**Uniformity**

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

**In this chapter**

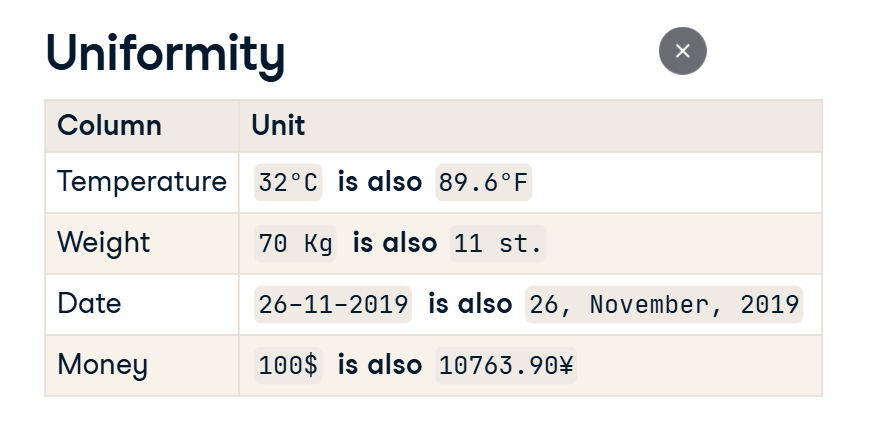
In this chapter, we're looking at more advanced data cleaning problems, such as uniformity, cross field validation and dealing with missing data.

**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.

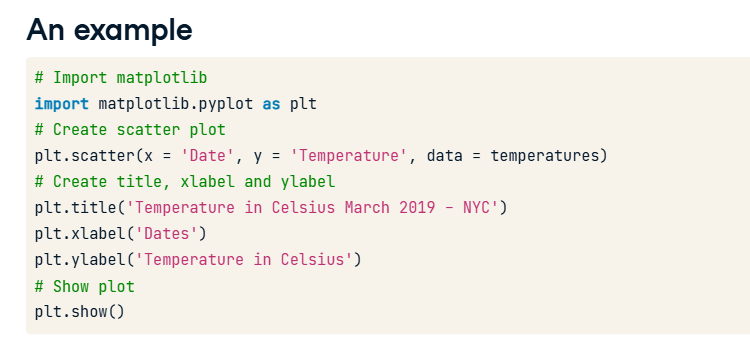
**Uniformity**

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.

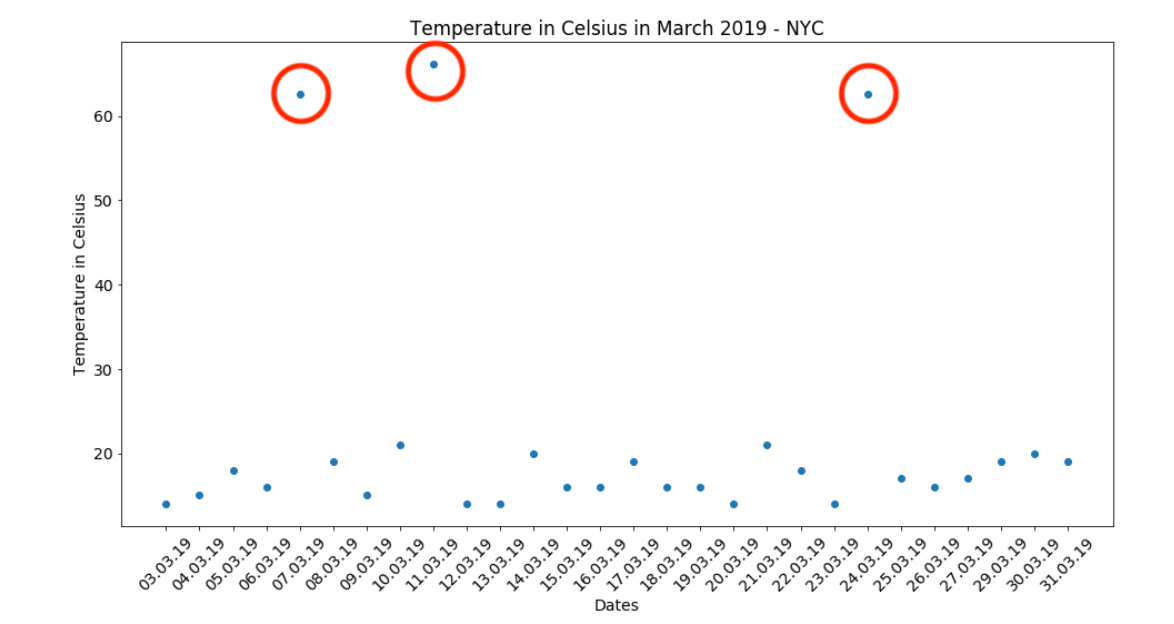


**An example**

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,

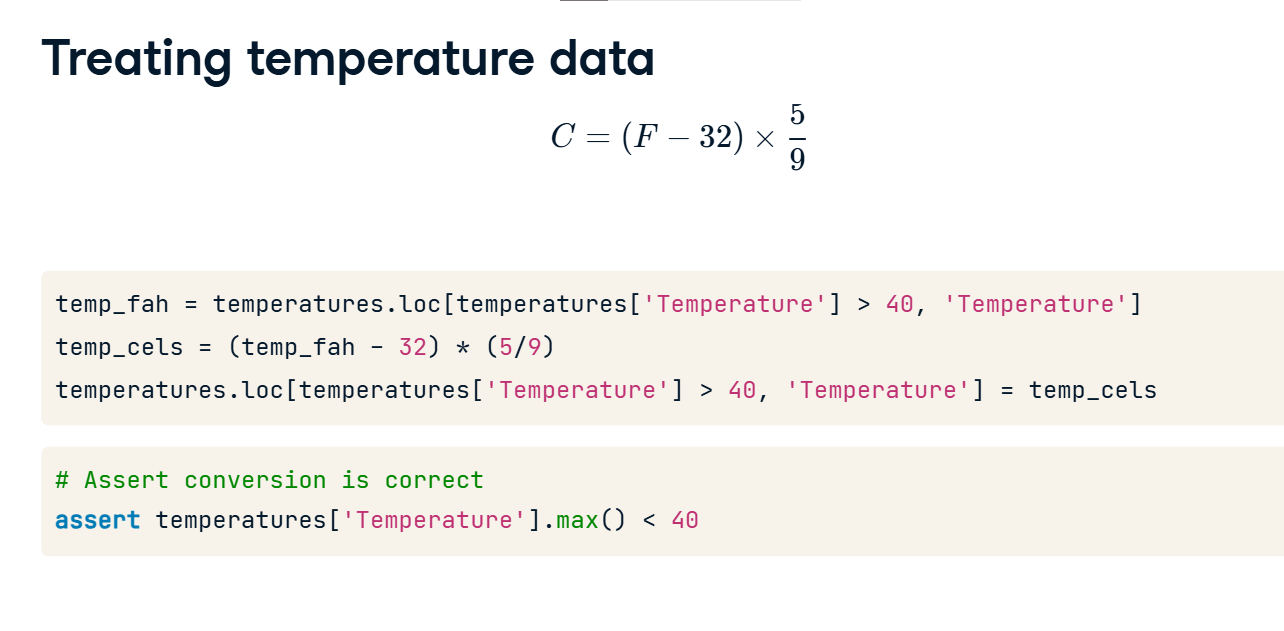
and voila.

Notice these values here? They all must be fahrenheit.



**Treating temperature data**

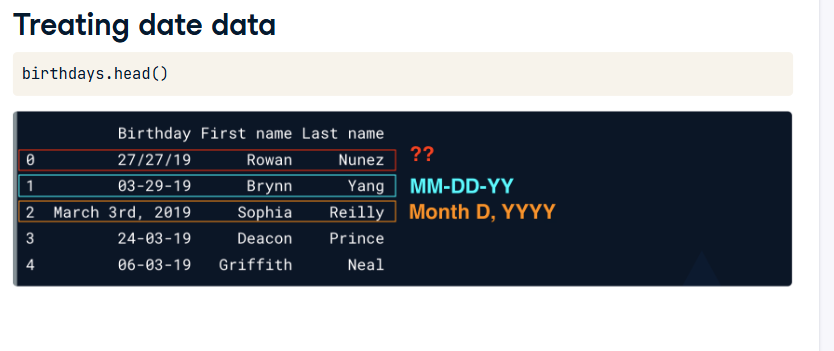
A simple web search returns the formula for converting Fahrenheit to Celsius. To convert our temperature data, we isolate all rows of temperature column where it is above 40 using the loc method. We chose 40 because it's a common sense maximum for Celsius temperatures in New York City. We then convert 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**

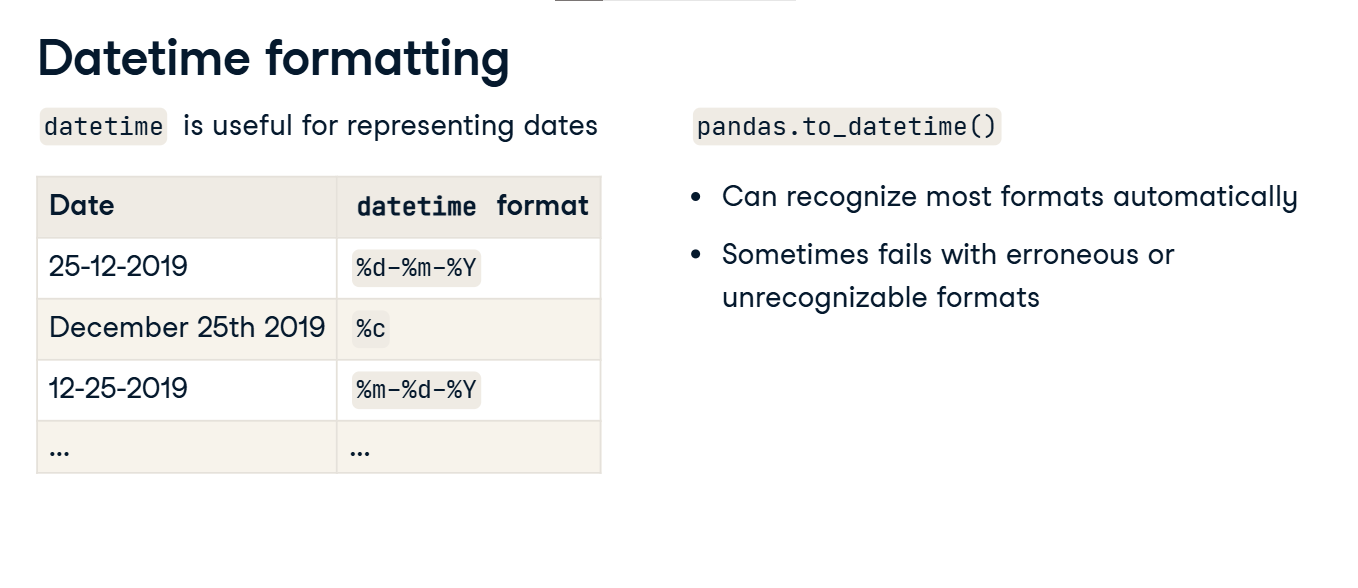
Here's another common uniformity problem with date data. This is a DataFrame called birthdays containing birth dates for a variety of individuals. It has been collected from a variety of sources and merged into one.

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!

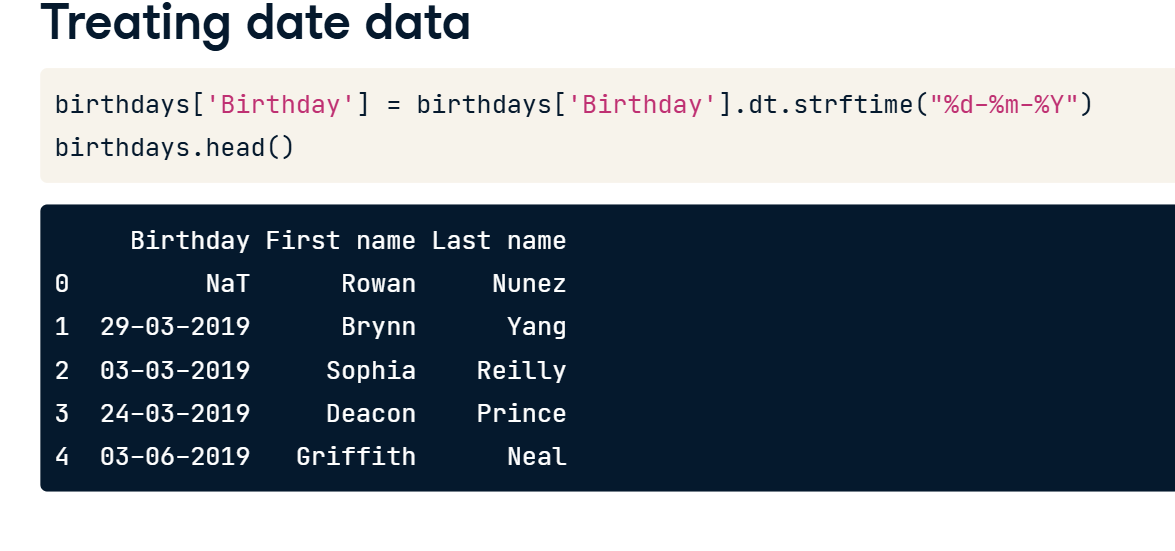


**Treating date data**

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.

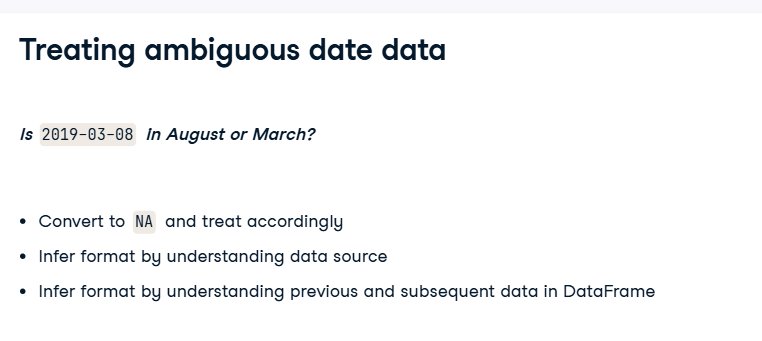


**Treating date data**

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.

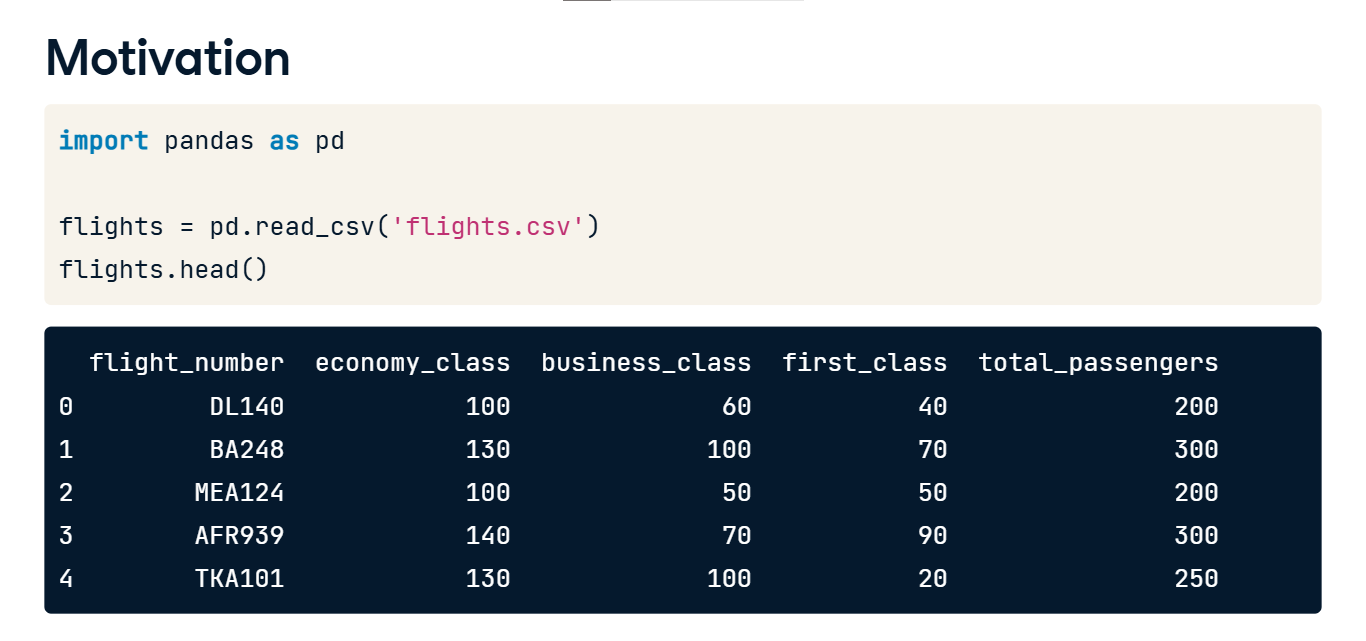


**Cross field validation**

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

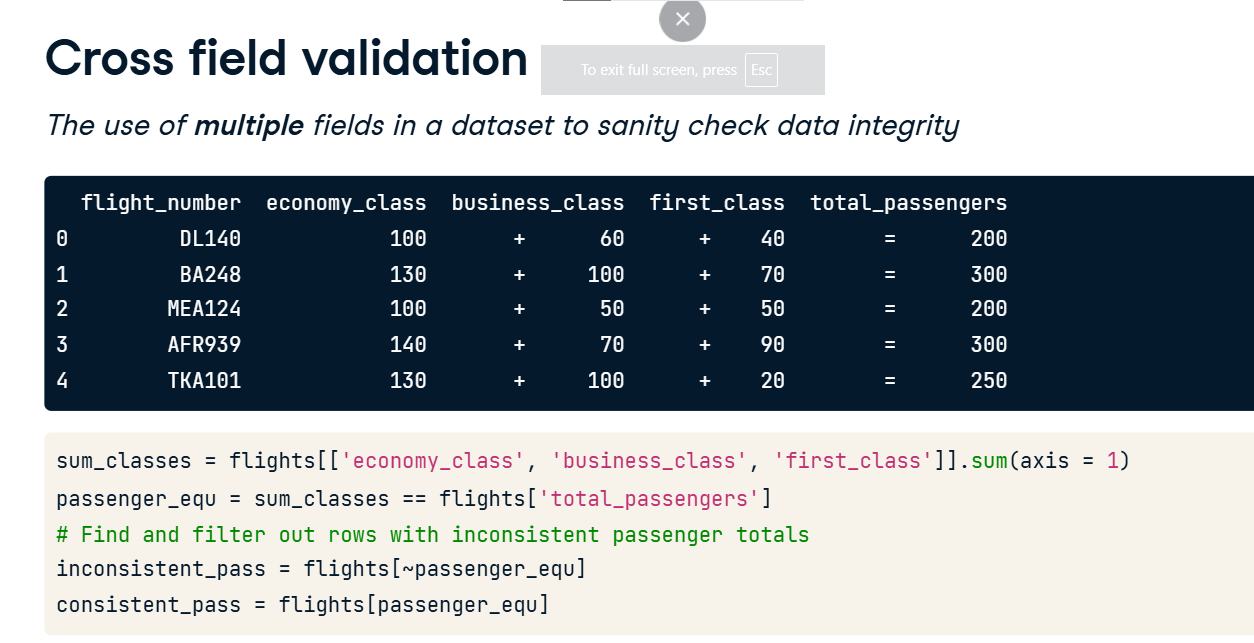
**Motivation**

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**

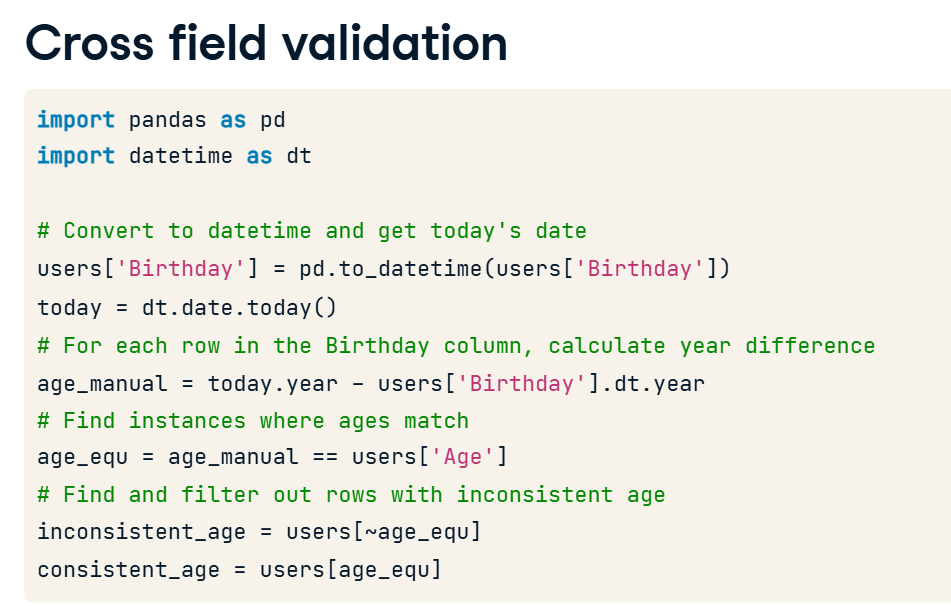
This is where cross field validation comes in. Cross field validation is the use of multiple fields in your dataset to sanity check the integrity of your data. For example in our flights dataset, this could be summing economy, business and first class values and making sure they are equal to the total 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.

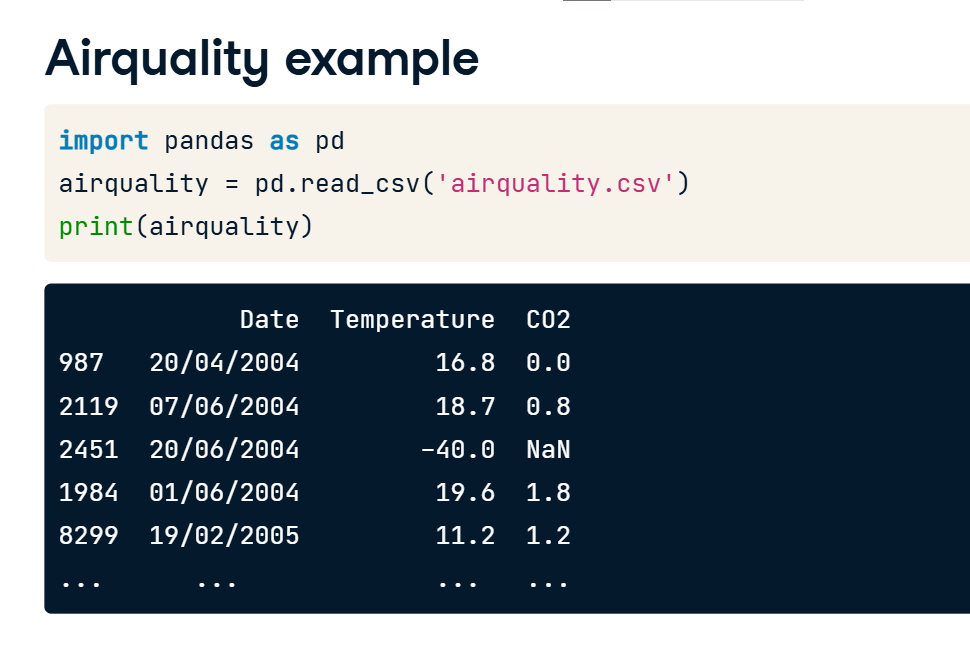
**Cross field validation**

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.



**Completeness**

Hi and welcome to the last lesson of this chapter. In this lesson, we're going to discuss completeness and missing data.

**What is missing data?**

Missing data is one of the most common and most important data cleaning problems. Essentially, missing data is when no data value is stored for a variable in an observation. Missing data is most commonly represented as NA or NaN, but can take on arbitrary values like 0 or dot. Like a lot of the problems that we've seen thus far in the course, it's commonly due to technical or human errors. Missing data can take many forms, so let's take a look at an example.

**Airquality example**

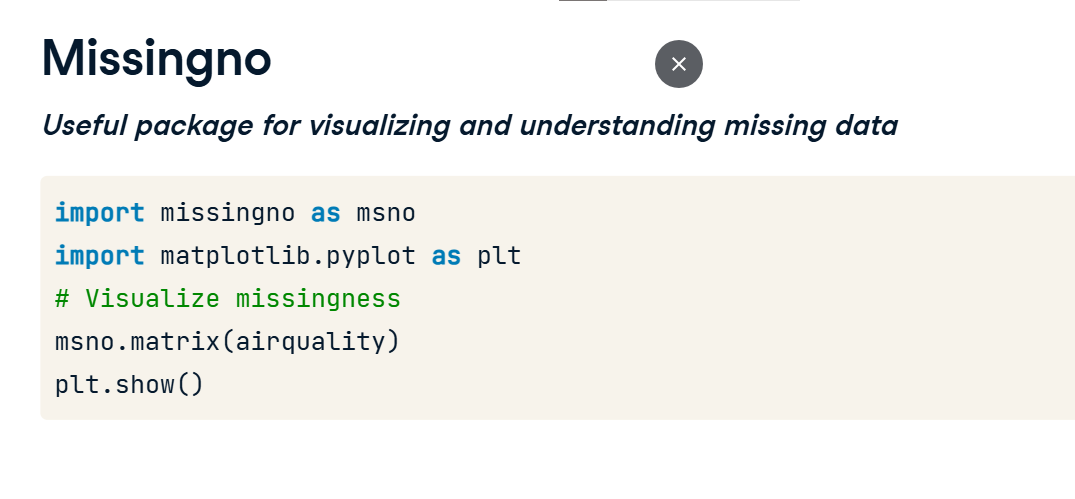
Let's take a look at the airquality dataset. It contains temperature and CO2 measurements for different dates.

**Airquality example**

We can see that the CO2 value in this row is represented as NaN

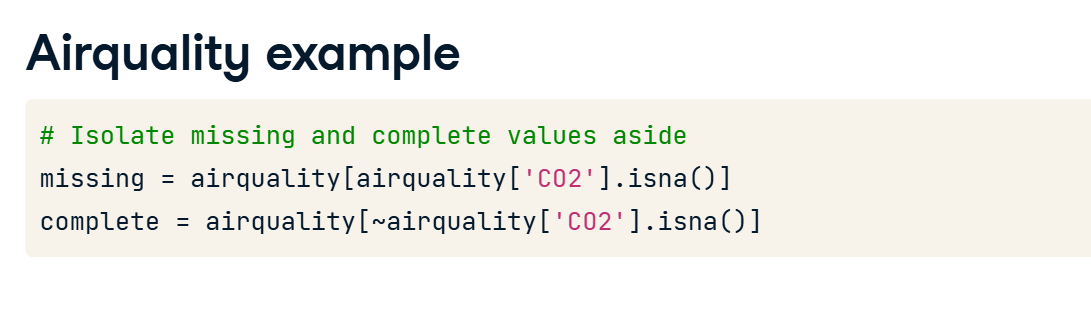


We can find rows with missing values by using the dot is na method, which returns True for missing values and False for complete values across all our rows and columns.



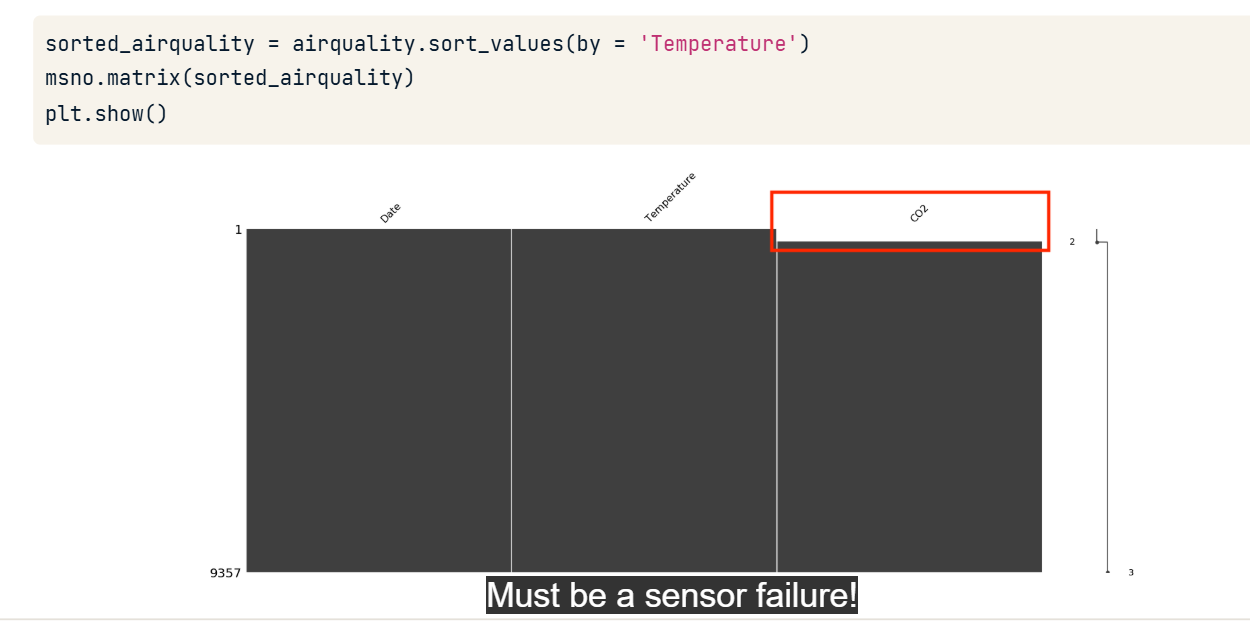
**Airquality example**

We can also chain the isna method with the sum method, which returns a breakdown of missing values per column in our dataframe. We notice that the CO2 column is the only column with missing values - let's find out why and dig further into the nature of this missingness by first visualizing our missing values.



**Missingno**

The missingno package allows to create useful visualizations of our missing data. Digging into its details is not part of the course, but you can also check out other courses on missing data in DataCamp's course library. We visualize the missingness of the airquality DataFrame with the msno dot matrix function, and show it with pyplot's show function from matplotlib, which returns



**Insert title here...**

the following image. This matrix essentially shows how missing values are distributed across a column. We see that missing CO2 values are randomly scattered throughout the column, but is that really the case? Let's dig deeper.

**Airquality example**

We first isolate the rows of airquality with missing CO2 values in one DataFrame, and complete CO2 values in another.

Then, let's use the describe method on each of the created DataFrames.

We see that for all missing values of CO2, they occur at really low temperatures, with the mean temperature at minus 39 degrees and a minimum and maximum of -49 and -30 respectively. Let's confirm this visually with the missngno package.

**Insert title here...**

We first sort the DataFrame by the temperature column. Then we input the sorted dataframe to the matrix function from msno. This leaves us with this matrix.

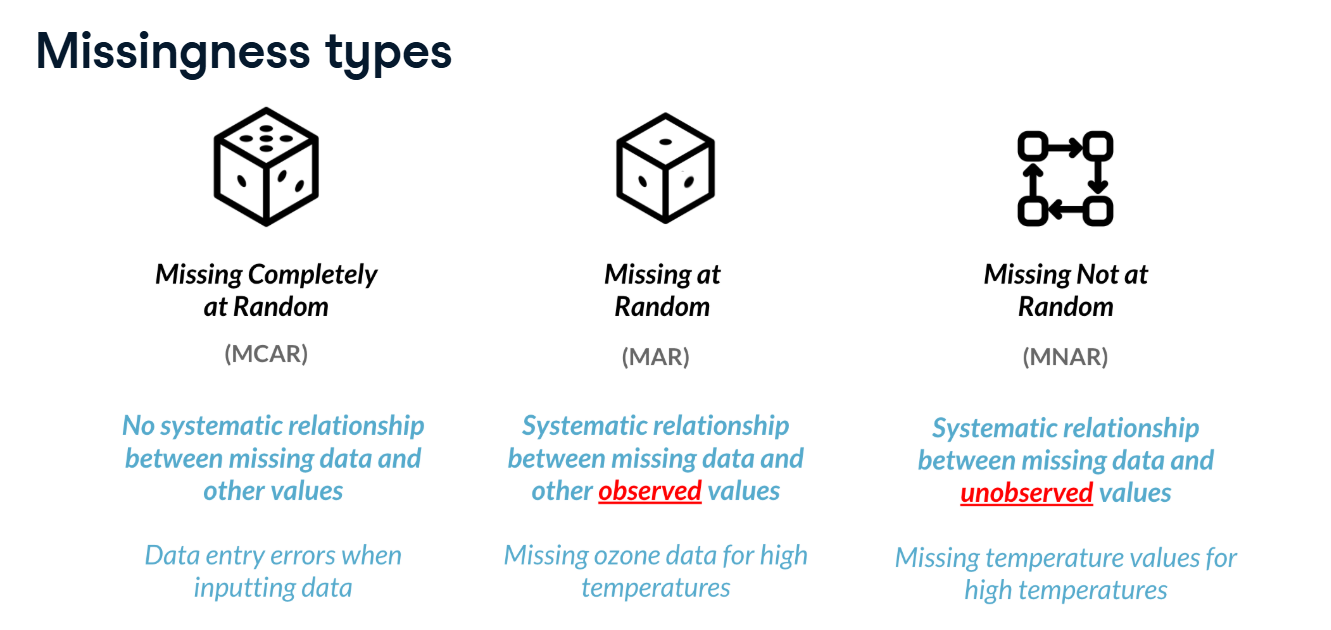
**Insert title here...**

Notice how all missing values are on the top? This is because values are sorted from smallest to largest by default. This essentially confirms that CO2 measurements are lost for really low temperatures. Must be a sensor failure!

**Missingness types**

This leads us to missingness types. Without going too much into the details, there are a variety of types of missing data. It could missing completely at random, missing at random, or missing not at random.

Missing completely at random data is when there missing data completely due to randomness, and there is no relationship between missing data and remaining values, such data entry errors.



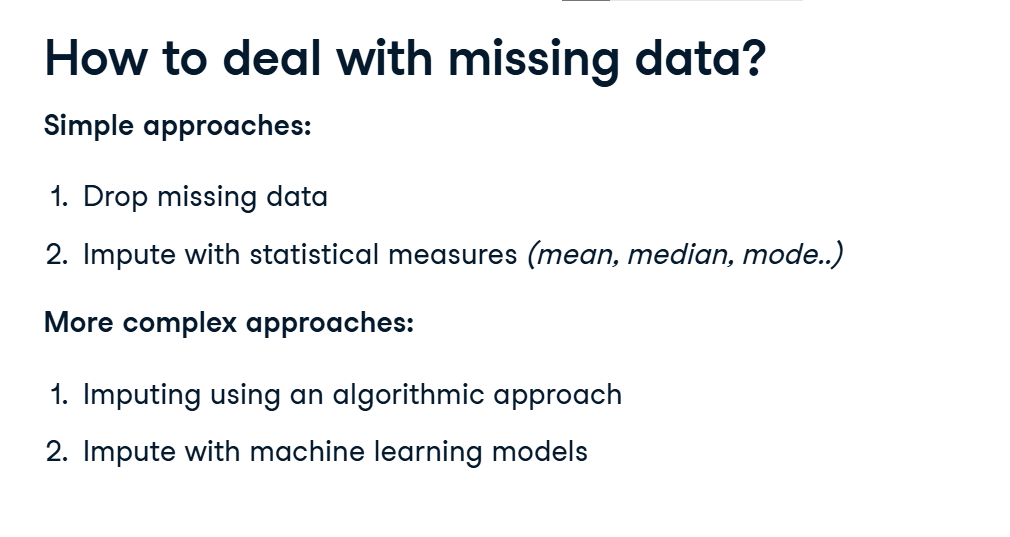
**Missingness types**

Despite a slightly deceiving name, Missing at random data is when there is a relationship between missing data and other observed values, such as our CO2 data being missing for low temperatures.

When data is missing not at random, there is a systematic relationship between the missing data and unobserved values. For example, when it's really hot outside, the thermometer might stop working, so we don't have temperature measurements for days with high temperatures. However, we have no way to tell this just from looking at the data since we can't actually see what the missing temperatures are.

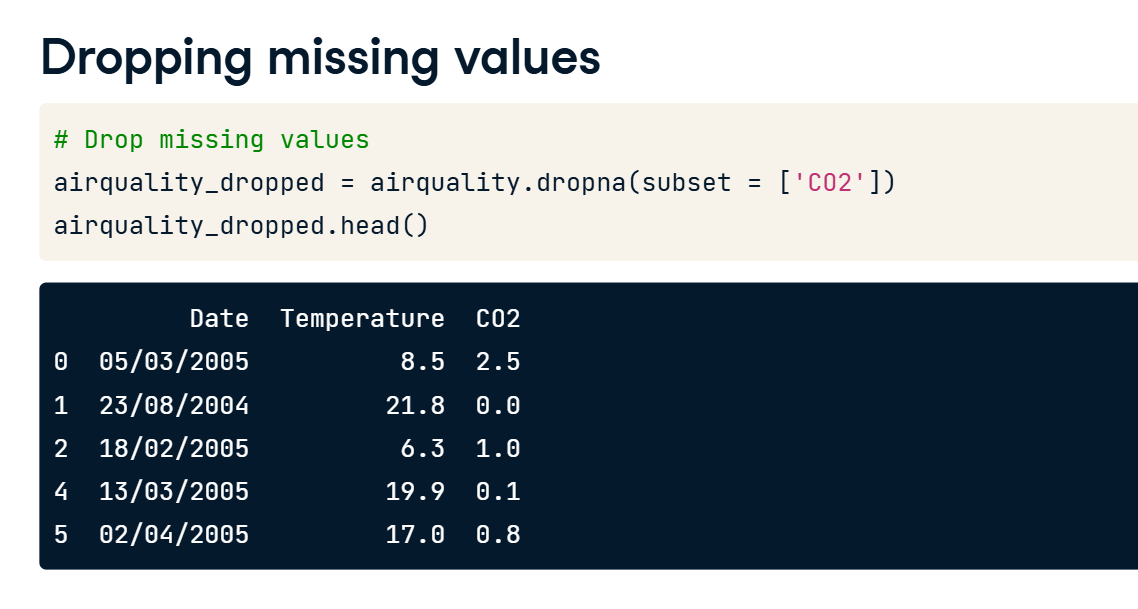
**How to deal with missing data?**

There's a variety of ways of dealing with missing data, from dropping missing data, to imputing them with statistical measures such as mean, median or mode, or imputing them with more complicated algorithmic approaches or ones that require some machine learning. Each missingness type requires a specific approach, and each type of approach has drawbacks and positives, so make sure to dig deeper in DataCamp's course library on dealing with missing data.



**Dealing with missing data**

In this lesson, we'll just explore the simple approaches to dealing with missing data. Let's grab another look at the header of airquality.



**Dropping missing values**

04:07 - 04:16We can drop missing values, by using the dot dropna method, alongside the subset argument which lets us pick which column's missing values to drop.

**Replacing with statistical measures**

04:16 - 04:39

We can also replace the missing values of CO2 with the mean value of CO2, by using the fillna method, which is in this case 1.73. Fillna takes in a dictionary with columns as keys, and the imputed value as values. We can even feed custom values into fillna pertaining to our missing data if we have enough domain knowledge about our dataset.

