Pandas Playbook: Manipulating Data

by Reindert-Jan Ekker

Course Overview

Course Overview

Hi everyone, my name is Reindert-Jan Ekker, and welcome to this playbook about manipulating data with Pandas. I'm a senior developer and freelance educator, and in this course I'll teach you about Pandas, the most popular Python framework for doing data science and analysis. The role of Pandas in data analysis has been growing hard over the past few years, and you simply cannot go without Pandas anymore. This course goes over the core tasks that you will need to perform when working with any real-world dataset, and we'll do so in a very hands-on way. Some of the major topics that we will cover include exploring a new dataset; selecting, sorting, and filtering your data so that you can drill down into your dataset to answer specific questions; cleaning a dataset, which means doing things like fixing bad or missing data points and removing outliers; and transforming your data either by doing calculations on it, or by changing the structure of your dataset. By the end of this course, you'll have a good understanding of the core functionality of a Pandas DataFrame, and you'll be able to handle most everyday tasks. Before beginning the course, you should be familiar with the very basics of Python and data science. I hope you'll join me on this journey to learn how to manipulate your data with Pandas with the Pandas Playbook: Manipulating Data at Pluralsight.

Course Introduction

Course Introduction and Overview

Hi, my name is Reindert-Jan Ekker, and welcome to this course about data manipulation with Pandas. Pandas is not just one of the most popular software packages for data analysis, it's also, without a doubt, the most convenient and fun way to work with your data. In this course, I will cover the most important core functionalities of Pandas, so that after watching this course you will be ready for just about any data wrangling job that you might come across. In this short introduction module, I'll tell you what to expect from this course, what you should already know, and I'll give you an overview of the structure of the course. So, this course focuses on core Pandas functionality for data manipulation, and we will look at that functionality in the context of very common everyday tasks and problems. Just about all of the functionality we will explore is made available through the two main Pandas classes, the Pandas Series and the Pandas DataFrame. There's also several things that you will not find in this course. Fist of all, this is not a general Pandas introduction. I will assume a basic knowledge of what a DataFrame and a Series are. If you do not know the very basics, there's a great intro course here on Pluralsight that you may want to see first. I'm not going to dive deeply into data visualization because that's a completely different topic from the core data manipulation tasks I want to talk about in this course. The same is true for machine learning, or all the various ways to do input and output with Pandas. Furthermore, there are some Pandas features that you may consider important, or even core features, but that I have chosen not to cover in this course because in my opinion they are slightly too involved. I'm talking about such things as categoricals, advanced indexing, or time series. So, what should you already know before watching this? Well, of course, you need some basic Python knowledge. At the very least, you should have an understanding of the built-in data structures like lists and dicts, you should be able to write functions, and you should have a minimal understanding of what classes and objects are. You should also have at least a tiny bit of knowledge of Pandas. Let's say you have played around with it a little and you have an understanding of what a Series, a DataFrame, and an Index are. Finally, there's the Jupyter Notebook, which is a tool I will use to write and run my code. You should have this installed and running, and you should know how to work with Notebook cells, write, edit, and run code. Now in case this is new for you, there's a Pluralsight intro course about Jupyter as well. By the way, my tip to you, install Anaconda from anaconda. org. This will install Jupyter, NumPy, and Pandas, as well as a wealth of other popular packages for data science and analysis. In a moment we'll start with the next module called

Exploring Data, and it's about the first steps you take when you start working with a new data set. How much data do you have? What do the rows and columns look like? We'll also do some basic statistical exploration of our data. Then we'll move on to a module called Selecting, Filtering, and Sorting, and this module focuses on getting information out of your data set. Basically, it's about asking the right questions and drilling down into your data set. The third module is about cleaning a data set. Just about every real-world data set will have some problems, and this can range from missing data to unwanted data or outliers, and we'll see various ways to deal with those things. And the final module is about transforming our data. So here we'll see how to run Python functions against our data, including functions we write ourselves, using a very cool and powerful feature called groupby, changing the structure of our columns and rows and combining multiple DataFrames into one. Very well, that's it for this introduction. Let's move on to the next module called Exploring data.

Exploring Data

Module Overview

Hi, in this module we'll look at the first steps for a data scientist when working with new datasets, exploring it to get a feeling for the kind of data we have, the statistical properties it has, and more. Whenever you start to work with a new dataset, the first thing you normally want to do is to explore what the data looks like. How many items are there in the set, and what kind of measurements are there? To get a feeling for the dataset, you usually also want to see a quick overview of basic statistics like the mean, max, and minimum values, etc., etc., and maybe even make a few simple plots while you're at it. So this is exactly what I'll do in this module. I'll show you how to get up and running with a new dataset.

Demo: Exploring a Dataset

So now let's start off with a short demo. We'll look at some of the core Pandas features that we can use to get a first impression of the contents of a new dataset, including the number of rows and columns and their data types, as well as a preview of the actual data itself. I've started a Jupyter Notebook and navigated to the directory where I keep the data for this course. All files I use will be available to download, so here you can see a file called weather. csv, which I have prepared so that it's really easy to read with Pandas. So starting a notebook here in this directory,

I select Python 3, and in the new notebook let's start by setting the title and calling it Exploring a dataset. Then I start the notebook by loading the essentials I always need. I import numpy as np, pandas as pd, and I say matplotlib inline, which enables us to view graphical output in the notebook. Then I press Shift+Enter to execute the cell. Next let's read the weather data from the file. To do so, I call the pd. read_csv method with the name of the weather. csv file. I make a new variable, df, which is short for DataFrame, and in there I store the data from csv. Now I've taken care to make it so that weather. csv contains data that we can read with this simple function call. In real-world situations, reading your data will probably be slightly more complicated, but for now I'm going to ignore that and just assume that you're getting your data loaded into a pandas DataFrame successfully. Let's go ahead and see what this DataFrame looks like. I might start by saying df. shape, and this is a property, not a function, and it contains a tuple telling us the size of the DataFrame. So now we know that there are 8784 rows and 5 columns to the DataFrame. But kinds of data are in there? Let's call df. info. And this function call gives us a little more information. It tells us that our df object is of class DataFrame, it has 5 columns, and it also tells us the name and type of each column. Now this is a very clean dataset in the sense that it doesn't contain any empty or missing values or errors, and you can see that reflected in the fact that each column has a value count of 8784. We have four integer columns which cannot hold missing values anyway, and a single floating-point column. The info method also shows us the amount of memory occupied by our data, and the type of index for this DataFrame. In this case, the index has been added automatically by Pandas, so we have a range index which is the default, a range of numbers from 0 to 8783. Now I also like to get an impression of what the actual data looks like. For that, there are two methods you should know about, head and tail. As you probably can guess, df. head shows you the first rows of data. Executing it, we see that it shows five rows, but I can optionally add the number of lines to show as an argument. So let's change this to 50 here, and now we see the first 50 lines. In this case, we don't see anything unexpected. The data does seem to be sorted by month, day, and hour though. Let's check the last few rows as well with df. tail, and this seems to confirm that our data is nicely sorted.

Demo: Statistical Exploration

Now that we know the dimensions and type of our data, let's do a little statistical exploration. We'll see how to get a general statistical overview, as well as a few more specific statistical functions, and a few basic plots. The next function we call on our dataset is df. describe, and this function gives a high-level overview of our data from a statistical standpoint. For each column it gives some basic statistical data, like the mean, the standard deviation, minimum and maximum

values, and the quartiles. For the first three columns, which tell us about the moment that each sample was taken, this information doesn't make a lot of sense, but for the last two they do. Apparently for this weather station, which is located in the Netherlands, the average temperature was around 10 degrees Celsius over the whole year with a standard deviation of about 6 degrees. Looking at the quartiles, at first glance it looks like the temperatures are pretty evenly distributed around the mean. We can also see that the Month column is in the range from 1 to 12, and the Day counts from 1 to 336, which makes sense because this data is from 2016, which was a leap year so it has one extra day. By the way, the behavior of the describe method depends on the type of data in your DataFrame, so if you have any columns that don't have just numbers, but, for example, objects or categorical types, you will see different results, and some columns may be ignored by the describe method altogether. Now you can get all of the numbers generated by describe in another way as well, by asking for them explicitly. To get the mean for the columns, for example, you call df. mean, and for the max df. max, etc., etc. But you can also run these methods against a single column. For example, to get the minimum pressure measured, I can say df followed by square brackets, and in there I put the name of the column I want to use, PRESSURE, and then I say. min, and this returns a single number, which is nice if I need to store it in a variable to do calculations with. There's a large number of things you can calculate in this way, and it's pointless to show all of them one by one, so let me show you how to get sort of an overview of what's possible. If you click on Help, and then Pandas Reference, this takes me straight to the Pandas documentation. And then I scroll all the way down to the API for the Series class. And then I have to scroll even further down to find the part about Computations/Descriptive Statistics. So here we see a list of many basic operations you can do on a Series, and almost all of these also work on your entire DataFrame. Here we find, for example, the count of all observations in the series, the kurtosis, the max and mean and median, quantiles, like percentiles and quartiles, ranks, standard deviation and variants, the sum of all the values, etc., etc. So if you want to know what the most often measured temperature is in this dataset, which in statistics you would call the mode, you should be able to find out that you can call the function mode for that, because it's in the reference I just showed you. So we can say df square brackets, temperature,. mode. Now this method always returns a Series, even when there's only a single value, so we see here that the first value of this Series is 6. 4, making 6. 4 degrees Celsius the most often occurring temperature. Finally, even though this course doesn't focus on visualization, I want to give you a few pointers for looking at the data in a graphical way. Most plotting functionality is accessed through the. plot attribute of a DataFrame or Series. To plot all temperatures, I can say df TEMP. plot, and this plots all temperatures as they occur in the series. Or maybe it's more informative to ask for a histogram by saying df TEMP. plot. hist. Now such a histogram you can make more or less fine grained with

the bins parameter, so let's set bins to 100 and we get a histogram with 100 bars. Now, of course, there's a lot more to be said about visualization and data exploration, but I think this short demo gives a good first impression of the things Pandas can do.

Summary

So, after you load a new dataset into a Pandas DataFrame, usually you want to do more or less the following: check the shape property to get the dimensions of the data, call the info method to see what data type each column has, what kind of index you have and how much space all of this occupies in memory, call head or tail to check out what the data itself actually looks like, and by the way, don't forget you can pass an argument to tell those methods how many rows to show, and call the describe method to get a high-level overview of important statistics like the mean, max, and min, etc. A lot of the time, you may want to go one step further and check some more specific statistical properties. Pandas has a lot of common statistical operations built in, and all of the functions I mentioned here can be used both on a series or on a DataFrame. First of all, there are the max, min, mean, and median functions. All of these are also generated by the describe method, but sometimes you need to generate them for a specific column, or you may need them for a calculation. The mode function tells you the most common value in the dataset, and will return multiple values if necessary. There's also a method value_counts which returns a series with the number of times each value occurs in your dataset. But notice, as an exception this method can only be called on a Series, so on one of the columns in your data, not on the entire DataFrame. In some cases, you want some more information about the distribution of your data, and for these cases Pandas offers methods like std and var for the standard deviation and variants, or the skew, kurtosis, and quantile methods, which do exactly what their name implies. Of course, it's also quite easy to make the most beautiful and informative plots and graphs with Pandas, but I won't go into these much in this course because we will be focusing mainly on the DataFrame and Series classes, and what you can do with them. Now, I only showed you a very minimal demo of all of this, and let me take a step back and look at a realistic scenario. I loaded a dataset about UFO sightings and called info on it, and this is what I see. It's a lot less friendly than what we saw in the previous demo. The count for the columns differ a lot, which means that a lot of columns are missing values. Almost all the columns are objects, which can mean several things, but one thing it surely does not mean is that these columns contain clean number data. This is actually a much more realistic example than what we saw in the demo a moment ago. In a case like this, we need to clean up and transform the data, see what to do with missing or erratic

values, and more, and that's exactly what we will do in this course. So let's move on to the next module about selecting, filtering, and sorting data.

Selecting, Filtering, and Sorting Data

Module Overview

Hi, in this module I'll introduce you to several ways to select, retrieve, and sort data with Pandas. After watching this module, you'll be able to retrieve exactly the subset of data you need from any Pandas DataFrame. Usually there are several different ways to accomplish this, and I will compare and contrast those for you. Most of this module is about indexing, which is the primary way to get to your data. There's a lot to say about indexing, because Pandas offers many powerful options there. I'll start by covering the basics of the indexing operator and what happens when you pass a single value to it to select a single row or column. Next we'll see what happens when you index a DataFrame with a more complex data structure like a list or slice, and we'll see two DataFrame attributes called loc and iloc, which allow us to index in even more powerful ways. Then we'll move on to Boolean filtering, selecting data using a comparison like all rows that have a value higher than x, for example. We'll see that you can assign values to the part of a DataFrame that you select when you use an index, and finally, I have a short word about sorting. Let's start with a demo about indexing. I'll start with the basics, selecting single rows and columns, and we'll gradually move towards more complex use cases like indexing with slices, lists, and indexing with loc and iloc.

Demo: Indexing Basics

I've started a new notebook, and you see the familiar imports. I'm reading the same csv file as before, and I'm getting the first five rows only by calling the function head on it before assigning it to the variable df. This way we have a small and simple dataset to work with for our demo. In a sense, a DataFrame is nothing more than a list of series containing values, and an index that gives each of these series a name, and each series is one of the columns we see here, like Time, Temp or Pressure. So the most straightforward thing we can do with a DataFrame is to select one of these columns, and we can do that with straight brackets and the name of the column we want, like

this. And this selects a series of temperatures from this dataset. Of course, when I use a name that doesn't exist I get an error. But let's assume we use an existing column and we retrieve a series. On this series we can now call methods like max or min, or we can select a single value from the series by using the same operator again, but this time on the series itself. Since the first part of the expression here evaluates to a series, we can put another subscript behind it, and that will work on that series. So to get the second temperature, which has index 1, I add square brackets 1. Now before I go into a little more detail here, there's something you should know. What you see in a lot of tutorials and examples is that people leave out the brackets when selecting a column, which looks like this, which, as you can see, selects the same series. Now if you know about Python classes and objects, you will know that this is a syntax for retrieving an attribute from a Python object. For our convenience, DataFrame has an attribute for each of its columns, but this will only work when the name of the column is a valid Python attribute name. To put it simply, this will not work when the column name has spaces or certain special characters like an equals or plus sign, or a dot, and you can also not add a new column with this syntax, which is possible with the brackets, and we'll see that later. So for all of these reasons, in my opinion it is better not to use this syntax, or at least don't use it in any production code. It's just not a best practice, even if it's shorter to type. Let's talk a bit more about simple indexing. For my next example, I'm going to transpose our DataFrame with the. T attribute and assign that to a new variable df. T, and let's see what that looks like. So in this case the names of my columns are integers, and the row index contains strings. Let's see if indexing still works the same way. Using the same approach as before with brackets and quotes doesn't work, and that's because the quotes here create a string asking the DataFrame for a column associated with a string 2. But our column names right now are actually integers, and you can check this by asking for the attribute dft. columns. And this tells us that the columns are indexed by a range of numbers from 0 to 4, so this means that to retrieve column 2 we need to remove the quotes. And now we retrieve the series we want. So to retrieve a column we have to use exactly the kind of value that's in the column index. In this case, it's a number and that means we can also not use the dot syntax. This isn't even valid Python code. Now interestingly, for series this works differently. Once I have a series, like here where I retrieve the column with index 2, I can retrieve a row from that series by a value that's in its index, so in this case let's get the Time for column 2. Let me just remove the other cells here to make things clear again. And similarly, in this case, I can use the index of the row I want to retrieve, since the row for Time is the third row so it has offset 2, and as you can see, I can retrieve it by position as well as by name. Now the subtly here is that you can retrieve rows from a series both by position and by name, but you can retrieve columns only by label. Scrolling back up, we see that we retrieved columns by their index label. Temp is the name of the

column as it occurs in the index in the first example, and if you try to retrieve this column with its position, 3, because it's the fourth column and we start counting with 0, you would find that it doesn't work, so you can retrieve this column by its name, but not by its position. And in the other example back down, here we are retrieving a column with a number, but that only works because in this case the column index also contains numbers. Once we have extracted a series from the DataFrame, though, like we would get column 2 here, then we have a series and we can retrieve rows from that series, both by label and position. So you see here that we retrieved the Time row both by saying Time and by saying I want row 2. Well, actually what I'm saying isn't completely correct, you can retrieve columns by position as well, and we'll see that in a moment. Now let me give you a final strange example. Consider the following DataFrame. In this case we have rows indexed with numbers, but the numbers are not a simple range. As you can see, there are even multiple rows with the same index. Now in this case my earlier comment is not true. Here you can only retrieve rows by name. Now my question to you is what do you think will happen if I get the series with values like this and then select row 4 like this? The answer I leave to you. I encourage you to play around in the notebook and figure this one out for yourself.

Demo: Indexing with Lists and Slices

So we've seen what happens when we put a single value between these brackets like a number or a string, but we can use more complicated things in here as well, like for example, a list of column names. And this returns a DataFrame with the columns I selected. As you can see, the columns can be in any order, regardless of the order in which they appeared in the original DataFrame. Similarly, when I select a single series I can use the same logic on that series to select a number of rows. Now note that in both cases here we have double square brackets. The outer brackets mean we are indexing into the DataFrame of our series, and the inner brackets mean that we're using a Python list to select multiple things at once. Another thing we can pass to the indexing operator is a slice. You may know this from Python. A slice a uses a colon to select a range of things. So far, we've used indexing to select columns from a DataFrame, but with a slice you always select rows, and this expression means we're selecting rows 2 up to, but not including 4, so that means rows 2 and 3. Because this returns a DataFrame, I can select which columns I want immediately after this, so let's use the double square brackets to select the temperature and pressure at these two rows. Now if we look at our transposed DataFrame, let's try to get exactly the same data points from this data structure. Again, indexing with a slice will always select rows, so here to select pressure and temperature let's say 3 followed by a column, which means I want a range of all rows starting from the fourth row, which is offset 3 up until the very last row. Again,

this returns a DataFrame, so if we want to select columns 2 and 3 from this we cannot use a slice, because that always selects rows, but I can use a list to select multiple columns. So let's use double square brackets, and now we have the same data as above, only transposed. Now if you select only a single row like this, we now have a series, and here, again, we can also use slicing, and again, a slice always selects rows, so I can say column 4 to select only the first four rows. Even in the case of our transposed data structure where the row indices are strings, slicing works, and it selects rows. And we can even use slicing with strings. So in this example I'm selecting the rows Time up until Pressure. Note that in this case the end of the slice is inclusive, so the Pressure row is included in the results, whereas with integer slices the endpoint of the slice is not included in the results.

Demo: Row-based Indexing with loc and iloc

So with the regular indexing operator on a DataFrame, you can select one or more columns. But what if you want to select just a row or several rows? We've seen that you can use slicing, but there are several other possibilities that we haven't seen yet. Before I start, let's make a new DataFrame to play with. This time I will create a DataFrame using Python data structures. You can type along if you like, or you can download the notebook from the course materials. What we have here is a list of some of the smallest capital cities in the world. I'm creating a DataFrame by calling pd. DataFrame, and I pass the list of cities as a list of lists. So these outer brackets contain five lists, and each of those lists is a row containing the data for a city. For example, we see that the country of Tuvalu has a capital city called Funafuti with about 4. 5 thousand inhabitants, making up 45% of the total population of the country. As a second argument, I'm passing a list of countries which will become the row index, and I'm giving an argument columns where I set the name of the columns, which is also a list. So now when I say capitals, this is what our data looks like. Now I want to introduce two other ways to retrieve data from this DataFrame. First of all, there's the loc attribute, which allows us to do row-based indexing as opposed to column-based indexing which you've seen in the previous clips. In this case, I also use square brackets, and I start with selecting a row, let's say Nauru. And here loc allows me to immediately select a column as well, inside the brackets. So let's ask for the population of the capital of Nauru by typing a comma and population within quotes. Now remember that to retrieve the same data from the DataFrame with column-based indexing, we have to first select the Population column and then drill down into the row for Nauru. And this is called chained indexing, and it takes two operations, where the line with loc only takes a single operation, and for this reason the use of loc is usually preferred over chained indexing, and I'll get back to this point in a moment. Both arguments of

loc also support lists and slices, so for example, here I use a slice to select the rows for Palau up to Nauru, and the list to select two specific rows. Of course, you can also use a list as the first argument and the slice as the second. Please take a moment to try and verify for yourself that this works. And, again, you can do this with column-based indexing as well, but it will take two separate steps, where this is a single operation. Now by the way, you can also leave out the second argument altogether. So in this example I'm using capitals. loc to quickly get the rows for San Marino and the Vatican. Now one thing you should realize is that loc only indexes by label, not by position. To retrieve items by position, we use another attribute called iloc. It works in a similar way, we start by selecting one or more rows, but this time we use integers to denote the positions of the rows we want. And, again, we can add a second argument here to select columns. I can select all columns except the first by using a slice that starts at 1. Now with iloc, indexing is strictly location based. So even if you have an integer index and there is no row with index 4, or the indices are not nicely ordered, this line of code will always select the fifth row and the second row. Now as a final example, let me show you, you can do something with iloc which you cannot do with the column-based indexing from previous clips. It allows us to select a column by position. Let me show you. I start by selecting all rows by using only a colon, which is a slice that selects the whole range, and then I can select column 2 for example. So loc and iloc allow you to select rows, columns, and even single cells with a single operation. You can use two arguments. The first one selects rows and the second one selects columns, and the big difference between the two is that loc uses labels and iloc uses positions. In other words, it uses integers to denote the position of a row or column. With loc, of course, you can also use integers, but only if your actual index labels are integers.

Review: Indexing

So the most straightforward thing you can do with indexing is to pass a column label, which will retrieve the column from the DataFrame as a series. Note that there has to be a column with that index label in the DataFrame, so in the second example here we are retrieving a column with label 5, not the column with position 5. To do that, we need to use iloc. Once you retrieve a column, you can use indexing on that to retrieve one or more of its values, and we call this chained indexing. If you need to select multiple columns at once, you can use indexing with a list. note that we have double square brackets in the example here. The outer ones are for the indexing operator, and the inner ones are for the list. And this way you can retrieve any number of columns in any order you like. You can also use a slice, and this behaves differently in the sense that it always selects rows instead of columns. With slices you can select by position or by name, and it both works. And, by

the way, you can use lists and slices with all kinds of indexing, so you can use them with loc and iloc, on Series and DataFrame, and it all works. The DataFrame has an attribute called loc which allows for row-based indexing. In the example here, we retrieve the row with label San Marino. If you also want to select the column you want to see, you can pass it a second argument. So with loc you can drill down to a single cell in a single operation. By the way, both arguments can be lists or slices as well, so you can select multiple rows or columns here. Now if you compare this to column-based indexing, you see that we first select a column and then the row, and we need two operations for this. And this is slightly less efficient, and in fact, using loc is usually preferred. I'll get back to this in a moment. Then there's iloc, which also does row-based indexing, but it works by position, whereas loc works by label. So with iloc I can pass an integer to retrieve a single row, and in this case it will retrieve the sixth row because, of course, we start counting at 0. Iloc also lets us select the column in the same operation. Again, this will select by position, so we're getting the third column, and like I've said before, you can use slices and lists as arguments both with loc and iloc.

Demo: Filtering with Boolean Expressions

In the second part of this module, I want to move on from just indexing. First I'd like to take a look at filtering data using Boolean expressions, then we'll see that we can assign new values to the parts of our DataFrame that we have selected with the techniques we have learned so far, and I'll finish up the module by taking a short look at sorting. As you already know, passing a list to the index operator on our DataFrame we can select columns from our data. So here I'm selecting the capital names with their population. But there's another kind of list we can also use. Let me show you. I can pass a list of Booleans, and this list selects only some rows from our data. The list has to have as many elements as there are rows, so in this case I need five elements, and for every row in the data I say whether I want it in my output. So here I'm selecting the first two rows, skipping the third, selecting the fourth, etc. Now you may wonder what's the point of this. If my data contains 200, 000 rows, I'm not going to type 200, 000 times true or false. Well, the beautiful thing is that we can generate lists of Booleans. Let's say I want to see only the rows where the capital city contains more than a quarter of all the population in the country. To start, I select the Percentage column, and remember this expression will return a series containing only the percentages, and now I can compare a series to a single value. So let's say is greater than 25, and this returns a series with Booleans, telling me for every row whether the percentage in that row is greater than 25. And because this expression returns a list, I can use it in an indexing operation like this. So let me explain the syntax another time. Selecting the Percentage column returns a series. Comparing

this with a value returns a series of Booleans which you can use to index your DataFrame, so this will return exactly the rows where the percentage is greater than 25. Now just to make a point, let me add another DataFrame and it contains a number of student names and their grades on two different tests. Now suppose I want to select those students whose score didn't improve after the first test. I can actually compare two columns against each other like this, and this, again, returns a series with Booleans, which I can use inside an indexing operation. So this shows me exactly the students who didn't improve. You can also use these Boolean expressions with loc and iloc. For example, I might want to see only the tests that have an average over 5. 5, and this means I want to create a list of Booleans that has the same length as the number of columns in my data. First I will calculate the averages, and this already creates a series, so all I have to do is to compare this with 5. 5 and now we have a series of Booleans, and I can plug this into loc like this. So here I use a colon as the first argument, which means I want to see all rows, and as a second argument I pass my Boolean list, which selects only the first column because that's the only test with an average over 5. 5.

Demo: Assigning Values with Indexing

So far, all of this module has been about retrieving data, but in this clip I want to talk for a short moment about changing data. You see, almost all of the indexing operations return a view on the original DataFrame, and you can assign values to that. Let's say I made an error in marking test 2, and both John and Laura deserve better grades. I can select their marks using loc, and then I can increase their scores with 1 by using += 1. Checking the contents of the DataFrame, we see that the grades have been updated. I can even upgrade an entire column at a time. Let's increase the grade on the first test, or for a specific student. So these indexing operations don't return new data, but a view on the original DataFrame allowing us to update it. You can also pass multiple values, so let me just set the grades for Pete, and passing a list updates multiple grades at once. You can even use assignment with the Boolean filters we've seen before. Let's convert the values we have right now into a string saying Pass or Fail. First I select all grades smaller than 6 and set them to the string Fail, and then I take everything larger than 6 and covert those to Pass. Surprisingly, this converts everything to Pass, and that's because after the first line we have a DataFrame holding both numbers and strings, and comparing the strings to the number 6 in the next line always yields true, so we convert everything to the string Pass. Now one way to fix this is to first store the Boolean arrays in a variable. Remember that these expressions, grades smaller than 6 and grades larger than 6, return series of Booleans, so I can assign these series of Booleans to variables, and then I can use these variables inside the square brackets. Of course,

right now this is not going to work because I already changed all my data into strings. So let's rerun the whole notebook from the start, and now we have the result we want. Now, of course, this is kind of a silly example, so let's make it a bit more realistic. I'll start by recreating the original data, because I want to leave the code above for you to read. So a more common scenario for a teacher is to take the average for each student, not for each test, so we want an average by row. But the mean method, by default, takes averages per column. Fortunately, we can add an argument, axis, and set it to 1, which means we want to average all of the rows, and this returns a series containing the averages per student. Now let's convert this to Booleans by saying greater than 6, and this gives us a series of Booleans. And we can actually add this data as a column to our DataFrame by simply using the name of new column for an index. So here I'm creating a new column and assigning it a Boolean value denoting whether this student passed. Now looking at the data, you can see that this column has now been added. So indexing also allows the creation of new columns. Sometimes when you make a mistake this also causes the creation of a new column. For example, I want to set Ann's grade on test 2 to an 8, but I forgot to use loc. Now this indexing operation is nonsense. We cannot pass a column and a row like this, so instead Pandas assumes I want to create a new column. Looking at the results, you see that there's now a new column that we didn't mean to create at all. We've already seen that one way to do this uses loc, but you might also try to do this without loc, selecting first the column and then the row with two different index operations. But this gives a warning. Checking grades, we see that the statement has been successfully executed, so what went wrong here? Well, using multiple index operations after each other is called chained indexing, and if you use those and try to assign a value to it, Pandas cannot guarantee that it will work, so in these cases it's better to use loc because that's more likely to work and it's faster anyway. Now you might wonder when exactly an indexing operation returns a view on the original data that we can safely assign to. Well, to quote the Pandas documentation, outside of simple cases, it's very hard to predict whether indexing will return a view or a copy, and that's exactly why Pandas gives the warning above, to let you know when you are in a situation where assignment may not work exactly the way you expect it.

Demo: Sorting

To wrap up the module, let's talk about sorting for a moment. There are two methods you need to be aware of. First there's sort_index, which, as the name implies, sorts by index. So I can say capitals. sort_index, and I get a new DataFrame sorted by index, which in this case means we sort by country names. By default, this method does not change the original DataFrame, so that means if we check capitals it's still unsorted. If you need the original data to be sorted, use

inplace=True, and now the capitals variable itself is sorted. You can also do a reverse sort by saying ascending is False. And you can also sort by the column index. To do so we add the argument axis is 1, so now our data is sorted in descending order along the row index and in ascending order along the column index. But, of course, sometimes you don't want to sort by index of your data, but by value. For this there is the method sort_values. And if you run it without any arguments you get an error, because you need to supply it an argument so that it knows by which column to sort, so we can say we want to sort it by the Percentage column. Sometimes you need a secondary column to sort by. As an example, let's sort our grades. And here I'm passing a list of columns to sort by. The primary sort column is test_1, but students who received the same score on that test, for example, Mary and Pete who both received a 6, are sorted by test_2. By the way, sort_values also supports the other arguments we've seen like inplace, ascending, and axis. As always, I suggest you take a moment and play around with these methods to verify that this works the way you expect it to. And that brings us to the end of our demos for this module. Let's review.

Review and Summary

We've seen that we can use Boolean comparisons to select parts of our data by using the comparison inside the indexing operation. The example you see here selects all rows from a DataFrame where the column called Temp has a value greater than 20. Note that we have two pairs of square brackets, one to select the column and another one that accepts the Boolean expression and selects matching rows. The name of our DataFrame, df, also appears twice. Now this works because the comparison of a series with a number returns a new series with Booleans, so comparing the Temperature column with the number 20 returns a Boolean series and we can use that inside square brackets to retrieve matching rows. You can also use this with other ways of indexing like loc and iloc, and here I have an example where I first pass a colon, which is a slice that selects everything. So this first argument selects all rows, and then the second argument is a list of Booleans. In this case we select columns, and that means that the list must contain as many Booleans as there are columns. Well, grades. mean returns a mean for every column, and comparing that to the number 6 returns a list of Booleans. Most of the time using an indexing operation with Pandas returns a view on the original DataFrame, and that means you can assign values to the selected part of your data. In the first example here I'm I'm selecting a column and increasing every value in it by 1, and the second example selects a row and sets its values using a list. This tends to work with all indexing operations, so it also works with loc and iloc. In this example, I'm selecting two cells by using loc and selecting a row and two columns, and I'm setting

both values to 8. You may have noticed that I was using uncertain wording with the previous slide. This is because I cannot give you certainty that these Pandas operations always return views. The Pandas documentation is quite clear on the uncertainty here. They cannot give any guarantees, and this is especially true in the case of chained indexing, like you see in the example here. I'm first indexing a DataFrame which returns a series, and then I'm indexing that series. When I try to assign a value to that, I get a SettingWithCopyWarning, and this means that Pandas cannot tell you whether you're assigning to a view of your original data or to a copy. So, did the assignment work? I don't know, it depends. The best thing to do is test the result of your code, and in general you will usually see this warning when you use chained indexing, so try not to do that when you assign values, instead use loc and iloc because those usually work. We also saw two methods for sorting your data. First there's sort_index, which as the name implies, sorts your data by the index, and the other one is sort_values, which sorts on the actual values of your data. In this case you need to specify which column of your DataFrame to sort by. If necessary, you can also select multiple columns to sort on. There's a number of extra arguments that you can pass to the sort methods. First of all, there's the ascending option, which makes Pandas sort in reverse when it's set to false. Then there's the axis argument, which you can set to 1 if you want to sort along rows instead of columns. Actually, there are a lot of methods that take this argument. In the demo, we've also seen it, for example, with the mean method. And usually sorting with Pandas does not change the data itself, but it will return a sorted copy. If you want to change the original DataFrame, pass the argument inplace=True. By the way, all of these arguments work with both sort_index and sort_values. And that's it for this module. We've learned a lot about indexing from the basics of retrieving single rows and columns to indexing with slices and lists and doing row-based indexing with loc and iloc. We saw Boolean filtering, which is a powerful way to select parts of your data based on logical expressions, assigned new values to our data, and took a short look at sorting methods. Thank you for watching, and I hope that you'll join me in the next module where we'll talk about cleaning your dataset with Pandas.

Cleaning Data

Module Overview

Hi, in this module we're going to take a look at the common ways to clean up your dataset. When you start working with a new dataset, usually there are some steps you want to take to tidy up. In

this module, I will go over some of the more common scenarios and how to tackle them. To start, we will see how to investigate missing data, also called null values, and we'll see how to detect these values and take a closer look at them, how to remove them, and how to replace them with other values either by replacing all missing values with a filler, or by interpolating them from surrounding data. You might also have some other unwanted data in your dataset like outliers, or data that occurs more than once, and of course we'll see how to handle these as well. And we'll see how to fix values that don't have the correct data type, and finally, we'll take a look at tidying up the index of your DataFrame.

Demo: Detecting and Inspecting Missing Values

So, let's start off with the first demo, in which I'll show you how to work with missing data. First we'll see how to detect missing values, then how to remove them, and finally we will learn how to replace them by other values based on your actual data. So here we are again in a new notebook, and we start with the familiar two imports of pandas and numpy. I'm reading a csv file again, and it's from the weather dataset we've seen before, only this time I haven't cleaned it up completely. When I call info on it, not only do we see some extra columns we didn't see before, there is another interesting thing you may not notice immediately. Not all counts for all columns are the same, and this means that not all columns contain the same amount of values. For example, we see that the MIN_TEMP_GROUND column contains only about 1500 measurements. If you check it, you will find that it only contains a value for every six rows. The VIEW_RANGE and CLOUD columns also have some missing data, as well as the columns for different weather types, MIST, RAIN, SNOW, THUNDER, ICE, and at the end we see the WEATHER_CODE column and it also has some missing data. Taking a look at what some of these columns contain, we see that some cells here have a value of NaN instead of a number, and this is an abbreviation for not a number, and it simply means that there is no value here, so no measurements. Basically, there are holes in our data. Sometimes you want this. Looking at the MIN_TEMP_GROUND column, the minimum temperature at ground level is only recorded once every 6 hours. It's the minimum temperature measured over that interval, and the number shows up in the data at the moment that the measurement was recorded and the data is left empty at other moments, so this is not an error. In the case of the missing WEATHER_CODE, I'm not so sure where this comes from. It might mean anything from a measurement error to an unknown weather type. Let's take a closer look at the rows and columns that contain missing values. I can call df. isnull, and null, by the way, means no value, and in the Pandas ecosystem null and not a number are equivalent. This is not the case everywhere though. In many languages, not a number and null are two very different things.

Anyway, in this case the function isnull returns a DataFrame with True whenever our data is null and False otherwise. On this DataFrame I can call any, and this function takes every column of our DataFrame and returns True if it contains any true values at all. So in this case it shows True for every column that has null values, and that's quite a lot of columns. Again, df. isnull is a new DataFrame that contains True for every null value and False for everything else. And the function any works similar to, for example, the mean function. It computes a value for every column in the data. In the case of any, it returns True if there is any true value in the column. Like we've seen before, we can give arguments to say that we want to work over the other axis, so over the rows instead of the columns. And now we have a True value for every row in our data that has one or more null values. And because this is a list of Booleans, we can use it to select these rows and look at them. Right now, all we can tell from this is that most of the null values are in the MIN_TEMP_GROUND column. We'll fix that soon. For now, I wonder if there are any rows that only contain null values. Again, I'll start by saying df. isnull, but this time instead of any I'll use the all function, and in this case we get an overview of all elements of our DataFrame where all values are True. In the case of the columns, we already know that every column contains some data at least, so all columns return False. To do the same for all rows, again, I say axis is 1, and I get a very long list telling me for every row in our data whether it is completely empty or not. I would like to know if there are any True values in this list, so of course I can say any again. So now we know for sure that every row in our data contains at least one value. Explaining the syntax here once here, df. isnull is a DataFrame that's True for every null value, and for every row in that DataFrame we check whether it's all true values by saying all with axis is 1. So now we have a series of True and False value. Calling any on that Series tells us whether it contains any True values, which it doesn't, so we know if we don't have any rows that are completely empty. Now, there's another function called notnull which does exactly the opposite from isnull. It returns True for every value that is not null. So to get all columns that have no values at all, I can say df. notnull. all. Now let's take it one step further. The column that has by far the most missing values is the minimum ground temperature, and by simply looking at the data it seems that every sixth value is a number. And actually, this is also what the meteorological institute says. This temperature is only recorded every 6 hours. But let's not just trust what we hear, let's check that it's true. So how do I check that every sixth value is true? Well, let's start by making a list of all positions in the column where I suspect there should be a number. So this uses the standard Python function, range, generating a list of numbers starting with 5, which is the position of the first number in our column, up to the length of the DataFrame with a step size of 6. To get a nice overview of what we just created, I can make this into a Pandas Series. And this shows us that the first number is 5, then 11, then 17, etc., etc., and scrolling down to the bottom we see that there are 1464 numbers in

this list. By the way, making this a Series is not actually necessary, I just did it to quickly display the contents of our list. Now, let's store this in a variable called every_6th_row. And you might remember that we can use a list of indices to retrieve rows from our data, so let's take the MIN_TEMP_GROUND column and index it with our list of positions. So this will select exactly every sixth row from this column. To check that these cells all contain a measurement, I say. notnull. all, and this returns True, so now I know that all these rows are filled. Now to be completely certain of the way this column is filled, let's check the opposite. Are all other rows empty? Again, I select the column, but now I remove every sixth row by using the method drop. Now the drop method does not work in place by default, so our original DataFrame will stay intact, so we're not actually removing anything form our DataFrame. Now these values should all be null, so I can say isnull. all. And, again, we get the answer True. So now I know for sure that this column contains a value every sixth row, and not a number everywhere else. Good. Now I would like to ask you to take a moment and see if you can rewrite the previous line to use loc instead of chained indexing.

Demo: Handling Missing Data

Let's begin cleaning up our dataset. I'll start by completely removing the weather type column because that one contains coded values that are useless to me. I can remove a column by calling drop on our DataFrame, and by default drop removes rows, so I should specify that I want to remove a column. And here I always run into the same pitfall. Actually the argument is not called column but columns, even if you only want to drop one column. So let's fix this, very well, and this gives a new DataFrame without the WEATHER_CODE column. The fact that we see a new DataFrame here might tell you something. The original DataFrame hasn't been changed. To actually drop the column from our original DataFrame, we add the familiar inplace parameter which we've already used with other methods. Now the most straightforward approach to dealing with missing values is to just remove them. Now in many cases this is not actually necessary because most calculations and plots handle null values in the way you probably want them to. Taking the mean, for example, simply calculates the mean from the data points that are actually there, ignoring any null values, so you might just be completely okay with null values in your datasets. Now in case of the MIN_GROUND_TEMP, I only have a value once every six rows, and removing all rows with null value is not an option at all. But I may want to fill in this data with something else. Pandas offers a function to do this for us called fillna. The simple way to use it is to just provide a value, let's say 0, and this fills every missing value with 0. Now, of course, this is not actually what I want in this case, I want to fill the missing entries based on the actual

measurements. Fortunately, fillna can also do this by saying method is ffill, which stands for forward fill, and Pandas will take each measurement and fill the empty slots after it. You can see the result here. Each measurement is repeated six times. Only the thing is that the measurements are taken over the previous 6 hours, and as a result our first five cells are still empty, so we actually want to fill backwards. For this we can use bfill. Very well. This looks better. Let's add the parameter inplace is True to update our DataFrame with these values. And let's check for the columns with null values again, and now it tells us that the MIN_TEMP_GROUND column doesn't have any null values anymore. So that's nice, we seem to be about halfway done, because almost half of our columns are now False here. And as a side note, in case you need more sophisticated interpolation of your values, instead of just copying a value across a number of missing values, there's also the interpolate method. And the interpolate method will do all kinds of interpolation. I encourage you to try out this method on the MIN_GROUND_TEMP and see what kind of results you get. Now let's go back to the notebook and let's inspect the missing values we still have left in our data. Remember, we can get the rows with missing values by using the same expression we already have, but specifying the other axis, and we can use that to select the actual data by putting it inside square brackets. And this shows us the rows with missing values we have left. Scrolling to the right a bit, we see that most of the time all the values are missing in the last seven columns. Now to look at the date where this occurs I can use indexing with loc, and this selects only the dates from these rows, and I'm going to say. value_counts. And now we see that there are three days in August where there are null values for a large part of the day, and to me this looks like there were technical problems at the weather station at that time. So let's just assume that for my data science project I need temperature, pressure, and weather types, so these rows will now be useless to me because they are missing for large parts of the date. And I see two possible solutions. Either remove this data or try to fill it in some way. Now let's start with the first one. To simply drop all rows that contain null values, we say df. dropna. Now I don't want to do this inplace because I want to show another approach in a moment, so I don't want to mess up my data. Instead I wail save the output from this command into a new variable called nulls_dropped. So this variable will contain a new DataFrame where all the rows with null values have been dropped. And calling info on it now shows that all columns have the same count of non-null values, but note that the index still goes from 0 to 8783. So the side effect of this approach is that now our data isn't nicely continuous anymore. If I ask for rows 5300 to 5310, we can see here that the index is discontinuous. It jumps from 5305 to 5318, because that is where we just removed some rows. You can also see that the row with position 5306 now has index 5318, which is kind of weird. We'll see how to fix this later. Now a very useful option with dropna is thresh. It allows me to determine how many values must be present in a row to keep it in my

dataset. So here, again, I have the original dataset df and I call dropna on it, but I say that I want to drop only those rows that have less than seven measurements in them, and then I use isnull and any to inspect the rows with null values. And as you can see, now we have only two rows left, and only the last five columns are null. Now these are weather types and contain only 1 or a 0 depending on what is observed, so you could, for example, use fillna to fill these cells with zeros. Please take a moment to try this for yourself. Now, again, in this case we've dropped some rows and our index will not be continuous anymore. So let's take a look at another approach, which is to fill in our missing data instead of dropping it. I haven't used an inplace parameter, so our original DataFrame, df, is still intact. The expression to select all rows with null values should by now be familiar. Remember, this returns a series of Booleans, which is true for any row with null values. Let me just store this in a variable, rows_to_fill, and you can use it immediately to look at the rows in question. These are exactly the rows we were dropping from the DataFrame a moment ago. Now a very common approach here is to fill the empty cells with some kind of statistical estimate of what the value might be, like the mean of the column. Remember, we can calculate the mean of each column with df. mean, and I can simply pass this to fillna to use these mean values instead of the missing cells. And let's store this in a variable again. Now, of course, I want to see what the affected rows look like, but I cannot ask nulls_filled for its rows which used to be null. Luckily, I stored these rows in a variable, rows_to_fill. Okay, so this looks nice, but now the last six rows don't contain zeros and ones, but some floating points, and that's not what I want. So in the cases of VIEW_RANGE and CLOUD I'm okay with those rows having a mean value, but in case of MIST, RAIN, SNOW, THUNDER, and ICE I just want to see ones and zeros. So in this case it might be better to use the mode instead of the mean, and the mode of our data is the most occurring value. Now there is a trick to using it though, and we can read it in the documentation of the mode method. And here is a line that actually shows us exactly how to call mode with fillna. So let's copy this line and put it in the notebook, and now we can see that our data has been filled in quite nicely. So at this point I'm satisfied, and I want to store this result back in the original DataFrame. So I'm just going to use inplace is True to actually update our DataFrame, so this will be the version we will be working with from here.

Review: Missing Data

So to detect the missing data in our DataFrame, we use the function isnull, which returns a Series that is true for every cell that is not a number. And then we can plug that into any, which returns true if a column is true at least once. So if we say df. isnull. any, we get an overview of which columns have missing values, or you can use the axis is 1 argument to show the rows that have

missing values. And then if you plug that into an indexing, so here you see df. float by square brackets, and then in the square brackets the whole expression df. isnull. any with axis is 1, this will actually show you the contents of the rows that have null values. Now there are two other functions, notnull, which does exactly the opposite of isnull, returns a Series that's true for every cell that is not null, and there's all that does the opposite of any in the sense that it returns true only if a column is completely true. To remove parts of your DataFrame, there's the function drop, and of course you can use that function drop to remove missing values as well. But there's a more powerful function called dropna, and this is actually a workhorse, it can do a lot of things. First of all, if you just call it like that, df. dropna, it will drop all rows with null values from your DataFrame. Or if you call it with axis=1, it will drop all columns that contain null values. Sometimes you only want to drop those rows that have a lot of missing values, and for that you can add the argument thresh, which only keeps those rows which have more values than the threshold you specify. Then there's the how argument, which lets you either drop all the rows which are completely null, or all the rows where one or more values are null. So you can say how is all or how is any. And finally, of course, there's the inplace option which we've seen with almost every function that you can use on your DataFrame. Filling missing values you do with the fillna method, and you can just pass it the value and it will replace all the null values in your DataFrame with that value, so here it will replace all the null values with the value 5. But fillna can also accept a Series, and in that case, you can, for example, compute the mean for every column with df. mean, pass that to fillna, and per column all the nulls in each column will then be replaced by the mean for that column. And you can see that this is a very powerful way to fill missing values with a statistical estimate. Sometimes you want to fill the null values with values based on the surrounding data, and there's two ways to do this. You can either use the fillna method and you can give it a method argument that tells it how to fill the null values. Either give it the ffill option, which fills forward, so when it sees a null value it will fill that with the previous value it has seen, or bfill to fill backwards, which means if it sees a null value it will take the next value that it sees and fills backwards. Now of course fillna also accepts the options inplace and columns, as we've seen so far with every other method. Now if you need a little bit more advanced interpolation where you do calculations based upon the surrounding data, there's the df. interpolate method which lets you do all kinds of mathematic estimates for interpolating your data.

Demo: Handling Outliers

So we've either removed or filled all the missing cells in our DataFrame, but there might still be other unwanted values in our DataFrame. Let's look at two classes of unwanted values, outliers

and duplicates. There's not really something like a formal definition of what exactly an outlier is, but let's take a short look of ways to detect and remove them anyway. First of all, I'm reading in a dataset about athletes who competed in the Olympic games of 2016. And here we see the various columns in our data. There are actually some missing values in some of the columns, but nothing to worry about right now. For a visual indication of what our data looks like, let's start with a small scatterplot. First I need to say matplotlib inline to make sure that the notebook will show our plots. Then I type athletes. plot. scatter and I select the height column for the x axis and the weight column for the y axis. And here we see heights and weights of all the athletes in the dataset plotted against each other. You might say that there's some kind of cluster of data points, and there are some athletes that are not part of that cluster. Now suppose I want to filter out those athletes that are either very tall, or not tall at all. I get the height column from our DataFrame and store it by itself into a new variable, heights. This I can upload as a box plot by saying heights. plot. box, and this gives a nice visual representation of the distribution of the heights. Half of all athletes are within the box, and 25% of them are below the box, and another 25 are above it. The circles represent outliers. Now exactly how an outlier is defined can differ between box plots. The default behavior for this plot is that it takes the edges of the box, adds the total length of the box times one and a half, and anything further out than that is called an outlier. Of course, this is kind of an arbitrary definition of an outlier, and since there are quite a lot of circles here, it may not be the right definition for this dataset exactly, but let's just see if we can remove these outliers from our dataset. I start by defining two variables, q1 and q3, which are the first and third quartile, or in other words the 25th and 75th percentile. If these terms mean nothing to you, let me put it this way, if we sort all heights in order from short to tall, q1 here would be the data point where exactly one fourth of all data points are smaller, and q3 is the point that has exactly three-quarters of all data points smaller than it. And I'm setting these values because they are exactly the edges of the box on the box plot. Saying q3 - q1 in the next line gives us the height of the box, also known as the interquartile range, or IQR. Very well, now we can calculate the position of the ends of the whiskers like this. Pmin is the first quartile, minus 1. 5 times the iqr, and pmax is the third quartile plus 1. 5 times iqr. So now we have basically the same data that the box plot shows graphically. Pmin and pmax are the horizontal lines at the end of the vertical lines and everything beyond them are outliers. So we can use this to remove everything lying outside of the range pmin to pmax. And because I will be throwing data away, I prefer not to change my original data at first. So let's make a new variable, let's say nwh for new heights, and here I select all data points from heights between pmin and pmax. The function between returns a series of Booleans that is true whenever a value lies between pmin and pmax. So I use this series to index into the heights variable, and this leaves me with a list of heights that does not

include outliers. This I store in a variable, new heights. But actually, let's think about this a moment. Do I really want to remove the data points with heights that are outside this range? If I'm going to remove entire rows from the DataFrame, I'll be throwing away other data for these athletes too, like the medals they've won for example, and this might affect our results later on in a bad way, so in this case I think it's better to actually replace the outliers with null values, so actually I'll be introducing empty values in our dataset right now. And one way to do this is by using the function where. Now this method works a little bit like an indexing operator in the sense that I can pass it a list of Booleans, which I do by using heights. between as its first argument, and it returns the original height for every true element and null for every false element. In other words, where replaces everything that doesn't fall in the range pmin/pmax with not a number. Now let me just repeat that I'm not trying to tell you that this is the way to remove outliers from your data. What you consider an outlier will depend on your data and what you want to do with that data. For example, instead of the code here you may want to write code that selects everything outside of two standard deviations around the mean, or maybe you want to replace them with some value, let's say the mean of all heights, and in this case, you can add another argument to the where function and this will replace the outliers with the mean. Now in my case it doesn't really make sense to give a lot of people an average height so let's just remove this again and just replace the the outliers with null values. So now I have two series, one with outliers and one without, and I'd like to compare them. A nice way to do this is to put them side by side in a new DataFrame, so I call pd. dataframe and pass it a dictionary with the keys before and after, and our two series as arguments. Now let's compare these both in text and graphically. So I'm going to say compare. describe, and compare. plot. box. And now we see a comparison of the data before and after removing outliers. We've removed about 100 data points, and this has resulted in a slightly different mean value and a tighter standard deviation. The box plot, as well, shows that the distribution of our data has hardly changed, but all the circles are now gone. Let's apply this to the original DataFrame with athletes. I'll just copy the line where I replaced the outliers with null, and add inplace is true. Now if we draw another scatterplot, this shows us that the data is now a bit more compact.

Demo: Handling Duplicates

There is a third kind of unwanted data you may have in your dataset, and that is duplicate values. These may occur in your data for any number of reasons, and right now I'm going to show you how to get rid of them. First of all, there's a function called duplicated, which returns a list of Boolean values. For each row it will contain True if it's a duplicate and False otherwise. Just

running it like this tells us nothing, but fortunately we can use the any function to see if there are any duplicate rows in our data, and it seems that there are. So let's check them out by using the call to duplicated to index our DataFrame. So it seems that there's a total of five rows that are duplicated, so these rows appear more than once in our data, and we can simply remove them with another function called drop_duplicates. And because I want to update my data structure, I'll add inplace is True. And now just to check, let's rerun this line. There are no more duplicates in the set. Now the drop_duplicates function has more uses than just cleaning up a dataset with unwanted duplicates. Sometimes it's a nice tool for data exploration, for example, I might want to know the list of countries that has taken place in this tournament. But since most countries send more than one athlete, each country appears multiple times in the list, but if I take the column with nationalities and apply drop_duplicates, we get a long list of 207 nationalities in the tournament. Of course, you could also call sort_values on this series to get the countries sorted by name. But a lot of the time in scenarios like these there's another function I like to use instead of drop_duplicates, and that is value_counts. This also returns every value in the column once, but it also counts how many times each value occurs, so here we can see the same 207 nationalities, but we can immediately see which countries sent the most athletes, the USA followed by Brazil and Germany. Or if we want to know the ratio of women versus men, I can get the value_counts for the sex column, and we see that the ratio of men to women is about 6 to 5. We've seen that the duplicated method returns a series of Booleans that's true whenever a row has been seen before, so when it's a duplicate, and you can use this series, as always, to index into your DataFrame. So here we say df followed by square brackets, and then inside the square brackets we call df. duplicated, and this will show you all the rows that are actually duplicates. Now to remove all duplicates, there's a separate function called df. drop_duplicates, and this will remove all duplicated rows from your DataFrame. Interestingly, a DataFrame also has another method called unique which has the same behavior, but it doesn't return a new DataFrame, it returns a numpy array, so that means that most of the time you don't want to use it, you just want to use drop_duplicates.

Demo: Type Conversions

Now for the final part of this module, let's look at two more ways to clean up our data. First of all, let's convert those values that have the wrong data type into the right data type, and let's see how we can tidy up our indices. So we've done some cleanup of this dataset, but looking at the output of athletes. info, I see something I don't really like. Most columns have type object, and this basically means that they have been read from the csv file as strings, and I don't really agree with

that. For example, let's look at the three last columns. Apparently, these cells denote the number of medals an athlete won. In this case, we see that the third athlete in our dataset won a bronze medal, so these columns are numbers. Why were they not read as numbers? You see, when they're not seen as numbers by Pandas, a lot of calculations won't work the way we want them to. For example, summing the bronze column to get the total number of bronze medals awarded, this just concatenates all the strings and gives me a long list of ones and zeros, and this is not really helpful, so let's try to convert one of these columns to integers and see what happens. And for this there's the Pandas function astype. Now this gives an error, and when I scroll down all the way to the end we see that it tells us that there's an invalid literal for int. In other words, there's a value in the data that cannot be converted to an integer, and actually it turns out that this is the capital letter O. So this data was probably entered manually and someone typed an O instead of a zero. So let's fix this. I will retrieve the row holding this value. So this row we see here holds O's for the medals. I can change these values into zeros, let me just copy the row number here, and we'll use loc to retrieve the row and select the three columns. And I can just set them to 0. So let's try again to convert these values into integers. This time I'll try to convert them all at once by just selecting all three columns and saying. astype(int). And let's assign the result back to the original three columns. So I'm just going to copy this here and assign that back into itself. So now these columns should contain integers, and we should be able to do things like sum them to see how many of each medal were given out. And that looks a lot better. So now when I run athletes. info again, here there's only one data type left that I disagree with. If you look in the csv file, you will see that all the weights are given in whole kilograms, so why do these show up here as floats? Well, in this case it's because some of these weights are missing, and the only data types that can contain null values in a DataFrame are either floats or objects. So if you want to convert the weights to ints, we either have to drop the null values or fill them with some integer value. And personally, in this case I think both of these options don't really make sense, so that means that I'll have to live with a column that shows floating points even though the source values in the csv were integers. Now unfortunately this is just one of the limitations of Pandas. It's not possible to have a column of integers that also contains empty values.

Demo: Fixing Indices

Let's do some final cleanup by cleaning the index of our dataset. As it turns out, the original csv data already contains an id for each athlete, so why not just use that for the index. We can do this by assigning this column to the property athletes. index. Now the only thing is that the id column is now still here, so now we have the id data twice. You can, of course, just drop the id column,

but another approach is to not set the index directly, but use the set_index function. So here I say that I want to use the contents of the id column for the index, and after that please drop the id column and let's do this inplace as well. So now we have the athletes' id's as indices, and the id column is gone. Now let's say I want to change some column names as well. I can do this with the rename function, which lets me change column names using a dictionary. So let's change the name of the nationality column into the name country, and instead of sport we'll use the word discipline. And again, let's see the results. So this lets us change the labels on one or more columns. And you can use this function on the rows as well, but then the name of the first argument should be index instead of columns. As a final example, I want to take a moment to look at the weather dataset. Let me just read it in again, and let's just suppose I drop all rows with null values. And because in the original data only a small number of rows contained no empty values at all, we only have 831 rows left, so I just threw away the majority of my data, but the index, as you can see, still goes from 5 to 8783. Taking a look at the actual data, you can see the gaps in the indices. Now we can easily fix this by calling reset_index, and this resets the index to a simple range counting from 0 to the end of the DataFrame. One side effect of this is that the old index is stored in a new column called Index, as you can see. Sometimes that's nice to have, but right now I just want to get rid of the old index so I add an a argument drop is True, and now we have our DataFrame with a nice continuously increasing index.

Review: Type Conversions and Fixing Indices

To convert the type of a Pandas column, we can use the astype function, and we pass it the date type we want to convert to. So in this example here we take a column, in this case we take the column called some_column, and we call. astype(int) on it to convert all the values to ints. Alternatively, you can convert multiple columns into different data types by passing a dict into astype. So here I call astype on the DataFrame as a whole by saying df. astype, and I pass it a dictionary that says I want to convert the name column to a string and the age column to an int. Now it's very important to realize that all of this only works if all values in the column actually fit into the target data type. So let's say that the age column in this example has only one value that's not convertible to an int, let's say it contains a character. In that case, the df. astype will raise an exception. So you'll have to make sure that all your values actually conform to the new data type before you try to call astype. Now what kind of data types can you pass to astype? Well, there's four broad categories you have to be aware of. First of all, there's strings. These are nullable, so these cells can be empty, and the string data type can be denoted in two ways, you can just say str, which means the Python string data type, or you can say np. object. It doesn't

really matter which of these two you pass to the astype method, because in any case the type of your column will be objects, but it's good to realize that in a sense these are similar to Python strings. Then there's floats. These are also nullable, so these cells can also be empty, and you can use the Python float data type, or in numpy you can say np. float64. Actually, there are some more numpy float data types, they are, for example, float32, but normally you won't need those, and the use of those I consider to be a little bit too advanced for this course. Then there's integers. Integers cannot be null, they cannot be empty, and as you probably know, integers are whole numbers and floats are numbers with a dot and a decimal part behind the dot. Now for an integer you say either int, which is the Python int data type, or you use the numpy name, np. int64. And, again, there's multiple numpy int data types. There's also, for example, an int32, and similarly as with float. Usually you don't need it, you just use np. int64. Now if you convert a column to float and then you call info on your DataFrame, you will see that the column is of type float64. And similar with int, if you say astype(int) usually you will see that the column will be of type int64. And then there's some other data types as well. I don't really use those a lot in this course, but I just want to mention them here. For example, there's the bool data type that can just be true or false, and you can convert to that by saying astype(bool), and there's the complex data type for complex numbers, and those are the four most important data types that you might pass to the astype function. So suppose you have a dataset and initially it has a very nice index that has a range of numbers counting from 0 up to the end of the DataFrame, but you remove some rows and after that your index is kind of messed up, it's not continuous anymore. To fix it, you call df. reset_index, and it resets the index to a simple range. Now as a side effect reset_index creates a new column that has the data from the original index, and in many cases you don't really want that and you can add the drop is True argument, and that will drop the original index and you'll only be left with the new index. Something else that sometimes happens is that you read in a dataset and it already has a column with data that you would like to use for the index, and to do that, you call set_index. So here I say df. set_index, and as a first argument I pass it the name of the column that I want to use for my index. So in this case, the column is named id, and I take all the values from the id column to fill my index. And, again, here we see the drop is True argument to drop the original column from our dataset so that we don't have the same data twice. And the final thing we saw is the rename method, and this lets me rename either columns or indices by passing a dictionary that contains the original name and the new name for either each column or each row. So in the first example here I say rename with columns and then I pass a dictionary and I rename the column with name 'a' to Ann and the column with name 'b' to Bob. And then in the second argument I'm renaming rows. So here I say index, which means the labels in the indexes, and the row with index 'a' I rename to Ann and the row with index 'b' I rename to Bob. And that

brings us to the end of our module. What have we seen? Well, we started by looking at missing data. We learned how to find the missing data in our dataset, and how to inspect it. Then we saw how to remove it with the dropna method or if we don't want to remove it we can fill it with other values based on the data in our DataFrame. For this we saw two methods, fillna or interpolate. Next we saw two other types of unwanted data, outliers, for which it's very hard to give a general way to remove those, but I showed you some pointers of how to find and remove outliers, and then we looked at duplicates, for which there's also some very nice Pandas methods like drop_duplicates. Then we looked at type conversions with the astype method, which allows us to convert the type of data for a column into another data type. And finally, we saw how to tidy up our indexes, either make them continuous again or to set them based on the data in one of our columns. And that's all I have to say about cleaning data. I hope I'll see you back in the next module, in which we'll learn how to transform our data in a lot of different ways.

Transforming Data

Module Overview

Hi, welcome to the last module of this course, in which we'll see how to transform our data into something different. So you have a dataset, you've cleaned it up, let's look at some things we can do with our data. First of all, let's do some calculations. We'll see how to apply the basic math operators and how to apply mathematical functions. We'll also see how to define our own functions and apply those to our data. After that, we'll look at groupby, which is a very powerful feature in Pandas that allows us to split up our data into groups, apply functions to those groups, and then aggregate the results into a new DataFrame. Now this might sound a little bit abstract right now, but trust me, this is a really powerful and cool feature in Pandas. Then we'll move on to Pandas operations that don't change the values of your data, but the structure of your data. A lot of the time when you receive a dataset it doesn't have the structure you want. In this part of the module, we'll see several ways to restructure your data either from rows to columns, or vice-versa. Finally, we'll see several techniques to combine multiple datasets.

Demo: Mathematical Operators

So let's start with our first demo in which we'll see how to apply basic math operations, and how to apply functions, and we'll look both at mathematical functions from the numpy library and

functions that we wrote ourselves. As always, I start by importing numpy in Pandas, and here I'm creating a new DataFrame with five rows and four columns, and it only contains ones, and I create this using the numpy function ones. And I set the names of the columns to a list of characters, a, b, c, d. Now the mast basic thing you can probably do is apply a basic mathematical operator to the entire DataFrame, so let's say, for example, multiplying the entire DataFrame by 2. And this multiplies every cell in the DataFrame by 2. Now this doesn't update the DataFrame itself, so if you want to do that we have to use an equals sign here. And then to display the DataFrame I have to say df again. So after executing these two lines, we see that the DataFrame now contains the value 2 for every cell. We can also use these operations on single rows or cells. So, for example, I can select a row with loc and divide it by 2, or I can select a column and subtract 1. And here you see that only the column and the row we have selected have been changed. Now this is all very straightforward, but I can use these mathematical operations on more complex data types as well, so let's see what happens if we use one of these operations on two DataFrames instead of a DataFrame and just a number. So to do that, of course, I have to start by creating a new DataFrame. So here I have a DataFrame, again, with only ones in it, and it has three rows and two columns, and the columns in this case are called d and e, and the rows are 2, 4, and 5, so we have a non-continuous index here. Now let's see what happens if I try to add this second DataFrame to the first DataFrame by saying df + df2. And the result here may surprise you. We get a lot of not a numbers, and we get only two values. So how does this work? Well, first of all it's important to realize that Pandas always works in terms of indices and labels. Now when I add two DataFrames it will take every position in the first DataFrame and try to find the same position in the second DataFrame and add those. So scrolling up a bit, the first cell in df1 has column a and row 0. Pandas will try to locate a cell in df2 that has row 0 and column a as well. Now, as you can see in df2, there's no such thing, there is no column a, and there's also no row 0, but in the output we do see a column a and a row 0. So basically, Pandas takes every row and every column from both DataFrames, and puts them all in the output. So the columns a, b, c, d were all in our first DataFrame, so they all show up in the output, but column e is only in DataFrame 2 and it still shows up in the output. And then the actual calculation is only executed for those cells that are in both DataFrames. And in this example there's only two cells that overlap. Again, scrolling up, there's a d2 in DataFrame 1 and there's a d2 in DataFrame 2, so these values we can now add, So that's 2 + 1 is 3. Let's look at the output, and we see the result in cell d2 is 3. Similarly, there's a d4 in both DataFrames. Those get added and the result shows up in the output. So basically, what happens is that Pandas takes what we call the Cartesian product of all the rows and columns, so basically there will be a row for every row index in both of the inputs, and there will be a column for all the column labels in both of the inputs. Now for every combination of a row and column

that doesn't contain a value for both inputs, we get NaN, as you see here. Now the second question is, what will happen if I use a mathematical operation on two Series. So let's take a row from both DataFrames, and basically here we see the same thing. There's five values here, one for every column from each of the inputs, so a, b, c, d come from our first DataFrame, and the e column is added by our second DataFrame, and again, we only see a result for the case where there's a value in both inputs, and in all other cases the result is NaN. And then, of course, there's the question of operating on a DataFrame in a series, and the classical example for this is removing the mean from your columns. As you may remember, the mean of a DataFrame returns the mean value for every column. So here we have a Series that has a value for each column, and that means that when I say df - df. mean, I'm subtracting a Series from a DataFrame. And this probably does what you would expect it to do. For every column it takes the corresponding value from the Series and subtracts it from each of its cells. So for column a we take the a value from our df. mean series and subtract it from each cell in column a, and for the b column we take the b value from df. mean and subtract it from every cell in b, etc., etc. Now here I have a different example of subtracting a Series from a DataFrame. I'm creating a series that has four values, but in this case the labels of the values are a, b, e, and f, so the labels don't completely correspond with the column names of our DataFrame. When I execute this, we see that the same thing happens as we saw before when we tried to add two DataFrames. For every column label that isn't in both of the inputs, we get NaN. Now what if I want to apply one of these operations on the rows instead of the columns, so I want to take the mean for every row and then subtract it from every row? Well we know that to take the mean for every row we can say df. mean with axis=1, but if I try to subtract this from my DataFrame, I get a result with only null values in this. This is because my df. mean works per row, so the indices of my df. mean are the index labels for my rows, which is 0, 1, 2, 3, 4, and these are different from the column labels. So subtraction like this only works per column. If we wanted to work over the other axis, I have to use different methods. I have to say df. sub. So df. sub is a DataFrame method that does subtraction and basically it does the same as the minus symbol, but it gives us some more options. So here, for example, I can give an axis argument, and this allows me to subtract the mean row-wise for every row for my DataFrame.

Demo: Function Application

So we've seen how we can apply simple mathematical operations, but what if you want to do some more complex mathematics? Well, let me show you some examples. First of all, here I'm creating a DataFrame, and I'm adding two columns, sin and cos. And the sin column I fill with

values from 0 to 5 pi with a step size of 0 to 0. 01, and the cos column I fill similarly, but here I start at 0. 5 times pi and I go all the way up to 5. 5 times pi. Now right now this DataFrame contains simply a range of numbers that counts from 0 upwards. Now let's apply a function to this. I want to apply the sin function to each cell in my DataFrame. Now numpy already has these kinds of functions for us, so numpy offers a very complete set of mathematical operations we can apply to our data. So in this case I can simply say np. sin. df. So np. sin is what numpy calls a universal function, and this is the kind of function that we can simply apply to our DataFrame and it will calculate the sin for every single value in our DataFrame. Now to show you the results, let's immediately plot it. Of course, I have to say matplotlib inline to actually show the plots, and then I say df. plots. So here we can see that all the values I generated at first have now had the sin function applied to them, and this gives us this beautiful graph here. Now as I just said, numpy is very complete and it contains functions for just about everything, not just trigonometry, but exponentials, logarithms, etc., etc. So that's very nice if you want to use a well-known standard mathematical function, but what if you have a function that you wrote yourself that you want to apply to your data? As an example here, I wrote a function iqr that takes a column from a DataFrame as input, and it calculates the interquartile range. Now, if you don't know what that means, it doesn't really matter, the point here is I wrote a function that works on a column of a DataFrame, and to apply this to my DataFrame I can simply say df. apply iqr. And here I don't use parentheses, because if I would use parentheses here like this I would be calling my function, and that's not what I want to do. I want to pass the function to df. apply, and then df. apply will call the function for me on each column in my DataFrame. So now by executing this I apply my iqr function to each of the columns in my DataFrame, and it returns these two values here. Now, of course, you can also apply a function to each of the rows, and in that case you add the axis=1 argument. Now there's a lot of rows here, and they all contain only two values, so this doesn't really make sense, so let's remove this. Very well. Now what if you wrote a function that you don't want to apply to a column, but you want to apply it to every single value in your DataFrame? So here I wrote a very simple function. It takes a value, x, adds. 25 to it, and then calculates the absolute value of that, and I want to apply this function to every single cell in my DataFrame. Now I cannot use the apply method because it works for rows or columns. Now to apply a function to every cell, you say df. applymap, and again, you pass the name of your function. And don't forget, no parentheses here, because, again, we want applymap to apply the function for us, so we're not going to run the function right here. Now I'm just going to plot this immediately so we can see the output. So this has applied my somefunc function to every single cell in my DataFrame, and here we see the result.

Review: Math Operators and Function Application

So let's go over what we've just seen. First of all, here's an example of using a binary operator with two Series. In this case I'm using the plus operator, and as you can see, the first Series has indices a, b, c, and the second Series has indices b, e, and the result of adding these two together is a new Series object that contains all of the indices from both inputs. Now, the one green row here has the index b, and this is the only index that is contained in both of the inputs. That's why I made it green in the output because it's the only one that actually has a result. All the other rows, a, c, and e have as an output value NaN. So, the general rule is that the results will only be filled for those indices that are in both inputs. In all other rows, the result will be NaN. Here we see a similar example, but I'm multiplying two DataFrames. The rules are more or less the same though. So I've used colors here to make it a little bit more clear what part of the results come from which input. So, for example, column a in the output is blue because it comes from the first input, but because the second input doesn't have column a, all the values in the output in column a will be NaN. And, again, in this example there's only one cell that actually contains a value, and that's cell b1. This is, of course, because both the first and the second input have a column b and a row 1. Now this is the third example, and here we're operating on a DataFrame and a Series, and in this case it works a little bit differently. Now what I like to do in this case, I like to rotate the Series in my mind so that it's horizontal instead of vertical. You see, in this operation the indices of my series are matched on the column labels of my DataFrame. You can also see this in the output. There's a column a in the output, and that comes from the first label a from my Series. Now these kinds of operations are executed per row of my DataFrame. So we take the first row of my DataFrame, apply the operation with our Series, and that becomes the first row of my results. Then we take the second row of the DataFrame, apply the operation with the Series again, and that becomes the second row, etc., etc. Now what if you want to apply a mathematical operation on a DataFrame and a Series, but you want to apply the Series to every column instead of every row? Well, that's not possible with the normal mathematical operators, so for those cases Pandas gives us a set of functions. All of these support the axis argument so you can tell them to operate on the columns instead of the rows. There's df. add if you want to add a Series to your DataFrame, and there's df. radd if you want this, but you need your DataFrame to be on the right-hand side of the operator. Similarly, there's df. sub for subtracting something from your DataFrame, and there's df. rsub if you want your DataFrame to be on the right-hand side of the minus sign. There's mul and rmul for multiplication, and div and rdiv for division. There's also floordiv and rfloordiv for Python floor division, and finally there's functions for powers and the modulo function. Numpy offers us a large number of mathematical functions that we can apply

directly on our DataFrame or Series, and in numpy these functions are called ufuncs, for universal functions. So here in the example you see np. sin for computing the sin function and np. exp for computing the exponential function. Now for a complete list of the ufuncs that numpy has to offer, here's a link to the numpy documentation. Now if you wrote a function yourself that you want to apply to your data, or maybe there's a function from a different library that you want to apply, there's the df. applymap function. And what you do is you pass your function to df. applymap, but you don't put parentheses after your function name. So this way you pass the function as an object to applymap, and it will call applymap for every cell in your DataFrame, and this will return a new DataFrame with the results. Now, an important thing to realize is that you can also do this on a Series object, but in that case the method on the Series is called apply and not applymap. And to make things a little bit more confusing, a DataFrame also has an apply method and you can also pass a function to it, but this will apply the function to every column of your DataFrame or to every row of your DataFrame. So DataFrame. apply and Series. apply have the same name, but they do different things, because Series. apply applies a function to every value in every cell of the Series, whereas DataFrame. apply applies a function to whole rows and columns. Now the first two examples here show you how to apply a function for every column of your DataFrame, and the third example has the axis argument and this will make apply work for every row.

Demo: Grouping and Aggregation with groupby()

Let's take a look at grouping and aggregation with groupby, which is one of the coolest and most powerful features in Pandas. So let's take a look at a very powerful mechanism in Pandas called groupby. And to show you a demo, here we have the athletes dataset, which we're now familiar with, and I cleaned it up a little bit so there won't be any problems with duplicates or having the wrong data types or any of that. Now what groupby lets us do is it lets us break down our dataset into groups and then apply calculations to these groups. Let's just dive straight into an example. If I say athletes. groupby nationality, this returns me a DataFrameGroupBy object, and this by itself tells me nothing. So let's just assign this to a variable g, and I'm going to say g. sum, and this returns me a new DataFrame. And in this DataFrame the first thing that strikes me is that the index is now nationalities. So what groupby does is it takes our original data, splits it up into groups by nationality. So basically you can think of the groupby object containing a set of smaller DataFrames, each with one of our nationalities. So it takes every value from the nationality column and makes new little DataFrames for each of the values there. And then on these groups we can do calculations. So here we see the sums of all the ids for every athlete from Afghanistan.

Of course, this doesn't really make sense, but that's what it does. This is the sum of all their heights, and this is the sum of all their weights. And the same data is in here for all my different nationalities. So scrolling down, for example, here we see the sums of heights and weights for Belgium. Of course, the most interesting thing here is the sum of the medals, so how many gold medals and silver medals did a country win. And since g. sum here is a DataFrame, of course I can ask for these columns here. So I can select just a set of columns for my DataFrame. But actually it's smarter to do this on the groupby object, so I'm just going to move this selection here to the place where I define my groupby object. And why this is better? Well, this selects the columns before I start doing my calculations, so now the sum will only be done for the three columns that I'm asking about. Now let's look at a different example. Let's group our athletes by sport. Again, I'm assigning the groupby object to the variable g, and let's say I'm only interested in weights for now. And on this groupby object I can do different things than just sum. Let's ask for the mean weight. And in this case, because we only selected a single column, I get a series with the mean weight per sport for my dataset. You know what, let's also select the height, and in this case I get a nice DataFrame. But apart from selecting multiple columns, I can also select multiple levels to groupby, so let's not just group by sport, but also be gender. So to do this, I pass to groupby a list, so using square brackets again, with the name of multiple columns I want to group on. So here I first group by sport, and then by sex. So let's again go over what this actually does. Groupby takes our DataFrame and makes nice little DataFrames for every combination of sport and gender. So it makes a DataFrame for female archery players, and one for male archery players, and one for female badminton players, and one for male badminton players, etc., etc. Then from there we select the weight and height columns only, and after that when I say g. mean, for every one of these groupings we calculate the means of these two columns. So then we have a mean, again, for every female archery player, and a mean for every male archery player, etc., etc., and at the end, all of this gets combined into this DataFrame you see here.

Review: groupby()

So let's visualize what happens when I do a groupby. Let's take a look at this DataFrame here. We have a number of students, and we have a class column that tells me per student in which class he or she belongs. Now if I say df. groupby class, this splits up my DataFrame into groups per class. So internally the groupby object will have groups of data for my DataFrame. It doesn't really make internal DataFrames, but it's a nice way to think of it like that. Now the groupby object at this point hasn't done any calculations yet. It didn't even break up my data into groups really. It only starts doing real calculations when I call an aggregation function on it, like g. mean.

At this point, it calculates means for all the columns in each of the groups, and then creates a new DataFrame out of that. So this results in a DataFrame with a row for every group, and then in each row for every column where we can calculate the mean, you will see the means. Now, in many cases you're only interested in the data in some of your columns, and the best thing to do here is to select these columns immediately when you make your groupby object. So like in the example here, I call groupby and immediately after the call to groupby I already select the columns I'm interested in. So here I say groupby sport and I select only the gold column, and this will cause groupby to only do calculations for that column. So then when I say. sum, only the sums for the gold column will be calculated. You can also groupby multiple columns. So the second example here shows you a call to groupby where I pass a list of columns, and in this case I'm grouping by both sport and nationality. Now if you do a groupby operation where you group on multiple columns, the result will have a multi-level index, like the example shown here. Now a multi-level index is something I'm not talking about a lot in this course, but the general concept is really easy to understand. In this case we have an index with two levels, sport and sex, and first we see a value for the sport index, boxing, and below that we see the different values for the sex column, female and male. Then we have another value in the sport index, cycling, and again below that we have the values for the second level index, sex. So if you use groupby and you select multiple columns, you will get a multi-level index, and in a moment we'll see some other operations that either result in or use multi-level indexes. Now what kind of functions can you call on a groupby object? Well, there's a number of functions that are a part of the groupby class, and those are optimized functions. You can see them listed above here. They're functions like count, sum, mean, min, max, and the standard deviation, etc., etc. But you can actually call all the functions on the underlying object as well. So if you groupby on a DataFrame, you can call all DataFrame functions on your groupby objects. Or you can use groupby to apply to actually call a function you defined yourself on the groupby object.

Demo: Structural Transformations with stack() and unstack()

So we've seen how we can transform our data by applying functions to it, but let's see how we can transform the structure of our DataFrame. So we'll see how to move our data from our columns to our rows, or from our rows to our columns, and there's four functions that are of interest here, stack, unstack, pivot, and melt. Sometimes you just don't agree with the way your data is structured into rows and columns. As an example, let's take this dataset, monthly_data.csv, and here it shows a row for every year, and then the months are in the columns. So here for 2009 we have a value for January and one for February, etc., etc., and then the values for 2010

are on the next line. Now this is vey nice, but what if I want to see the chronological order of all these values, let's say I want to plot how these values change over time. Then it would nicer if all these values are in a single column. Now there's a function to do just that, but before we do this let's remove the year column and put it in the index. So for that I have the set_index function, as you may remember, and I use inplace=True to immediately update our DataFrame. So now we have the years as the index, and then I can call the. stack method, and this will move all the data from my rows into a single column. Let me show you. So now we have a Series that has all our data points below each other. So first we get the values for the first row, and then for the second row, etc., etc. Our values are indexed first by year, and then inside the year they're indexed by month. Now I'm not saying a lot in this course about multi-level indexing, but we have already seen them with groupby. So groupby and stack are two typical scenarios where you will see a multi-level index. So after calling stack, it's also suddenly possible to do calculations for the entire dataset. So if I want to know the sum of all my data, I can simply say m. stack. sum. And because after stack all my data will be in a single column, calling sum on that column will give me the sum of all my data. Now stack also has a reverse called unstack, and that will put my data from my columns into my rows. As an example, let's look at what we saw in the previous demo the mean weight of athletes grouped by sport and gender. So here we have a value for every combination of sport and gender, and all these values are below each other in a single column. Now, as I just said, unstack will move values from columns into rows. So here we have a multi-level index, and the inner level is our gender. So the first level is the sport and the second level is the gender, and unstack, by default, will take the inner level of our index. So we will get two columns, one for females and one for males. So let's execute this and here we see the same data, but instead of everything being in a single column, it's now in two rows.

Demo: Structural Transformations with pivot() and melt()

Now here's another scenario where I don't like the way my data is structured. Now I'm creating a DataFrame here, and remember, I'm making this file available for download so you don't have to type along here, and the DataFrame here has data about products sold in a store. And as you can see, for every product id here I have different lines, and then per line I give some item. So here I have a product where I have a price, a unit, and a stock. Now although you do sometimes come across data that is structured this way, it's not really the way I like it, because in my opinion, every product should be on its own row, and then all its properties in the columns. Of course, as always, Pandas has a function for this. It's called pivot. Now pivot is a little bit like unstack in the sense that it moves my data from columns into rows, so I will get a new DataFrame that is both wider

and shorter because we will only have one row per product. And as the first argument I have to tell it what my row index will be. So I'm going to say please take the id's as row indices, so for every unique id we will now get a row. And the second argument will be where to get my column names. So I'm going to say please take your column names from the item column. So this means I will now have columns prize, unit, and stock. And as the third argument, I can pass the column with which to fill my DataFrame, so the one that actually contains my values, and of course, this is the value column. But because the value column is also the only column that is left after id and item, I can also completely leave it out, so let's just leave out this third argument. And here we see our same product data again, but now we can really clearly see that there are exactly three products, and the id's for the products are now in the index. Then per item, price, stock, and unit we now have three columns, and in these columns, we have the data that was previously in the value column. Now to see the opposite of pivot, let's take a look at the grades dataset. We've seen this before. It lists test scores for students in a classroom. Now in this case what I want to do is quite similar to the stack function. I want to move all my grades into a single column, but I also want an extra column that tells me from which column the data came, so I want an extra column that tells me per grade for which test that grade was. So to do this I call grades. melt. And then we see here all our grades with a separate column that tells us for which test they are. But the problem here is that we now lost the information about the students, so we can't see for which student each grade is. Let's fix this. First I'm going to call grades. reset_index. And the result of this is that now we have an index from 0 to 4, and the student names are now in a column called index. So now if I call grades. melt I get something unexpected. Basically I get a single column with all the values from my dataset, and it tells me from which column they are. And this is not exactly what I want, because I want these names in a separate column. And to fix this I can pass to melt a list of columns that I want to use to identify what values we have. So basically I say here this index column with these names, those are not just values, they are used to identify what kind of value we have. So the 6 here we see, we don't just want to know that it's for test 1, we also want to know that it's a grade for Mary. So here I say id_vars is index, and note that this is inside a list so there's square brackets here, and then executing this now we have the result I want. We see all our values in a single column, we have a column that contains the name for each student for each grade, and we also have a column that identifies the test for each grade.

Review: Structural Transformations

Let's take a slightly more formal look at how pivot works. Pivot transforms one column of data into multiple different columns. And you can give it three arguments. The first one is the index

column, and you pass the name of a column and for each unique value in that column there will be an index for one of your rows. The second argument is the name of a column that you will use for the column labels, so from that column it will take unique values and for every unique value there will be a column. And the third argument is the column from where the data will be taken. So a new DataFrame will be constructed with rows for every value from the first column, columns for every value from the second column, and the values will come from the third column. Now if this is a little bit abstract, let's visualize this. So here I have a DataFrame with product id's, and then for every product we have prices or stocks, and we can know if the value in the value column is a price or a stock by whatever is listed in the item column. Now with pivot I can convert this into a DataFrame where I have one row for every product. To do this, I say df. pivot, and as the first argument I give the name of the prod_id column. So this will take every unique product id and make one row. And the second argument is item, and this means that for every unique value in the item column there will be a column in the output. So when I run this there's a new DataFrame created, and as you can see, the row labels 1, 2, and 3 are taken from the prod_id column in the input. The column names, price and stock, are from the item column, and the values are taken and filled in in the correct places. So the price for product 1 is 11, and the stock for product 2 is 2, etc., etc. The opposite of pivot is melt. It moves data from multiple columns into a single column, and the most important argument that you can give to melt is called id_vars. Id_vars identifies the column that contains id's in your dataset, so in our previous example, this would be the product id column. Now melt works by taking all the values that are not in this id_vars column, and puts them all in a single column, and then it takes the column labels from your original data and puts these in a separate column under the name variable. Now let's look at a little visualization to make this a little bit more clear. Let's take the same example as with pivot, but let's work backwards. So here I have the DataFrame that was the result of our pivot operation a moment ago. And in this case, before I can use melt I want to move my index into one of my columns, because in this case my index contains my product id's, but melt only looks at the data in my columns, so I want to move my index into a column. So I start by preparing the DataFrame, and I call df. reset_index on it, so this moves my index into a new column, index, and then I say. rename and I rename the new index column to prod_id. So this results in the DataFrame on the right-hand side here. I have a new index 0, 1, 2, which is simply a range, and I have a new column, prod_id, that contains the indices from the original DataFrame. So now all the data I want is in my columns, and I can call melt. So now I say df. melt and I pass it the id_vars argument, and I tell it that my product id column contains the id's. And you can see the result here on the right-hand side. I now have a row for every combination of a product id and a price or a stock. So price and stock here are values that are taken from the column labels of our original DataFrame, and the

corresponding values are in the value column. So now you can see that the price for product 1 is 11, and the stock for product 2 is 2, etc., etc. Another function we saw is stack, and this is quite similar to melt in that it moves data from multiple columns into a single column. The difference is that stack simply puts all your data in a single column and adds a multi-level index. So now we have all our values below each other, and we have a multi-level index where the first level is our original index, so the product id's, and the second level tells us whether it's a price or a stock. And, of course, stack also have an opposite called unstack. This looks at your innermost index level, so in this case it's the index that says price or stock, and for every unique value in this index it contains a column. So in the result we now have a column stock and a column price, so as you can see, this is exactly the opposite as the stack method from the previous slide.

Demo: Combining Datastructures

And that brings us to our last demo. Let's see how we can combine data from multiple data structures. So here I have my familiar grades dataset. Now let's say I have a Series that contains the result from a new test I took. So here I have a Series with results for every student, and to add this as a column I can simply assign this to a new column, and this simply adds the Series as a new column to our DataFrame. So this is a familiar scenario, and in a similar way we can add a row to this DataFrame. In this case I will use loc to do row-based indexing. So let's add grades for a student called Bob. Now in this case I can use a simple list to create the new row, but this wouldn't have worked for the column, because when I add a column I need something that contains indices, so I have to use a Series. Now just like Python lists, DataFrame also has an append method which lets us append rows to the end of our DataFrame. But when I pass a simple list to it, it doesn't work as I expected. You see, when I pass a list to append it interprets every element of this list as a single row, so it will add a row for the value 5 and a row for the value 4, etc., etc. And because these elements don't have indices, we get this strange result here where there's a new column called 0, and the new rows also have indices like 0, 1, 2, 3. Now to do this correctly, I actually want a data structure that has indices, so let's make a Series again. So here I've created a new Series called new_row, and it contains indices test_1, test_2, test_3, test_4, so we should be able to append this to my DataFrame. Let's see if it works. But now I get an error. It says it can only append a Series if ignore_index is true or if the Series has a name. Basically, Pandas wants to know how to set the row_index for this new_row, and the best way to do this is to actually give our Series a name. So the name of this Series will be used as the row_index. So here I'll give it the name of a student, Kim, and then if I append it, now the grades for Kim have been appended to my DataFrame. So that tells us how to add single dimensional

data to our DataFrame in the sense of adding a row or a column, but what if I want to combine multiple DataFrames into one DataFrame? Well, to show you that, I'm going to add student numbers, so student id's to my DataFrame. So here I'm creating a new column, stud_nr, and I'm adding an id for every student, and then in the second line here I'm selecting the columns from my grades DataFrame in a specific order. So this lets me move the student number column to the first position, because otherwise if I wouldn't do this, then the student number column would now be the last column in my DataFrame, and I just think that's ugly. Very well, so now we have grades, and student numbers, and let's create a second DataFrame that I want to merge with this DataFrame. So here's a new DataFrame that has a different set of student numbers and two columns for two different exams. Now, I want to combine all of this data into a single new dataset, and for that there's the merge method. If I call grades. merge other, this is my result. I see columns for all my tests and my exams so that looks good, but why do I only see four rows? Well, the answer is simple, grades. merge, by default, does something we call an inner join, and what that means is it looks for a column that's present in both inputs, so in this case that's the student number column, and then it takes only those values from that column that are present in both inputs, so in this case only the four students that we see here were present in both inputs. Now merge takes an extra argument, how, and that lets me do different kinds of joins. So I can do a left join by saying how is left, and in that case I get a new DataFrame for all the student numbers that were present in my left input, so in the grades DataFrame. And you can also see that in the output now there are two students that have NaN for the two exams from my new DataFrame, because those two students are not present in my new DataFrame. Similarly, if I say how is right, I only get the students that are present in my second DataFrame, and here you can see that the four tests are empty for my last two students. Now there's also a final option called an outer join, and this will simply contain all the data from both inputs. And that's the end of this course. What have we seen in this module? Well, we started by doing calculations on our data. We saw basic math operations, we applied mathematical functions, and we wrote our own functions and applied those to our DataFrames. Then we took a good look at groupby and all the different kinds of operations you can do with that. We saw several ways to transform the structure of a DataFrame by moving data from rows to columns, or from columns to rows, and in the last demo we saw how we can combine multiple data structures into a single DataFrame. Thank you for watching this course. My name is Reindertjan Ekker for Pluralsight.

Course author

Reindert-Jan Ekker

After studying Artificial Intelligence, Reindert-Jan has worked in many different roles in the software development life cycle, from tester and developer to Scrum Master. Then he decided to go...

Course info

Level	Intermediate
Rating	★★★★★ (35)
My rating	★★★★★
Duration	2h 15m
Released	24 May 2018

Share course