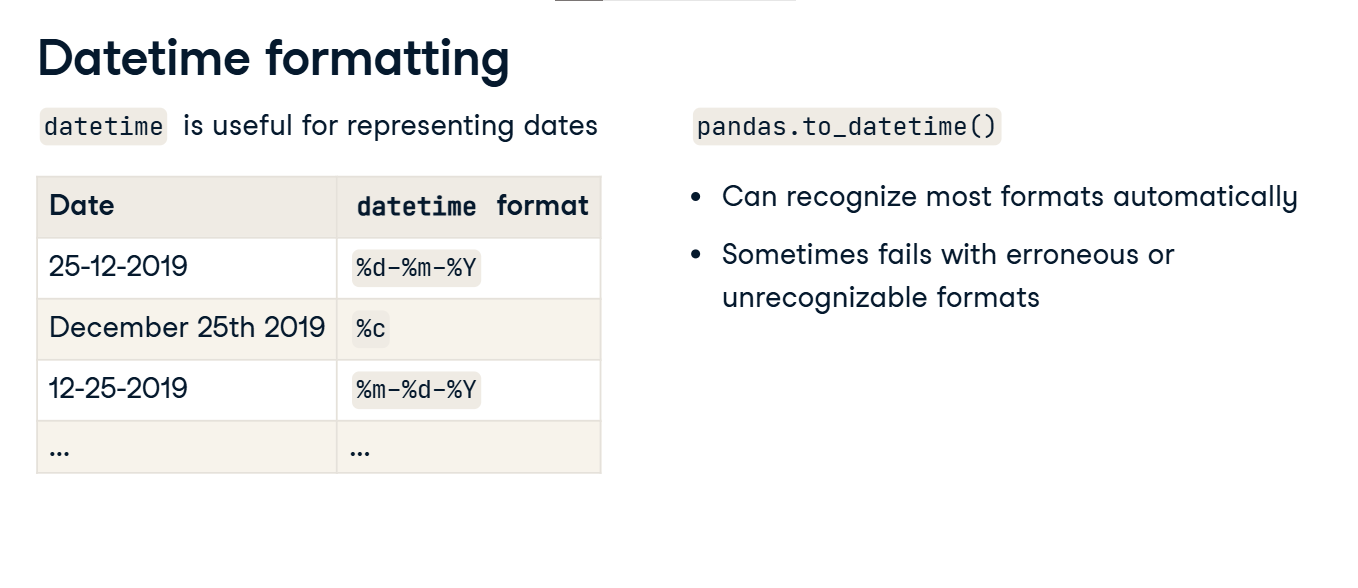
Here's another common uniformity problem with date data. This is a DataFrame called birthdays containing birth dates for a variety of individuals. It has been collected from a variety of sources and merged into one.

Notice the dates here? The one in blue has the month, day, year format, whereas the one in orange has the month written out. The one in red is obviously an error, with what looks like a day day year format. We'll learn how to deal with that one as well.



**Datetime formatting**

We already discussed datetime objects. Without getting too much into detail, datetime accepts different formats that help you format your dates as pleased. The pandas to datetime function automatically accepts most date formats, but could raise errors when certain formats are unrecognizable. You don't have to memorize these formats, just know that they exist and are easily searchable!

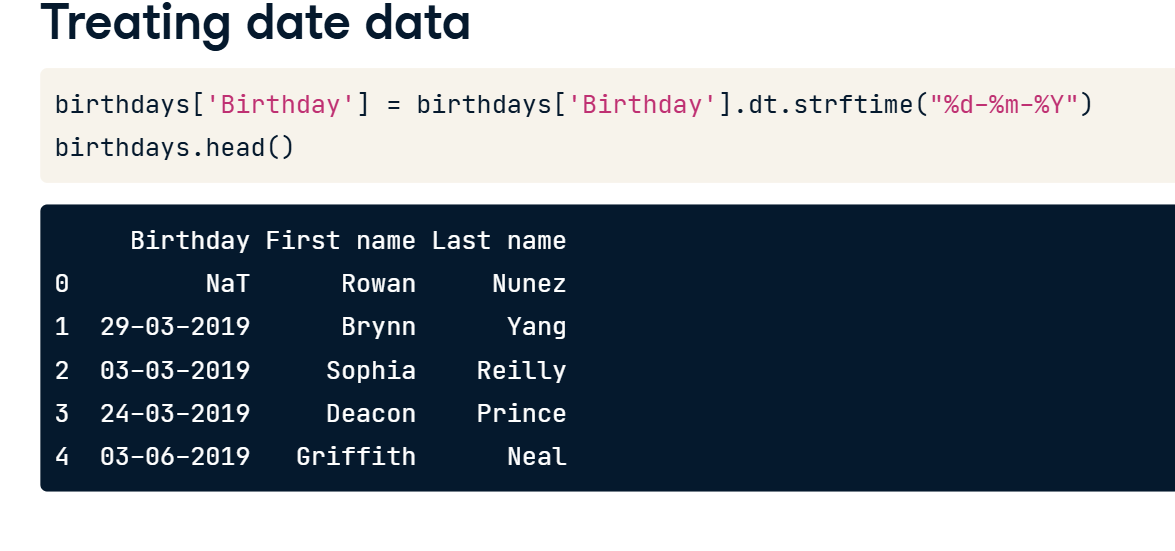


**Treating date data**

You can treat these date inconsistencies easily by converting your date column to datetime. We can do this in pandas with the to\_datetime function. However this isn't enough and will most likely return an error, since we have dates in multiple formats, especially the weird day/day/format which triggers an error with months. Instead we set the infer\_datetime\_format argument to True, and set errors equal to coerce. This will infer the format and return missing value for dates that couldn't be identified and converted instead of a value error.



This returns the birthday column with aligned formats, with the initial ambiguous format of day day year, being set to NAT, which represents missing values in Pandas for datetime objects.

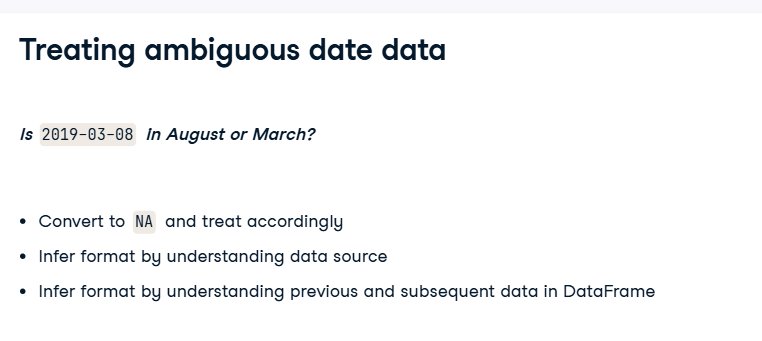


**Treating date data**

We can also convert the format of a datetime column using the dt dot strftime method, which accepts a datetime format of your choice. For example, here we convert the Birthday column to day month year, instead of year month day.

**Treating ambiguous date data**

However a common problem is having ambiguous dates with vague formats. For example, is this date value set in March or August? Unfortunately there's no clear cut way to spot this inconsistency or to treat it. Depending on the size of the dataset and suspected ambiguities, we can either convert these dates to NAs and deal with them accordingly. If you have additional context on the source of your data, you can probably infer the format. If the majority of subsequent or previous data is of one format, you can probably infer the format as well. All in all, it is essential to properly understand where your data comes from, before trying to treat it, as it will make making these decisions much easier.

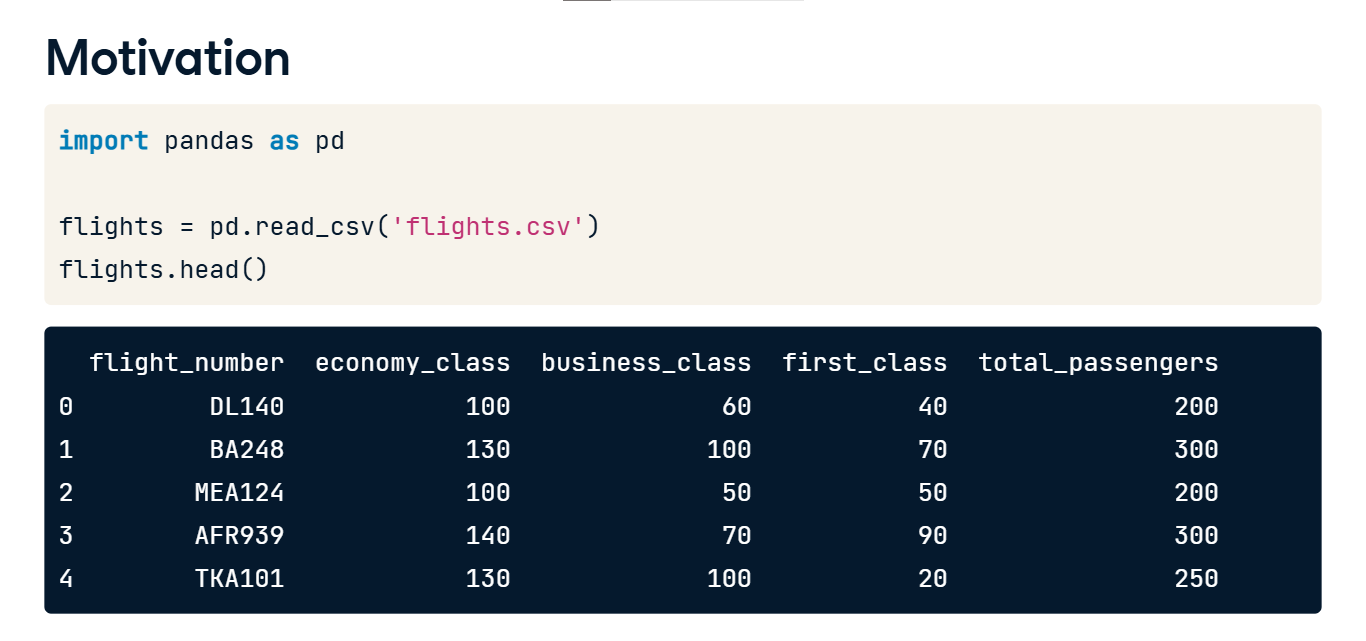


**Cross field validation**

Hi and welcome to the second lesson of this chapter! In this lesson we'll talk about cross field validation for diagnosing dirty data.

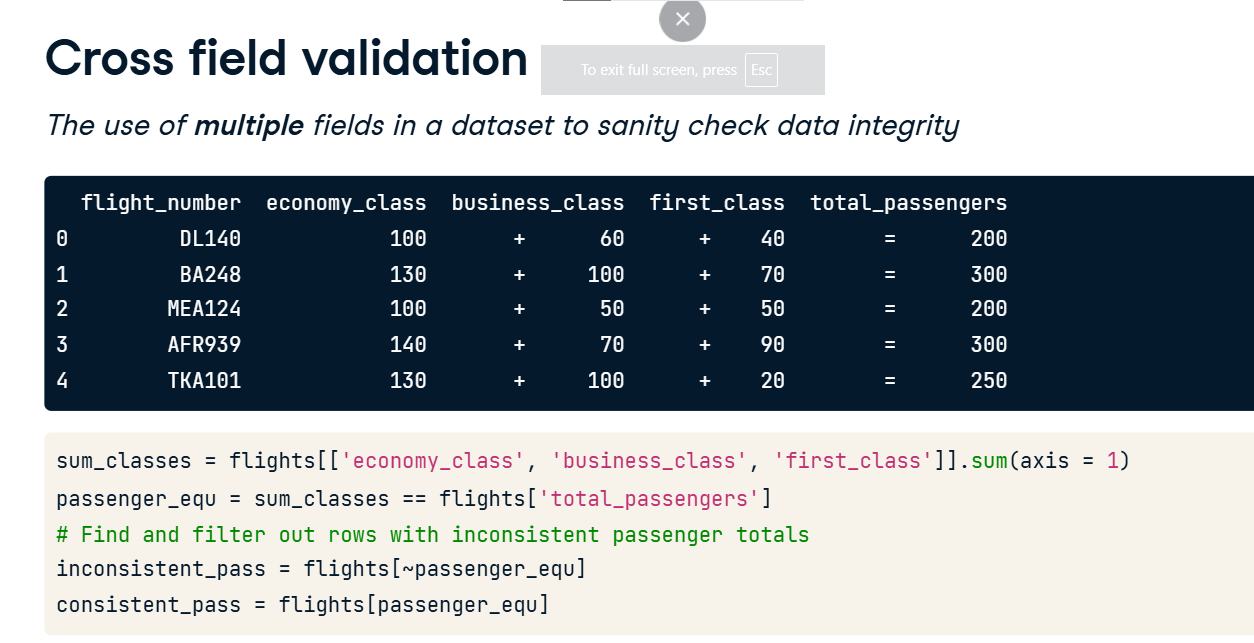
**Motivation**

Let's take a look at the following dataset. It contains flight statistics on the total number of passengers in economy, business and first class as well as the total passengers for each flight. We know that these columns have been collected and merged from different data sources, and a common challenge when merging data from different sources is data integrity, or more broadly making sure that our data is correct.



**Cross field validation**

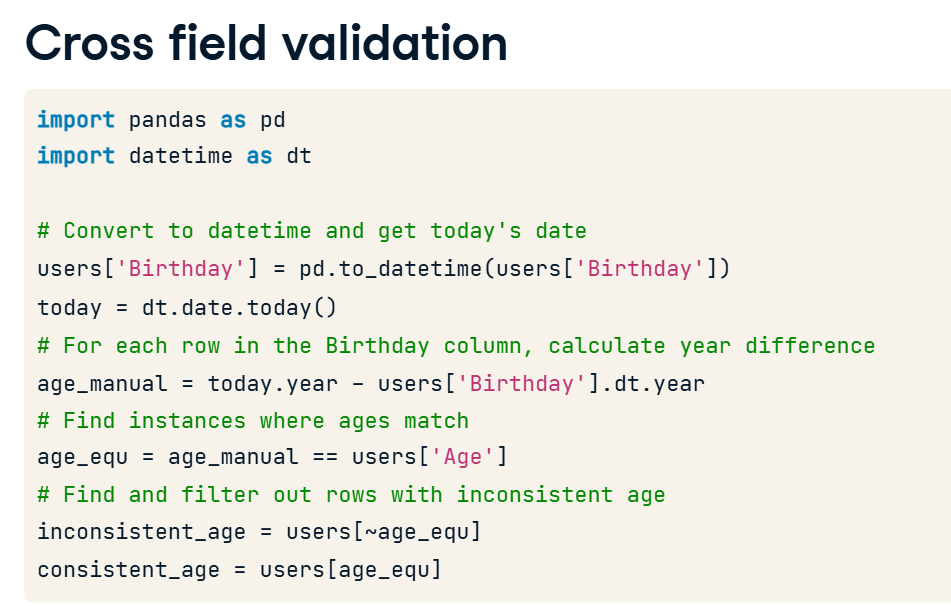
This is where cross field validation comes in. Cross field validation is the use of multiple fields in your dataset to sanity check the integrity of your data. For example in our flights dataset, this could be summing economy, business and first class values and making sure they are equal to the total passengers on the plane. This could be easily done in Pandas, by first subsetting on the columns to sum, then using the sum method with the axis argument set to 1 to indicate row wise summing. We then find instances where the total passengers column is equal to the sum of the classes. And find and filter out instances of inconsistent passenger amounts by subsetting on the equality we created with brackets and the tilde symbol.



Here's another example containing user IDs, birthdays and age values for a set of users. We can for example make sure that the age and birthday columns are correct by subtracting the number of years between today's date and each birthday.

**Cross field validation**

We can do this by first making sure the Birthday column is converted to datetime with the pandas to datetime function. We then create an object storing today's date using the datetime package's date dot today function. We then calculate the difference in years between today's date's year, and the year of each birthday by using the dot dt dot year attribute of the user's Birthday column. We then find instances where the calculated ages are equal to the actual age column in the users DataFrame. We then find and filter out the instances where we have inconsistencies using subsetting with brackets and the tilde symbol on the equality we created.



**What to do when we catch inconsistencies?**

So what should be the course of action in case we spot inconsistencies with cross-field validation? Just like other data cleaning problems, there is no one size fits all solution, as often the best solution requires an in depth understanding of our dataset. We can decide to either drop inconsistent data, set it to missing and impute it, or apply some rules due to domain knowledge. All these routes and assumptions can be decided upon only when you have a good understanding of where your dataset comes from and the different sources feeding into it.