**Data type constraints**

Hi and welcome! My name is Adel, and I'll be your host as we learn how to clean data in Python.

**Course outline**

In this course, we're going to understand how to diagnose different problems in our data and how they can can come up during our workflow.

We will also understand the side effects of not treating our data correctly.and various ways to address different types of dirty data.

In this chapter, we're going to discuss the most common data problems you may encounter and how to address them. So let's get started!

**Why do we need to clean data?**

To understand why we need to clean data, let's remind ourselves of the data science workflow. In a typical data science workflow, we usually access our raw data, explore and process it, develop insights using visualizations or predictive models, and finally report these insights with dashboards or reports.

**Why do we need to clean data?**

Dirty data can appear because of duplicate values, mis-spellings, data type parsing errors and legacy systems.

**Why do we need to clean data?**

Without making sure that data is properly cleaned in the exploration and processing phase, we will surely compromise the insights and reports subsequently generated. As the old adage says, garbage in garbage out.

**Data type constraints**

When working with data, there are various types that we may encounter along the way. We could be working with text data, integers, decimals, dates, zip codes, and others. Luckily, Python has specific data type objects for various data types that you're probably familiar with by now. This makes it much easier to manipulate these various data types in Python. As such, before preparing to analyze and extract insights from our data, we need to make sure our variables have the correct data types, other wise we risk compromising our analysis.

**Strings to integers**

Let's take a look at the following example. Here's the head of a DataFrame containing revenue generated and quantity of items sold for a sales order. We want to calculate the total revenue generated by all sales orders. As you can see, the Revenue column has the dollar sign on the right hand side. A close inspection of the DataFrame column's data types using the dot-dtypes attribute returns object for the Revenue column, which is what pandas uses to store strings.

**String to integers**

We can also check the data types as well as the number of missing values per column in a DataFrame, by using the dot-info() method.

Since the Revenue column is a string, summing across all sales orders returns one large concatenated string containing each row's string. To fix this, we need to first remove the $ sign from the string so that pandas is able to convert the strings into numbers without error. We do this with the dot-str-dot-strip() method, while specifying the string we want to strip as an argument, which is in this case the dollar sign. Since our dollar values do not contain decimals, we then convert the Revenue column to an integer by using the dot-astype() method, specifying the desired data type as argument. Had our revenue values been decimal, we would have converted the Revenue column to float. We can make sure that the Revenue column is now an integer by using the assert statement, which takes in a condition as input, as returns nothing if that condition is met, and an error if it is not.



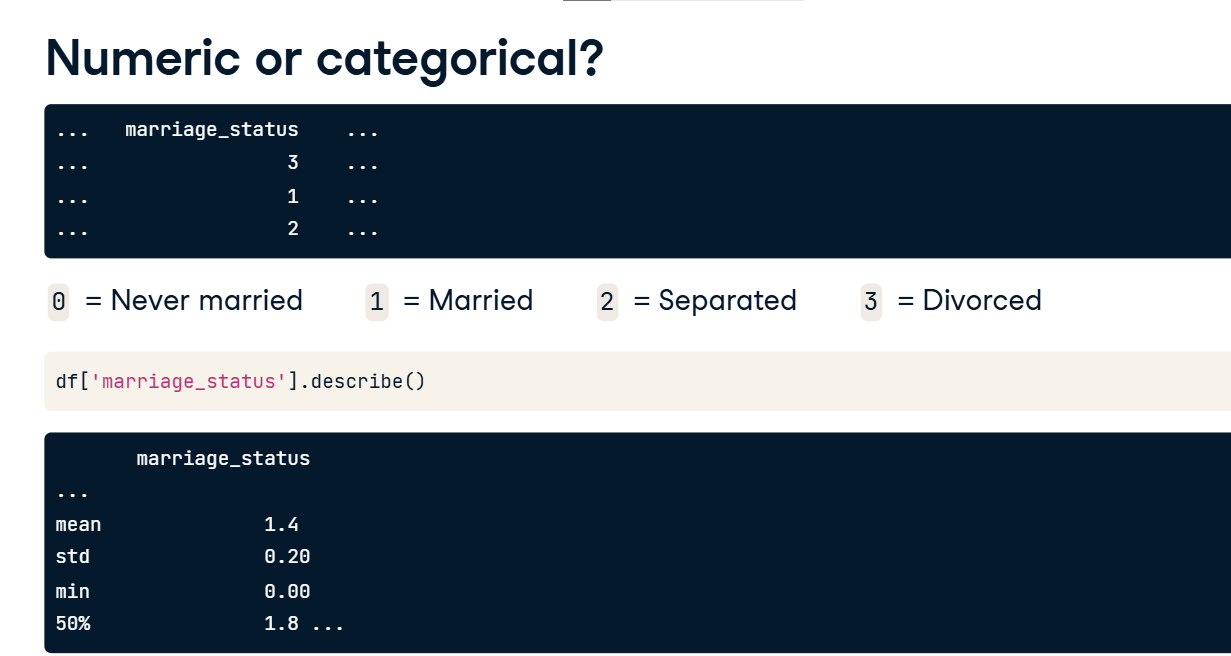
**The assert statement**

For example, here we are testing the equality that 1+1 equals 2. Since it is the case, the assert statement returns nothing. However, when testing the equality 1+1 equals 3, we receive an assertionerror. You can test almost anything you can imagine of by using assert, and we'll see more ways to utilize it as we go along the course.



**Numeric or categorical?**

A common type of data seems numeric but actually represents categories with a finite set of possible categories. This is called categorical data. We will look more closely at categorical data in Chapter 2, but let's take a look at this example. Here we have a marriage status column, which is represented by 0 for never married, 1 for married, 2 for separated, and 3 for divorced. However it will be imported of type integer, which could lead to misleading results when trying to extract some statistical summaries.



**Numeric or categorical?**

We can solve this by using the same dot-astype() method seen earlier, but this time specifying the category data type. When applying the describe again, we see that the summary statistics are much more aligned with that of a categorical variable, discussing the number of observations, number of unique values, most frequent category instead of mean and standard deviation.



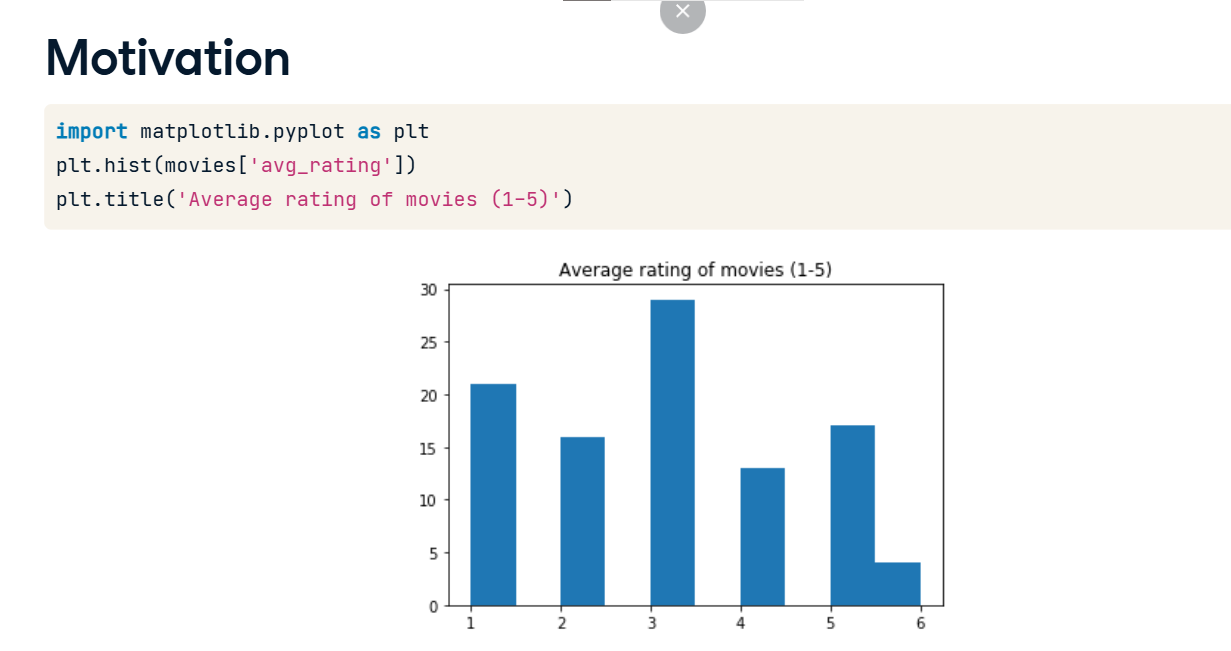
**Data range constraints**

Hi and welcome back! In this lesson, we're going to discuss data that should fall within a range.

**Motivation**

Let's first start off with some motivation. Imagine we have a dataset of movies with their respective average rating from a streaming service. The rating can be any integer between 1 an 5.

After creating a histogram with maptlotlib, we see that there are a few movies with an average rating of 6, which is well above the allowable range. This is most likely an error in data collection or parsing, where a variable is well beyond its range and treating it is essential to have accurate analysis.



**Motivation**

Here's another example, where we see subscription dates in the future for a service. Inherently this doesn't make any sense, as we cannot sign up for a service in the future, but these errors exist either due to technical or human error. We use the datetime package's dot-date-dot-today() function to get today's date, and we filter the dataset by any subscription date higher than today's date. We need to pay attention to the range of our data.

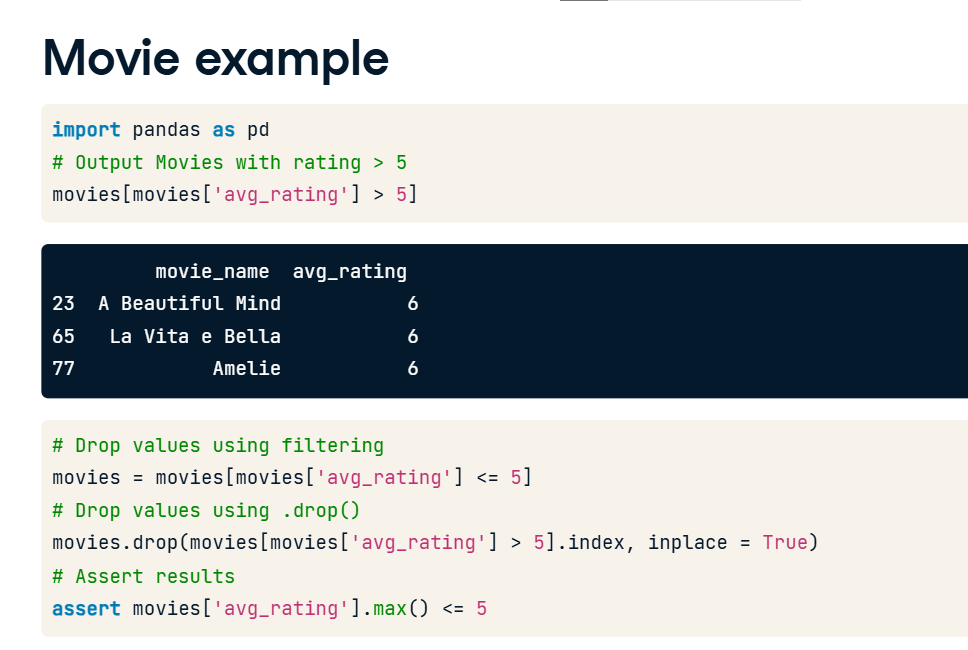


**How to deal with out of range data?**

There's a variety of options to deal with out of range data. The simplest option is to drop the data. However, depending on the size of your out of range data, you could be losing out on essential information. As a rule of thumb, only drop data when a small proportion of your dataset is affected by out of range values, however you really need to understand your dataset before deciding to drop values. Another option would be setting custom minimums or maximums to your columns. We could also set the data to missing, and impute it, but we'll take a look at how to deal with missing data in Chapter 3. We could also, dependent on the business assumptions behind our data, assign a custom value for any values of our data that go beyond a certain range.

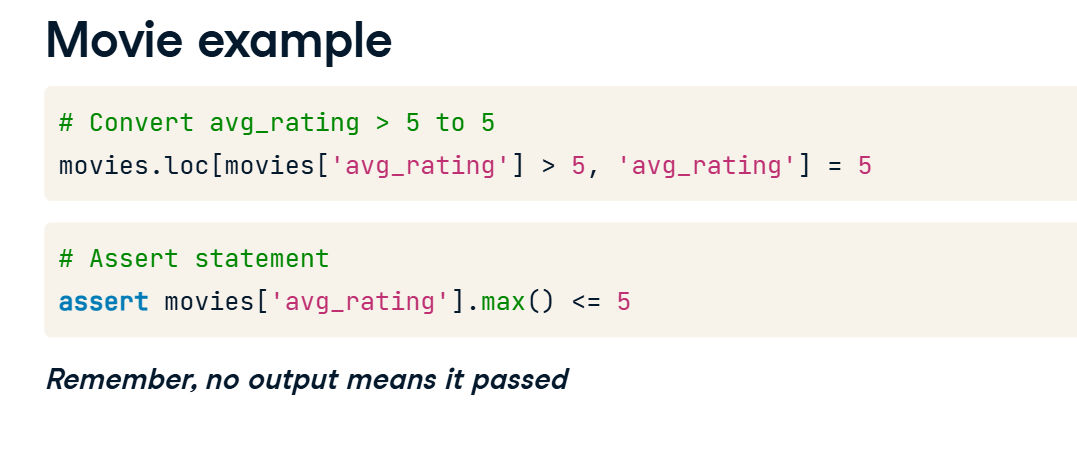
**Movie example**

Let's take a look at the movies example mentioned earlier. We first isolate the movies with ratings higher than 5. Now if these values are affect a small set of our data, we can drop them. We can drop them in two ways - we can either create a new filtered movies DataFrame where we only keep values of avg\_rating lower or equal than to 5. Or drop the values by using the drop method. The drop method takes in as argument the row indices of movies for which the avg\_rating is higher than 5. We set the inplace argument to True so that values are dropped in place and we don't have to create a new column. We can make sure this is set in place using an assert statement that checks if the maximum of avg\_rating is lower or equal than to 5.



**Movie example**

Depending on the assumptions behind our data, we can also change the out of range values to a hard limit. For example, here we're setting any value of the avg\_rating column in to 5 if it goes beyond it. We can do this using the dot-loc method, which returns all cells that fit a custom row and column index. It takes as first argument the row index, or here all instances of avg\_rating above 5 and as second argument the column index, which is here the avg\_rating column. Again, we can make sure that this change was done using an assert statement.



**Date range example**

Let's take another look at the date range example mentioned earlier, where we had subscriptions happening in the future. We first look at the data types of the column with the dot-dtypes attribute. We can confirm that the subscription\_date column is an object and not a date or datetime object. To compare a pandas object to a date, the first step is to convert it to another date. We do so by first converting it into a pandas datetime object with the to\_datetime function from pandas, which takes in as an argument the column we want to convert. We then need to convert the datetime object into a date. This conversion is done by appending dt-dot-date to the code. Could we have converted from an object directly to a date, without the pandas datetime conversion in the middle? Yes! But we'd have had to provide information about the date's format as a string, so it's just as easy to do it this way.



**Date range example**

Now that the column is a date, we can treat it in a variety of ways. We first create a today\_date variable using the datetime function date-dot-today, which allows us to store today's date. We can then either drop the rows with exceeding dates similar to how we did in the average rating example, or replace exceeding values with today's date. In both cases we can use the assert statement to verify our treatment went well, by comparing the maximum value in the subscription\_date column. However, make sure to chain it with the dot-date method to return a date instead of a timestamp.



**Uniqueness constraints**

Hi and welcome to the final lesson of this chapter. Let's discuss another common data cleaning problem, duplicate values.

**What are duplicate values?**

Duplicate values can be diagnosed when we have the same exact information repeated across multiple rows, for a some or all columns in our DataFrame. In this example DataFrame containing the names, address, height, and weight of individuals, the rows presented have identical values across all columns.

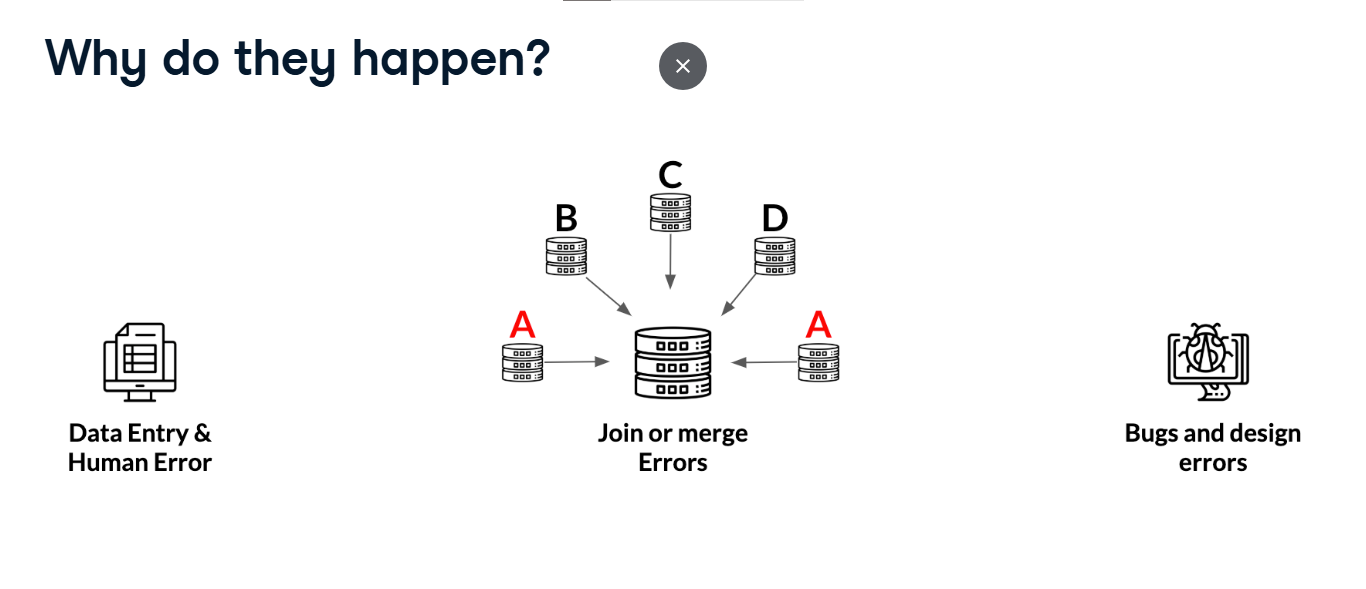
In this one, there are duplicate values for all columns except the height column -- which leads us to think it's more likely a data entry error than an actual other person.

**Why do they happen?**

Apart from data entry and human errors alluded to in the previous slide,

duplicate data can also arise because of bugs and design errors whether in business processes or data pipelines.

However they oftenmost arise from the necessary act of joining and consolidating data from various resources, which could retain duplicate values.

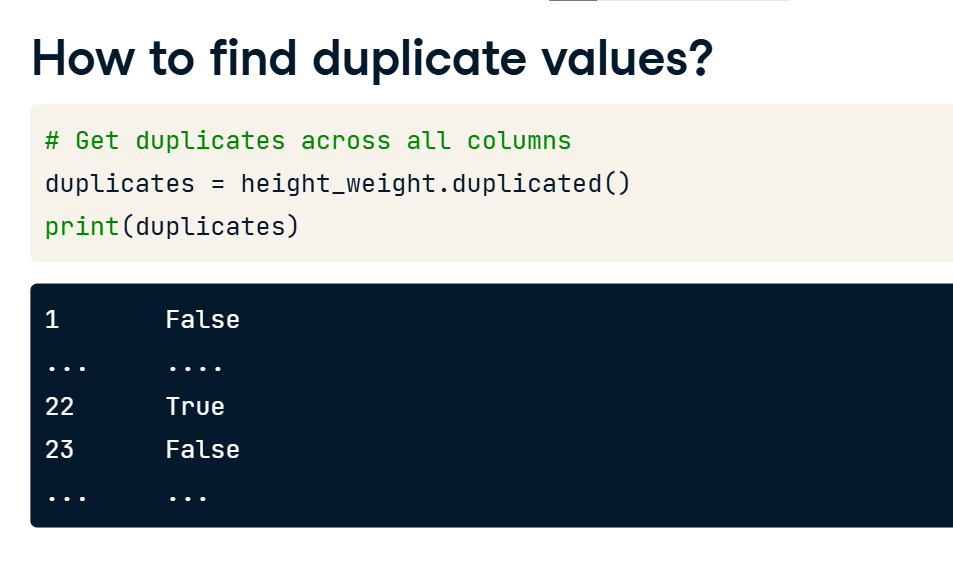


**How to find duplicate values?**

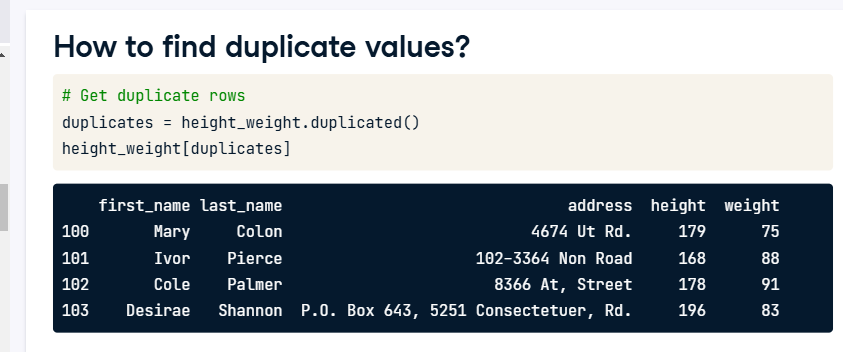
Let's first see how to find duplicate values. In this example, we're working with a bigger version of the the height and weight data seen earlier in the video.

**How to find duplicate values?**

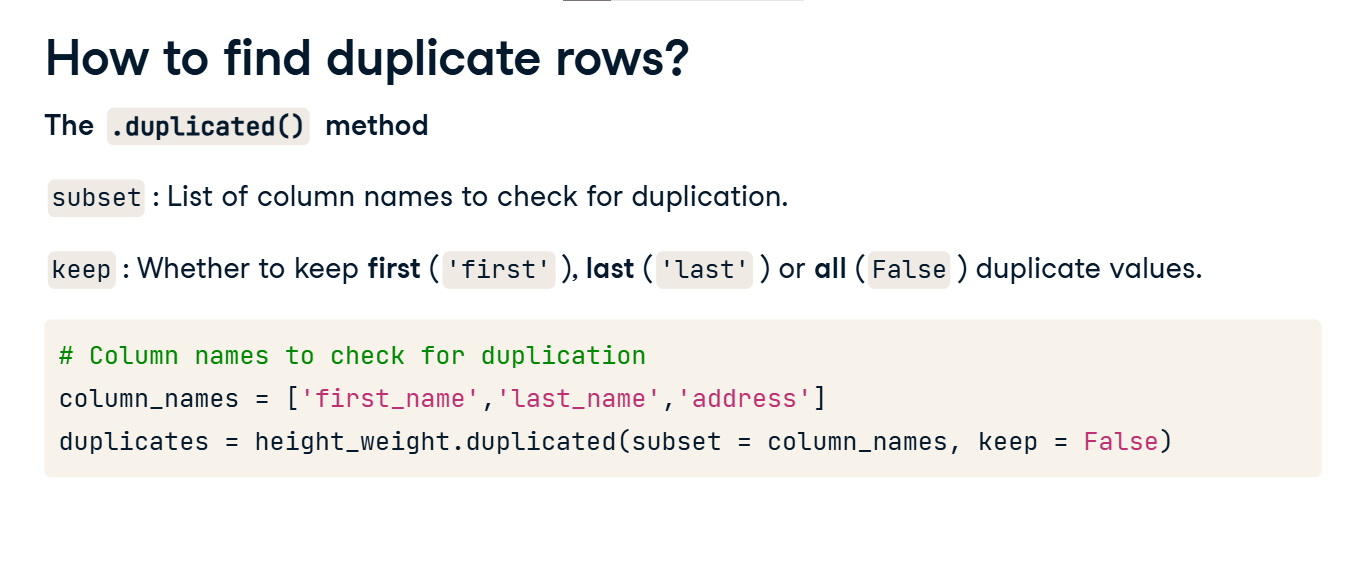
We can find duplicates in a DataFrame by using the dot-duplicated() method. It returns a Series of boolean values that are True for duplicate values, and False for non-duplicated values.



We can see exactly which rows are affected by using brackets as such. However, using dot-duplicated() without playing around with the arguments of the method can lead to misleading results, as all the columns are required to have duplicate values by default, with all duplicate values being marked as True except for the first occurrence. This limits our ability to properly diagnose what type of duplication we have, and how to effectively treat it.



To properly calibrate how we go about finding duplicates, we will use 2 arguments from the dot-duplicated() method. The subset argument lets us set a list of column names to check for duplication. For example, it allows us to find duplicates for the first and last name columns only. The keep argument lets us keep the first occurrence of a duplicate value by setting it to the string first, the last occurrence of a duplicate value by setting it the string last, or keep all occurrences of duplicate values by setting it to False. In this example, we're checking for duplicates across the first name, last name, and address variables, and we're choosing to keep all duplicates.



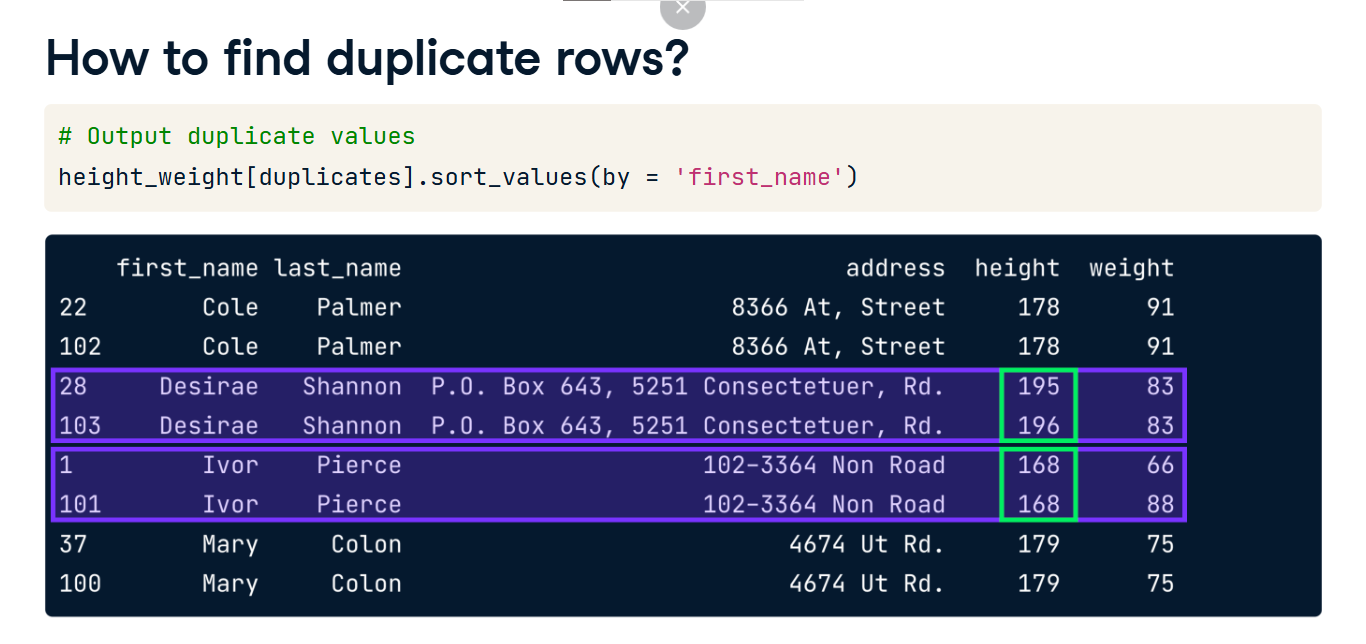
**How to find duplicate rows?**

We see the following results -- to get a better bird's eye view of the duplicates,

We sort the duplicate rows using the dot-sort\_values method, choosing first\_name to sort by.

We find that there are four sets of duplicated rows, the first 2 being complete duplicates of each other across all columns, highlighted here in red.

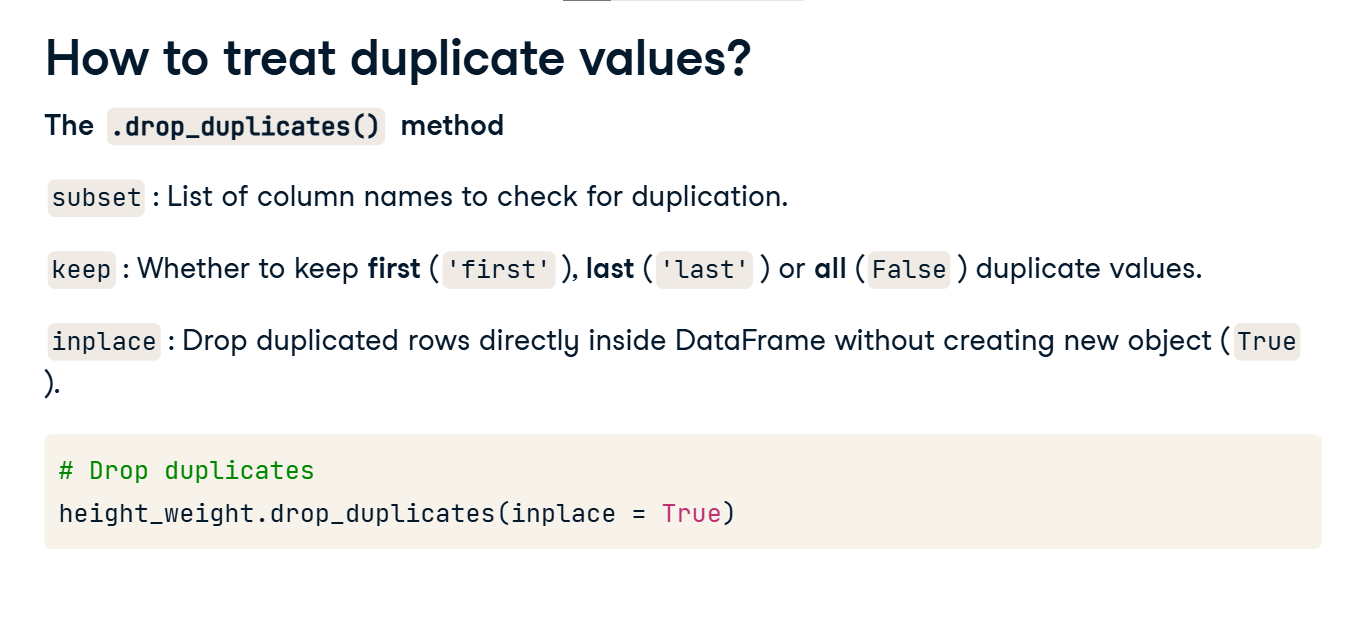
The other 2 being incomplete duplicates of each other highlighted here in blue with discrepancies across height and weight respectively.



**How to treat duplicate values?**

The complete duplicates can be treated easily. All that is required is to keep one of them only and discard the others.

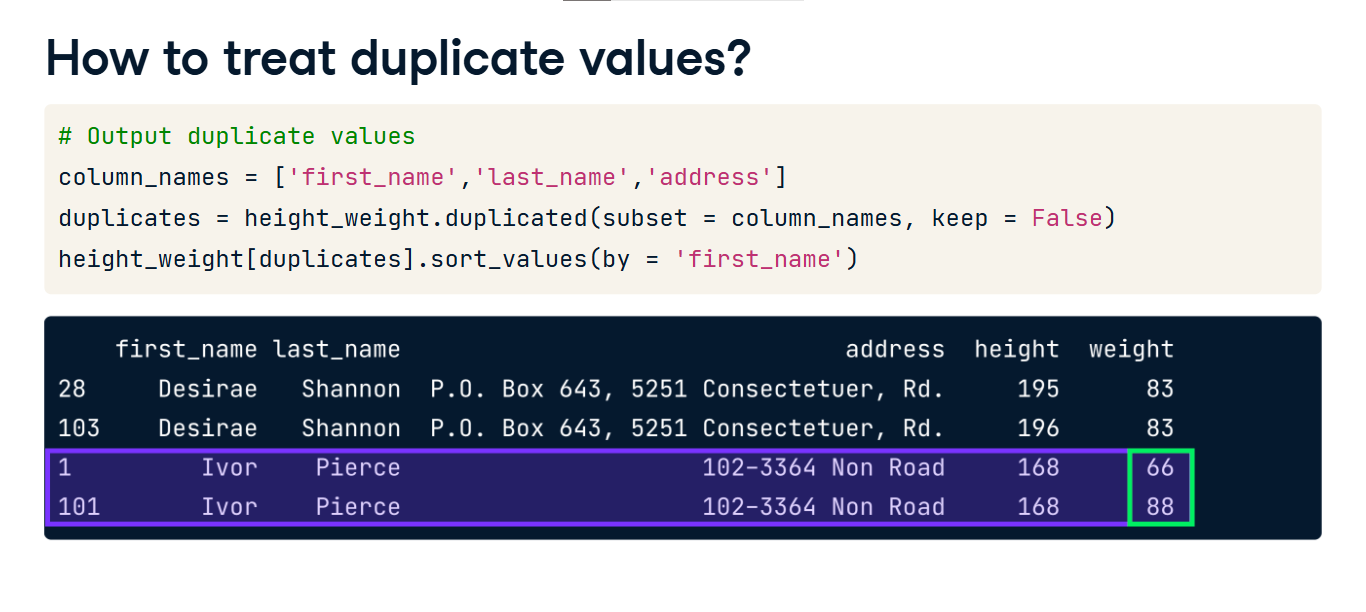
This can be done with the dot-drop\_duplicates() method, which also takes in the same subset and keep arguments as in the dot-duplicated() method, as well as the inplace argument which drops the duplicated values directly inside the height\_weight DataFrame. Here we are dropping complete duplicates only, so it's not necessary nor advisable to set a subset, and since the keep argument takes in first as default, we can keep it as such. Note that we can also set it as last, but not as False as it would keep all duplicates.



**How to treat duplicate values?**

This leaves us with the other 2 sets of duplicates discussed earlier, which are the same for first\_name, last\_name and address, but contain discrepancies in height and weight. Apart from dropping rows with really small discrepancies, we can use a statistical measure to combine each set of duplicated values.

For example, we can combine these two rows into one by computing the average mean between them, or the maximum, or other statistical measures, this is highly dependent on a common sense understanding of our data, and what type of data we have.



We can do this easily using the groupby method, which when chained with the agg method, lets you group by a set of common columns and return statistical values for specific columns when the aggregation is being performed. For example here, we created a dictionary called summaries, which instructs groupby to return the maximum of duplicated rows for the height column, and the mean duplicated rows for the weight column. We then group height\_weight by the column names defined earlier, and chained it with the agg method, which takes in the summaries dictionary we created. We chain this entire line with the dot-reset\_index() method, so that we can have numbered indices in the final output. We can verify that there are no more duplicate values by running the duplicated method again, and use brackets to output duplicate rows.

