Advanced Pandas

by Paweł Kordek

Start Course

Bookmark Add to Channel Download Course

Table of contents Description **Transcript** Exercise files Discussion Related

Course Overview

Course Overview

Hi everyone. My name is Pawel Kordek, and welcome to my course, Advanced Pandas. I am a software engineer at Feedzai where I develop data processing applications to fight financial fraud. Nowadays, Python is getting more and more popular in the field of data analysis, and pandas is becoming the preferred library for many data scientists. In this course, we are going to explore non-trivial parts of pandas and learn how to use it in order to get full control over your datasets. Some of the major topics that we will cover include joins, handling multidimensional data, time series and windows, and plotting. By the end of this course, you will know how to handle complex datasets and analyze your data in a principled way with pandas. Before beginning the course, you should be familiar with Python and have basic experience with pandas, for example, from my course on Pandas Fundamentals. I hope you'll join me on this journey to learn pandas with the Advanced Pandas course, at Pluralsight.

Overview

Overview

Hello. My name is Pawel, and I would like to invite you to watch my course on Advanced Pandas, in which you will discover non-trivial parts of the pandas data analysis library for Python. This course is a natural continuation of the Pandas Fundamentals course, but if you already have some experience with pandas or any other data analysis tool, like R, you should be fine to follow without having watched that one. Pandas is a powerful data analysis library that helps anyone take fine, great control over their data by exposing its rich feature set in Python. On the other hand, it must be said that this requires practice, as some more advanced dataflows make a struggle for inexperienced users. Skills acquired during this course will help you mitigate that. This should be especially appealing to data scientists, business analysts, or any data professionals who are hub based, as they will benefit from the ability to quickly iterate over data and swiftly translate their ideas into code, doing more with less and saving time. I have selected several topics that I believe are essential for advanced data analysis workflows with pandas. We will discuss them in depth, introducing all the necessary concepts, and exploring how to translate them into code. This course will give you the opportunity to learn and practice the following things. We will start smoothly by exploring our dataset, loading relevant parts and cleaning it a bit. We will then move onto combining multiple datasets into one by performing database-style joins. In the next part, we will see how to easily move between long and wide data formats. You will see that it is not only good to have your data in one table, but it is also important to organize it well inside this table. From there, it is very close to pandas special feature, multi-indexing, which brings analysis of more complex data to a whole new level. Having our data in the perfect shape, we'll see that pandas has a lot of functionalities to deal with time series data, and we will learn them well. Then, it would be a good moment to introduce window operations, which let us get more insights into temporal data. This will be already a lot, and with all this knowledge, we will be ready to move on to the final module where you will see how to prepare great, easily readable, and interactive plots. I am sure that after this course you will face any dataset with confidence, and you will have all the skills needed to shape and analyze it accordingly to your needs with ease. I promise it will be well-invested time.

Preparing Your Dataset

Introduction to the Dataset

Now you will get to know the dataset that will accompany us during this course. I will present to you how it is organized and describe the parts that will be relevant for us. To complete this introduction, we will use pandas to load our data and shape it in a way that will be most suitable for your learning experience. Let's go. To have a good ground for learning a data analysis library, we will need a solid dataset. As you may be aware, nowadays there are lots of publicly available datasets from all sorts of domains, be it finance, climate, transportation, you name it, but to explore pandas well, we need something carefully selected that has enough properties to be used for presentation of all the features that we will cover. I'm telling you this since apart from pandas' features, I would like you to additionally learn to carefully look at your dataset. For example, to learn how to work with time series data, we need some set of records that contain information about time, and not every dataset will provide it. For instance, the classic Titanic percentage survival dataset used in many data science demos does not have any column that could be used to present an aggregation by some time interval. On the other hand, if we use dollar to pound exchange rate, we would have perfect data for playing with time series, but unfortunately, that would not be the best for the hierarchical indexing presentations, as the data has only one dimension. Therefore, if I want to stay with one dataset throughout the course, not every single one will suit our need. Then what should we choose? From my experience, one very good type of data that fits well demonstrations purposes, is sports data. If you are into sports, great. If not, don't worry, and believe me that neither am I, but this does not prevent us from using some sports dataset and having fun with it. In this course, we will use a hockey database. This is a database from OpenSource Sports, more specifically, Hockey Databank project. For more information, you can go to the project's Yahoo group. It contains seasonal data about American ice hockey leagues, most notably NHL. NHL stands for National Hockey League, which is the primary professional ice hockey league in the world. I will now give you an overview of how this dataset is organized, what it contains, and what all this information means. I may introduce some very basic hockey concepts, but I will carry all to minimal, so don't expect a comprehensive hockey primer. Also, please forgive me if I mix up some details. I am not an expert, but I am definitely excited about what we can do with this data.

Introduction to the Dataset Part 2

Our data is scattered among many different files, but the good news is that we won't actually need big portions of it. All files reside in one main folder, and all the data is stored in CSV format. The folder also contains two files that give complete description of the dataset, README and notes. The first one contains essential information, while the second one contains some additional notes that are not crucial to understanding of the dataset. I encourage you to take a look inside; however, this is not mandatory, as I am now going to explain which parts we will need and describe them in more detail. In case you are ever lost or need a reference, please check the README file. As a side note, whenever I need to peak inside a CSV file, I usually use pandas directly, but if you like, you can always inspect it using some spreadsheet software. The parts of the dataset that we will be using can be divided into two groups. One describes individual player status state, whereas the other is concerned with teams' data. We'll prepare some DataFrames for both kinds and use them depending on our needs. Let's start with players. First file that we need is the Master. csv, so let's take a look at its properties. The file has 31 columns, but for the preview purposes, I've just selected a few of them so that you can get familiar. Most of them will be not necessary, and we will draw them directly or after using them for filtering. The main property of this table is that it allows for matching player's ID with their full name, and this is in fact the only purpose for which we will need it. Let's take a look at the first row. It simply contains mapping from the ID to the name and surname and also some other player-specific information, like date of birth, which we may want to keep in case we would like to play a little bit when discovering time series capabilities that pandas has to offer. In the second row, we see that it contains also coachID. It means that this player during his career has also served as a coach for some team. In the third row, however, there is no playerID. It means that we are dealing with someone that was never an NHL player, but only a coach. We will drop such records, as players data will be sufficient for our purposes. For now, this is all that you have to know about this table. We will also perform filtering on some other columns, but let's wait with this for the demo, as I wanted to have a good overview of our dataset beforehand. Next file is Scoring. csv. It contains, no surprise, scoring information for players. Each row describes a player's performance in a single season. This contains a lot of information, and it may be quite intimidating at first sight. But don't worry. I will now quickly explain what the columns mean, leaving out the ones that we don't need. So, the file looks more or less like this. These cryptic shortcuts are just simple numbers describing each player's season. Of course, there are more columns in this data table, but we will not use them directly. Let's take the first row, for example, and go through these columns one by one. We have some playerID, and we already know we will match it with full name from the Master. csv file. Then we have a year which corresponds to season. As you can see, each player can have many roles, each for one season he played. Then there is an ID of the player's team for that season. Next is the position, and what each of position acronyms means is irrelevant for now. GP stands for games played in total during the season. G stands for goals scored. A stands for assists, so it tells how many times the player passed on to a teammate who then scored a goal. Pts stands for points, which is the sum of goals and assists. And finally, we have shots on goal, which simple means how many times player tried to score. Take your time to look at this data. We are halfway through, and we can now proceed with teams. Next data table we will need is Teams. csv. Each row consists of team ID, season year, full name, and some statistics for that season. Statistics we will skip, as if we need it, we will obtain it with more granularity from the table we discuss next. We will use this file only as a mapping between the team ID and its full name. All team statistics we'll be interested in are contained in the last table that we are going to use and discuss, namely, TeamSplits. csv, which is quite detailed. It contains team performance statistics and splits it between months during the season so that we can see how the team's performance was changing. Here we can see an example for October. We have wins, losses, ties, and overtime losses. It will definitely allow us to have lots of fun when learning time series capabilities of the pandas library. This is all that you have to know about the dataset for now. It's time to finish dry descriptions and start coding. I know that you may not yet feel 100% confident navigating through the dataset, but the upcoming demo will help you, and I promise to not leave anything unexplained.

Demo 1

We will now load and clean all the relevant data tables, preparing our data for analysis during the rest of the course. We will also learn some new useful pandas functions and methods. We are now in the first notebook as in the demo supplied with the course. The only difference being that I am just displaying it in a presentation mode. If you have used Jupyter Notebook before, everything should look familiar. And if not, you will quickly grasp its basic functionality after playing a bit with the demos. For instruction on how to start a notebook, please refer to the course materials. I also encourage you to look up keyboard shortcuts for Jupyter, as using them tends to make moving around more convenient. We start with reading the master file, and it is often good to use ______ to get a feeling of how the data looks like. Here it shows us first five rows, but we can also see that there are too many columns to display, so they have been cut out. So, in fact, the first thing I usually do is to check how many rows and columns I have loaded. And for this, I use the shape property of the DataFrame. So we have almost 8, 000 rows and 31 columns. Let's now use the columns property of the DataFrame to display names of all of them. Looking at them, we have to decide which of them we want to keep, which to discard completely, and which we will need for filtering first. As I said before, we will want to keep only entries with a valid playerID. We can do a quick inspection to confirm that there are indeed rows with missing playerID. To do this, we can use pandas isnull function that will tell us whether there is a value or not. Then using value_counts, we can see that there are 241 rows for which the data is missing, and we can get rid of them. Here you can also see that sometimes things can be written in pandas in different ways and produce the same result, as alternatively, we could use the pipe method. This method takes an object, in this case a series, and uses it as an argument of the function that we passed, in this case, isnull. The benefit of this is that when we want to do more operations on a pandas object in one go, we can chain them in an elegant way and keep a readable indentation. Just know that the braces are necessary to avoid indentation errors. You are free to use whatever you want, as the output is always the same. I will use pipe throughout this course whenever it makes sense. Now let's leave out records with no playerID. We could use isnull again, but there is a simpler way to do this. Pandas DataFrames have a dropna method. It allows us to get rid of all rows that have a missing value in the specified column. Let's do that and override our master object. Know that I'm keeping the original DataFrame for future reference since we will revisit it later for a moment. The DataFrame is smaller now, but there are definitely some more rows that we can get rid of. Take a close look at the four columns that contain NHL and WHA in their names. WHA refers to some old alternative hockey league, and first and last prefixes tell when a player was active in the given league. First thing, we don't want WHA data. NHL is enough. So let's drop the columns that have no data for NHL years. We can use dropna again, passing our column names. But be careful. We need to supply a second argument, how="all". It means that for a row to be dropped both columns have to contain no data. If we don't supply it, pandas will use the default and drop if any of these columns contains empty data, which is not what we want. Actually, in this case, it would be enough to drop only the first NHL column since any player who is in the dataset and played in the NHL will have some number in this column, but it is very handy to know how dropna works. Now we have players that appeared at any time during NHL history. Do we need it? I suggest we limit the data we want to work with to the more or less last 40 years of NHL games. This will be more than enough for our demonstrations. To do this, we have to further filter on the last NHL column, filtering out rows we value less than 1980, since we will not be analyzing seasons before 1980. As far as rows are concerned, I think we have gotten rid of all the unnecessary data. Now let's take a look back at the columns and see which ones we want to keep, as the majority will not be necessary. I definitely want to keep playerID, full name, and lastName since otherwise this DataFrame becomes useless. As I explained, its main purpose is to associate playerID with his full name. I will also keep player's position and birth-related info as it may be useful later for other demonstrations. I am not 100% sure right now, but let's keep it. So we now want to get rid of the rest of the columns. One way of doing this is to specify columns that we want in square brackets. It works fine, but just look how much we had to type. We can do better. Pandas exposes a very useful method called filter, which in its most basic form works exactly like what we just did. But apart from that, it can also accept regex argument. As a result, it will keep only the columns that match the supplied regular expression. We first have playerID and position, then anything that starts with birth, or something that ends with Name. Vertical bars lets us separate all the alternatives. As you can see, result is the same. It's less typing, thus probably less prone to errors. Now we have our table clean and much smaller than at the beginning. It might seem obvious, but you may ask why do we want smaller DataFrames? First of all, they are easier to understand and infer information just by looking at them, but the second thing is the amount of memory they occupy. For example, if your machine has 4GB of RAM and you load a DataFrame that occupies 3GB of RAM, then you cannot load second table like this since it will not fit in RAM, which is the only way to store data for pandas, but what you can to is to take the first DataFrame and modify it so it occupies much less space. We can take a look at our original DataFrame and compare it to what we have now by using pandas' memory_usage method wrapped in a little cast on function that brings memory consumed in MiB. It is more than four times smaller. Of course, in this case, we are adding levels of 1MB so we are very, very far from exhausting memory, but it's very good to be aware of the amount of space you can save by wisely choosing what you have to keep. I would also like to quickly show you one last trick that can save you significant amount of space and speed up your operations. For example, take a look at our position column. For each row, we can have any value which is just a string, but if we take a look at all possible values, we can see that there are only seven. We can improve a lot by telling pandas that in fact this column contains only some predefined categories. In statistics, we say that we are dealing with categorical variable. To inform pandas about this, we just have to call pd. Categorical. That will do all the job. We can see what will happen to this column. Pandas will be aware that there are only seven categories, and instead of storing category value for each row, it will store its index in these seven-element list. That's where the saved memory comes from. It's time to actually apply this operation. I suggest we also repeat it for birth, country, and state. And if we repeat something, it is a good practice to write a short function for that, like here. After applying the transformations, we can see how much memory we have saved. Last thing regarding this DataFrame. If you look at the index, we now have numbers that change unpredictably since we have dropped many rows. In such cases, it is also good to set a new index on our frame. In this case, we can use playerID since this is a unique identifier of the player and is well suited for the purpose of indexing. We can easily achieve that withy set_index method and by specifying the name of the column to use. I think that now this DataFrame is ready for analysis, and after saving it to a file for future use, we can move on to the next one.

Demo 2

Next file we have to work through is Scoring. csv. As you see, it contains more than 45, 000 rows and 31 columns. As I mentioned before, we only want to analyze data for the NHL and for the seasons starting from 1980. I suggest we create a little function that does that filtering for us. When applied to the scoring data, we now get less than 30, 000 rows. Good. Now we have to keep only the columns that I've shown on the slide. First important thing to know is that each value, like goals, also appears in other places. Prefixed with PP, it means powerplay goals; with SH, shorthand goals; and with Post mean postseason goals. What it means is irrelevant, as I want to limit the data so we can focus on pandas, not on the ice hockey, and we will now drop all these columns with our prefix with PP, SH, or Post. How can we do that? We can again use filter method with regex, just modify and get ______. Here we say that we want columns which do not begin with any of these three prefixes. These question and exclamation marks are look ahead negative assertion. If you are not a regular expression expert, it's okay, and it is not crucial to understand it now. I just want you know that you can filter columns and rows by labels as you like just by tinkering with regular expressions. And by the way, they are very well documented in the Python official docs. After this operation, we still have more columns than I showed on the slide. I suggest to simply select what we want to keep by column number since now we can easily keep track of them just by looking at the DataFrame. Remember that for positional indexing we use iloc, as loc is for label-based indexing. Now we have the data that we need. To repeat with practices, we can also change team ID to categorical variable. Differently from the previous DataFrame, I would not set index here to the playerID, as in this case its values are not unique since each playerID appears for each season he played in. It is possible to have index with duplicates in pandas, but let's just leave it for now. Anyway, I would prefer to have consistent numbering here incremented by one for each row. This can be easily done by using the method reset_index. Index is good now, but you can see that the old index was baked up as a column. We don't need it, and we can inform pandas about this by passing argument drop=True. You can also see that if I want to replace some DataFrame with its new version, I'm all the time reassigning to the variable. Actually, many pandas' methods take a Boolean argument called inplace, which lets you substitute that DataFrame, or series, without having to reassign. I just usually avoid it since I found the first one more readable, and there are no general benefits to using inplace, so I will continue to do so throughout this course. Just be aware that you can use inplace, and this can come in handy, for example, if you have assigned some very long name to a variable and you really don't want to write it once and once again. Good. We are done with another DataFrame. Let's write it to file and continue to the next one. Next file is Teams. csv. It contains many columns, but as I told, we just needed to match team's ID with its name for a given season. So it is as simple as selecting the three columns that are interesting for us. Just before I filter it to discard old and any nonessential entries, should we change any column's type to the categorical? Let's take a closer look at the problem so we can understand it a bit better, as some things may not be clear at the moment. How do we decide which columns should be categorical and which should not? We have 792 rows, but how many unique values for each column we have? Previously, we used value_counts for these, but pandas has another nice method for such occasion, nunique. We can just call nunique on our DataFrame, and we will see information about the number of unique values for each column. You may ask why not treat year as a categorical variable? It has only 31 unique values, so we could save memory. Quick answer is that year represents time, and because of that, it is not really suitable for use as a categorical variable. We will come back to this when discussing pandas' time series capabilities. But second column, tmID, obviously fits the profile of a categorical variable, so let's make it one. Then if tmID's categorical, isn't team's name too? Well, it depends. We can make it categorical, but it could also limit some things. For example, we could not search for a substring in a team's name or perform any operation that we normally can perform on string. I don't know exactly how we are going to use this variable, so let's keep it as it is for now and don't limit our possibilities. That's all we have to do with team's DataFrame. Let's save it and move on. We will now tackle the last file that we will need, TeamSplits. csv. We'll load the file and do a quick inspection by checking its shape and columns. If it has year and lgID columns, and it does, we can use our recent NHL filtering function to get rid of some rows. Looking at the rest of the columns, you can know that it follows the pattern that I presented on the slide, which means that we have five values split for different months. There are also columns prefixed with h and r, which split the values between the games played at home and on the road. But months are enough for us, so let's discard those. How to drop them easily? Since these are columns from 4 to 11, the best would be to use the position slide. We can select by positions, but unfortunately, there is no direct way to drop columns by positions, but we can get around this. One way to proceed is to get labels that we would want to drop simply by getting relevant part of the columns property of the DataFrame. Now that we have the labels, we can pass them to the drop method as columns argument. And it's done, quick and easy. Keep in mind that you can also use drop to discard rows by passing rows argument. We have still one column that we don't want, lgID, but with drop, it's very easy to get rid of it. One last touch to convert tmID to a categorical variable, and we can save our DataFrame to a file. With four data files processed, we are done with reading the data and can move on to the next module. I hope you have learned new things and now understand how to quickly drop unnecessary data from your tables and how to use categorical variables to not use too much memory. All the operations we have performed in this module's demos should be definitely enough for the initial preparations of most of the datasets. Take your time to digest the module, go back to any specific point if you need, and see you in the next module.

Joining Multiple Data Tables

Introduction

In this part, we will explore pandas' capabilities in terms of working with multiple tables that are related to each other, namely, database- style joins. I will illustrate the purpose of joins in a very simple example and then discuss different kinds of joins and show how pandas supports these operations. Of course, we will also have a demo. I suppose that some of the viewers may be familiar with this concept, but I am going to do a quick introduction and explain the topic, as this will help anyone that is not fully comfortable with joins or have never heard of them before. The database-style term refers to the fact that this type of operation originates from the relational database management systems, and indeed, languages designed to query relational databases have the join keyword built in. Pandas is not a database management system, but the logic behind joins is generally the same. While there is nothing magic about them, it is such a common operation in data analysis that I believe a good understanding of what it can achieve and how is without any doubt essential. Typical use case for a join is as follows. We have two tables that are somehow related to each other. Imagine that we would have players names in one table and their scores in the second one. Let's say it's for one month, and we want to combine them into one. The way to associate these records with each other is to use playerID. Having this column, we can join the tables and obtain something like that. It is very easy as the only thing we have to know and provide is the name of the column that contains information that can be used to match one row with another. This is the most basic join, and most of the time that's all that you will have to do. Unfortunately, in the real world we are very often dealing with imperfect data, and often some information that we need may be missing. How to deal with missing data in case of joins will depend on the use case, and this leads us to the next topic, which is different join types. Joins that are usually available, and pandas is not an exception here, are left, right, inner, and outer. We'll go through their use cases to demonstrate how this works. In the simple case that I have just presented, they would all give the same result. It's with the missing data when the differences start to appear. Take, for instance, such situations in which we have some different playerIDs in two tables. It may seem strange, but real-world data often comes with deficiencies. So if it's all the data there is, we have different options to get a new table. First is to surrender and move on to different tasks because of the missing data. But as a data analyst, you have to be ready to face any unpleasant dataset, so this is not an option. Speaking of the valid possibilities, as I've explained, it all depends on the use case. For instance, you may be preparing some kind report that cannot contain any missing information. In such case, the target table would look like this, only with complete rows and other discarded. Such join, in which a join field must be present in both tables, is called an inner join. That's one way of doing this, but maybe your customer, manager, or whoever says if there is a name, then it's better, but if it's missing, I still need the information about scoring. This means using records from the first table and adding to them information from the other one if possible. This is called left join, as all the join field values in the final table exactly match the ones from the left table. Left just means first in the join's naming convention. On the other hand, someone might say if there is scoring information, then it's better, but I must have all the names. In such case, we just have to swap the tables and do what we have done before. But instead of swapping the tables, we can simply order that all the field values from the second table are present, and this is called right join. That's the case if you want as much information as possible. So the join field values of the final table will be the sum of the ones of the first table and the second one. It means taking all the join field values and all the columns from both tables and trying to fill the data. If there is no respective entry, the data will be simply missing. Such operation is known as an outer join. And that's all. Any time you will want to perform a join, you will use one of the types that I have just explained. And here is a simple cheat sheet with very basic examples, which you can consult whenever you have doubts which join does what.

Joins in Pandas

Having the basics clear, we can discuss how pandas supports database-style joins for DataFrames. Basically, all the functionalities provided by a top-level function merge. It takes two DataFrames as mandatory arguments and the number of optional keyword arguments that can be used to drive how the join is perform. Some of them are mutually exclusive. One of the arguments is how. That is used to specify the type of the join. It is optional as inner is the default, but how will pandas know what is the join column? It will look at both DataFrames and search for the columns that have the same name. If they exist, their values will be used for row matching. Although this works, I personally prefer to be explicit about what I want to do to avoid some magic results. Take such case for example. We have two DataFrames with different information for the same players, and we called merge on them. Pandas finds that the common column is playerID and joins the tables on the playerID. Perfect. Now imagine that a few days later you or someone who you work with adds position column to the second DataFrame, specifying full position name. Now, when you call merge, pandas will see that there are in fact two common columns in the DataFrame, and to join the rows, their values will have to be the same. And even if the playerID matches, where there are no matching values for position, the output DataFrame will be empty. How can we avoid such errors? We have to tell pandas which columns to use. This can be done using on keyword argument. Using it, we can explicitly specify the column, and everything will work just fine. In case the names of the columns differ, let's say we will have playerID and plrID, we can pass left_on along with right_on, and this will also work. If we have playerID as an index, not as a column, we can pass left_index=True. This, of course, has the right_index counterpart, so that gives you a possibility to flexibly come to your join columns. There are some more arguments that you can pass to the merge function, but they are not so critical, as some I will omit, and some I will introduce as we go through the demo.

Demo 1

It's high time we do some joins using pandas. We will explore the merge function in depth, but also take a look at some complementary utilities. Without a surprise, we start with importing all the necessary modules and then loading DataFrames that we saved previously. Just to remind, master table contains full information about some specific player, while the scoring table contains seasonal information about players' records. First thing I would like to do here is to enrich scoring table with player's name and surname from the master table. If we want to visualize data from the scoring DataFrame in the future, then it will be handy to have player full name, not just ID. In master we have playerID, which is the index, while in the scoring we have playerID as a normal column. We need to join all the index and the column, and as you might remember, pandas lets us specify left_index and right_on. So our function arguments are left DataFrame, right DataFrame, and information how to merge them. And that's it. As you can see, each of the right DataFrame records contains now all the information from the left table. This was easy, and that is exactly what I wanted. But let's stop and think for a moment. Where is the resulting DataFrame index coming from? In the left table, we had playerID as an index, so it's definitely not that one. It is the same as the index from the right table, but maybe pandas has just resent the whole index. To check that, the scoring table would need to have its index different from the numbers starting from 0, as this is the pandas default one, and we cannot tell if the one in the merge table is the original one or was reset. But modifying our index is easy. If you check the index property of the scoring DataFrame, you will see that this is simply a range of numbers starting from 0. What's very convenient is that we can simply add a number to this range, and we will get a shifted RangeIndex as a result. So let's make this an index of the scoring table and do the merge once again. If the index of the resulting table will start with 3, then it will mean that it is the original one. If it starts with 0, then it is a new default one. It starts with 3, so our index was taken from the right table. Index from the left table that we specified with left_index=True was just dropped. We have a playerID column, but this one comes from the right table. The thing here is if we join index on columns, the index that was used for join will be dropped, and the one from the DataFrame in which we used columns will be kept. What happens if we join index or index or columns on columns? Columns on columns we'll discuss later, and for index on index we can do a quick experiment just by setting playerID as the index in the scoring table and doing the join with the indexes. Quite intuitively, if we did index on index, our resulting index is the same. After explaining these nuances, let's just bring back scoring index to its previous state, and let's go back to our initial join. One thing that we noted is that we have not specified the type of the join, so it was entered by default. I said that I wanted to enrich our scoring table with the player's full name, so what I meant was more of a right join if scoring table is a right table. This would differ from the inner join if we had some playerID in the scoring table that would be missing in the master table. Do we have such records? How can we check that? One way to do this is to do both joins and compare number of rows. If some IDs are missing in the left table, the result of the inner join will be smaller. Let's check that. Apparently both results have the same shape, so there are no missing IDs in the master table, but to illustrate what could happen, let's drop five random rows from the master table using the sample method. Here, after getting these random rows, we have to get their indexes since this is what the drop method expects. Also, here you are seeing the drop method in action for rows. Now we can see the difference. There are more rows when doing the right join. But we can illustrate this difference even better, as the merge function has a very useful Boolean argument called indicator. It produces additional column in the resulting DataFrame, which indicates if the join field value was present in both tables, only in the left one, or only in the right one. Let's do the right merge again with this column. So now we have an additional column called _merge. First few rows have both there, but if we call value counts on the result, we can see that also right_only values are present there. Right_only marks the rows for which no data was found in the left table. We may take a look at them and confirm that indeed there are no values from the master table in this row. Obviously, there are no left_only values, as this was a right join. To get all the possible values of this indicator column, we can also drop some random rows from the right table and do the outer join. We just have to drop more in order to have all records for certain player missing. And here it is, outer join with DataFrames that do not entirely match. We can also take a look at those incomplete rows. It is a good exercise for row selection and also a good way to get to know some useful pandas feature, string operations. We want to select rows that in the _merge column contain either left_only or right_only value. One way to do this is to write a full expression that uses vertical bar pandas operator, which set an or condition on our row selection. This works, but to my taste it is a bit long and cumbersome to work with, especially taking into account that more concise methods exist. Apart from typical equals operators, pandas can perform string operations on columns that contain them. They are accessed via the str property of the series. There are many string methods available, usually known from Python standard library. They are all documented in the pandas docs, and the one that is of interest for us is endswith since we just want values that ends with only. This works perfectly, and you can see that now we have some rows that contain no data from the master table and some without data from the scoring table. I would say it's enough examples for the indicator argument and different join types, and we still have work to do with other tables. There is, however, one more interesting argument of the merge function that I would like you to see. It is called validate. All it does is to check in if your join respects some predefined condition. What type of condition? Think about what we have just done. We took a master table that has one record for each player and joined it with the scoring table that keeps multiple records for each player. This is called one-to-many join because single record from the left table is matched with multiple records from the right table. If we swap the tables, we would have many-to-one join, and if in both tables there would be only one role for each playerID, then it would be a one-to-one join. There is also many-to-many that you will do probably more rarely. So coming back to our validate argument, it can check, for example, if the many-to-one relation is respected. If it is not, it will throw an exception. This can be useful if you want to make your code more future proof and receive a quick feedback when the data supplied is not in the right shape. As I've said, in our case it is a one-to-many join, which we can encode as 1:m string. Given that value of the validate argument, our join will work, but this is defiantly not a one-to-one join, so the following code will throw an exception informing us that this is not a one-to-one merge. Now as you have seen almost all the merge function arguments, it's a good moment to save this merge table to a file. Before doing that, I just drop the birth-related info, as I just wanted to add player names to this scoring table. Do not produce too many files that just overwrite the existing scoring file.

Demo 2

As we are done with players, we can move on to the team's information. What we will do will be very similar with several differences. Just to recap, teams table contains the mapping between the tmID and its name with all the results information stored in the team_splits table. So this is just like for players, but here we have the mapping between ID and the names separated for each season, as the table originally contained also scoring information. This raises a question. Maybe the mapping between the tmID and the name is independent of the year, and if we draw the year information, we could deduplicate the entries, leaving us with single row for each team ID. Let's check that. There are different ways to do this, and my suggestion is to first select the ID and name columns and drop_duplicates. Any row that contains the same thing for both columns will be dropped. If mapping between the ID and the name is always the same, this should leave us with one row per team ID, but if we run value_counts, we see that there is one team with two values. This is the Chicago Blackhawks team, which apparently slightly changed its name between seasons. What would happen if we left the table after dropping duplicates and perform a join in such case? Let's just try it out on a very little DataFrame. For the left one, I will keep only these two records referring to the Chicago Blackhawks, and from the right one I will just select random two records referring to the Chicago Blackhawks also. So we end up with such two tiny DataFrames. What is your expectation for the outcome of this? Well let's see. We had two rows in both tables, and in the output we have four. This is an example of a many-to-many join. First row from the left table matched both rows of the right table, and then they were matched again by the second row of the left table. That's four results in the final table. So you probably agree that this does not make too much sense, and it's better to stick with our years' values that will guarantee uniqueness. We could in theory leave just one of these mappings between the ID and the name for Chicago Blackhawks and not care about such nuances, but let's do it well and leave the year information in the teams table. This will also demonstrate how to do joins on multiple join keys, as now we will have to match between the tables both on the tmID, as well as on the year. Doing this is very simple, as we just need to pass the list of columns to be used for matching, and we have a desired result. Know that a new default index was created. This will happen always if we match column on columns and not use any of the indexes. Be very careful now. This list of columns to be used has to be of equal length, and the order matters, so swapping here with tmID in one of these lists would break everything, as pandas would try to match year with tmID, just like here. It produces an error because it sees that we are trying to join on columns that have different types. Actually, in this case, column names that are used for join are the same in both tables, so it's best to just use on argument. This works perfectly, and we get the desired DataFrame with new default index, as this is the pandas behavior when joining columns on columns. We have the DataFrame, but let's spend a minute to learn another merge function argument that I would like to show you. It involves breaking some things, as I will do the join only on the tmID column. This produces some output, but as you may remember from the Chicago Blackhawks example, this also generates lots of useless rows. The interesting thing is what happens here to the year column. As it was present in both DataFrames and was not used as a join field, pandas left both columns in the resulting tables, marking them with X for the left table and Y for the right table. Please note how I'm selecting the columns now with a new argument, like. Like will queue the columns in which the value of the like argument is contained in the name of the column. Speaking of suffixes, it happens quite often that you will have duplicate column names ending up in the joined result. X and Y is not very informative, but luckily pandas lets us customize these suffixes. It is done by simply passing suffixes argument that must be a two-element sequence. So if we want to actually keep the DataFrame names, we could just pass them as the mentioned argument, and the result will look fine. These are all the arguments of the merge function that I wanted to show you, and frankly speaking, all the ones that you need. It's time to do the teams join in its simplest form. Make sure that the teams name is in the DataFrame, and save it to a file, overwriting the teams_splits file. As far as the joins are concerned, we are done with our tables, and feel free to proceed to the next module. But if you are hungry for some more examples, I invite you for a quick bonus session to present you some caveats and a shorthand method for joining DataFrames. As a starter, let's go back to the example from the presentation that showed why it is good to always explicitly tell pandas which columns to use, as I would like to stress the importance of it. Here you will also see how to quickly construct simple DataFrames for tests and experiments. So imagine we have two DataFrames with three rows, one that keeps goal and position information for players and the second one that gives the player's name. If we call pd. merge, passing just the tables as an argument, it will infer that playerID is the join field and will produce the desired output. But as we have seen on these slides, someone can add position column to the second DataFrame with the same name, but with incompatible values. This will result in pandas trying to match rows based both on the playerID and the position, which will result in empty rows. Therefore, always explicitly specify which columns you want to use, unless you have a very strong requirement for columns to be automatically selected. Based on such small artificial DataFrames, I would also like to show you that in some cases pandas provides a shorthand join method that can be used instead of merge. Imagine that we have three DataFrames, each with different information, and playerID is on the index. One contains goals, second contains positions, and the last one contains names. What if we would like to merge them into one? For merge, we would have to call it twice. First join two with each other, and then join the resulting DataFrame with the last one. But there is also a very handy DataFrame method called a join that can be used primarily when we are merging only index, not columns. If our DataFrames have the same index, we can call join method on one of them and then just pass arbitrary number of other DataFrames that share the same index, obtaining the desired result. This method cannot do anything more than the merge function, but it can be easier to use if you have many DataFrames with the right data and need to combine them. I think that it's enough knowledge about joins, and I'm sure that you will benefit from the things that you have learned in this module any time you need to combine multiple data sources. I am also sure that you will always be able to come up with a solution to such problems, even in the most complicated scenarios.

Moving Between Long and Wide Data Formats

Introduction

Now is the time to get familiar with wide and long data formats. As much is about pandas, this module is about good understanding of your dataset layout and its certain properties. This is something that I believe is a very important part of facilitating and speeding up your data knowledges, so stay focused. By understanding the dataset, I mean being aware what each column and row means, what information they convey, and what is the optimal layout of your data. This may sound a bit blurry at the moment, but don't worry. I will introduce all the concepts step by step and illustrate everything with examples. When we have a common understanding of what wide and long data formats mean, I will discuss key differences between them and give you some hints how to choose the right one. With this knowledge, I'm sure that you will be confident about what your desired format is, regardless of the dataset you face. Of course, we will conclude with a demo session where we will apply the concepts learned to our dataset, of course using pandas. Without further ado, let's go and get a sense of the basic dataset properties that you should be able to identify. Consider such a simple table with playerID and the number of goals they have scored, assuming this table is for some specific season. Each row contains number of goals that the player scored in this season. Using a statistical language, we can say that each row represents an observation of the number of goals for a player. We will also say that goals represent a measured variable, and in each row we have a value for a face variable. PlayerID is in this case an identifier variable. Here I introduce a bit of statistical language, but don't worry. It's just to help you understand the concept. Also, I believe that these terms are quite intuitive. Valuing the playerID column lets us identify specific measurement value, in this case the measurement being number of goals. That's the name, identifier variable. Having this very simple data set, let's combine two seasons' records in one table. This yields a new variable, year. So now the value of goals, which is our measured variable, does not depend exclusively on the playerID, but also on a specific season. This means that year is also an identifier variable, as it is required to identify specific goals value. Now single column playerID is not sufficient to determine the value of the goal variable. Sometimes a dataset is called a higher-dimensional dataset, and when we get to work with higher-dimensional datasets, an interesting property shows up. There is more than one way to represent it in the table. For instance, we can take this new variable, year, and use its values to create new columns. The table resulting from such operation looks like that. Now each value from the year column was taken and used to create new column, and each cell in our table gives the number of goals scored by player in the year given by the column name. If you take a look at these tables, you can see that the second one is wider, but shorter. In fact, it is not wider now, but simply imagine having data for 10 years. Then we would have more columns. Thus, that's where the wide and long data formats come from. Please be aware that there is no standardization on this naming. Some people say wide and long, some call the right one long and the left one tall, so don't rely on these names. And we will also see that there is another convention to refer to these differences. It's just important to understand the concept. I used wide and long to introduce the concept since that's what can be found in pandas' documentation. One very important thing to be aware of is that these tables carry equal amount of information, just presented differently. Now when we have introduced the concepts and you can see how the same data can be presented very differently, question arises. Which of these is more desired when analyzing the data, and why? You could also ask why do we even care? First thing is that the data in the wild exists in different formats, so you may receive your row data in any of these formats, depending on how it is collected and stored. Second thing is that you may be passing your tables to some external tools. Take, for instance, plotting libraries. These tools will sometimes expect some specific data formats and will simply not present desired results if, for example, you provide the data in wide format instead of long. So this is one reason to know that these formats exist and how to move between them. I would say it is very important for a data scientist to do this with ease, and we will practice this later with pandas. But between the stages of the data entering and leaving your environment, there is, of course, lots to be done by you. And upon receiving the data, apart from cleaning it, you will want to transform it into format that makes analysis simpler, faster, and is easier to reason about. Because of this, I would like to focus now on what makes a dataset like that.

Practical Tips

Luckily, there are some simple rules that can help you with that. If you stick to them, your datasets will be always in great shape. Let's take a look at our two tables and repeat our observations. In the first one, we have three variables as column names. In the other one, year variable values, in this case specific years, have been made column headers. For two years this produces just two columns, but imagine what would have happened if we had the whole NHL history. This would produce around 100 columns. So here is first simple and very important rule. Usually, you want to have only variable names as columns. You take a quick look at the table, and you see 2001. This is a year's variable value, so it should not be a column header. As simple as that. According to this rule, the left table is better for analysis. There is a simple intuitive reasoning behind this. In such format, each row represents an observation of values of one or more measured variables, and each observation can be uniquely identified using values of the identifier variables. In the second table, this is not the case, as to identify a value, you need to find a correct row and correct column. And just image what would happen if we stored more info there, like player's assists. So this format that you see in the first table is generally the preferred one. This rule in our example may lead you to a quick conclusion that the long data format is better suited for analysis, but let's take a look at an enhanced example in which in addition to goals we have assists in the table, a new measured variable. Let's say we are done with the year, and we know it should be a column, but having two measured variables, goals and assists, creates possibilities of keeping it in different format. Another valued layout for this table is as follows. We create a column named variable and indicate to which variable the measured value corresponds. So now we have twice the number of rows because we have two variables that we wanted to include in this column, and the data is in the longest format possible. This is, however, unadvised. When you have already identified an observation using identifier variables, in this case playerID and year, you should have all measured variables for it kept in a single row. So here is the rule. Not only will you want to have only variable names as column names, but you will also usually want to have all of your variable names as column headers. So whenever you spot something like this, a variable name being used as a value in the column, this is something to fix. Note that I'm using word usually a lot. The thing is there are no hard rules which tell that this dataset is right and this one is wrong. What I am doing is giving you some set of rules and concepts that will make your dataset easier to analyze and understand and which are generally considered to be in good practice. Let's agree that this is the data shape that is the right one to have when analyzing your dataset. Each column represents a variable, and variables and values are filling our table. But, as I mentioned, you will often need to move the data between the formats, as data comes in different shapes, and you may have to export it in different shapes. But without surprise, pandas provides simple ways to do that. There are two methods that we can invoke on our DataFrames, melt and pivot. Where are the names coming from? Melting is another term to refer to moving data from the wide format to the long one, while pivoting refers the inverse operation. You may know term pivoting from pivot tables in analytic software like Excel. While there are things in common, here it refers simply to moving data from long to wide format, and let's stop at that. I'm not going to present to you how these methods work now, as I think it is better to jump straight into the demo and learn on examples.

Demo 1

Let's learn how to move between long and wide data formats in pandas. First step is no surprise. We import what we need and load our DataFrames. It is good to remind ourselves how does the scoring table look like, as this is the first one that I would like to use for the examples. One side note here. When I started working with pandas with frames like that, I was wondering why are all the values floating-point numbers? Take a look at goals. You cannot half of a goal, yet these numbers are not stored as integers. The reason for that is that pandas uses NumPy, which apparently does not handle missing values for integers. So if our columns contain such data, pandas automatically converts the type of the column to floating-point numbers, which is a design choice. I suggest we accept it as it is since apart from looking strange, it does not affect our workflow. Coming back to our table after getting to know the concept of identifier and measured variables, we can point them out here. All the statistics, like goals, are definitely measured variables. PlayerID, year, and tmID are identifier variables. TmID is also here because the same player could play in different teams in a single season, so it is necessary to pinpoint single record. FirstName, lastName, and position are in fact just columns with additional information about the record, and they are always mapped one to one with playerID, so I will mark them as annotation columns since they just annotate specific observations. Generally speaking, this DataFrame is in pretty good shape, and normally there would be no need to change its format to wider or longer, but for learning purposes, let's try to change them back and forth by performing things similar to what I have shown on the presentation. I'd like to prepare a very limited version of this DataFrame to start trying out pandas' melt and pivot methods. In the first place, I only want to keep records for some arbitrary three_years. Let's make it 2001, 2002, and 2003. Then I draw up duplicates using playerID and the year columns. It is because for clarity I want to get rid of records for players that appeared in different teams in the same season. My next wish is to use only three players, so I draw three random labels of players, but restricting it to those that have values for all three seasons. To do this, I employ value_counts and pass the retrieved labels to the series method called isin, which returns true if the value matches anything from the list. For now, I leave only three columns, giving us the following DataFrame. Disregard the index, as these are just labels from the full-scoring DataFrame. Let's focus on the data format. Our DataFrame is now in the longest possible one. What you could want to do is to make it wider by spreading the year values to form columns. As much as it is a more convenient format for data analysis, you may want to make a plot of the number of goals scored by each player. And sometimes it is much faster to change your data format to match the plotting library expectations instead of playing with its settings. Before we try to do a plot, let's make a wider version of this table. To move from long to wide format, we need pandas' pivot method. It is important to understand how it works and what each argument does, so I will explain it slowly before and after showing the effect. One note here. We have two identifier variables here, playerID and year. You may be tempted to try to imagine how the things I present work for more of them, but stay calm. We will come to this. For the time being, I want you to focus on our example. The idea in our current transformation is that we want to keep one unique row identifier, which is a playerID, and we can signal this using index keyword argument. The second identifier variable that we have now, year, we want to decompose into its values and make them column headers. Intuitively, this has to be passed to the columns argument. Last thing is to tell pandas with which column values should it fill the new DataFrame, which is goals in our case. And the result is as follows. We get a DataFrame with playerID values as row index since we passed playerID as the index argument. Year column values have become column's names, which is in accordance with columns argument. Finally, the whole DataFrame has been filled with goals, as we passed it a values argument. You may be confused that sometimes index has no header above it, and sometimes it does. The reason is that sometimes it has a name, and sometimes it doesn't. By default, it doesn't, but the pivot method produces it. You can verify that by inspecting index and column name properties of our new DataFrame. If you set it to None, it will disappear, but let's bring it back. Now let's do the plotting experiment. I would like to have a single bar plot that would show how many goals each of the players scored in each year. Let's see what does the plotting method do out of the box for our initial long DataFrame. To be fair, we can set playerID as the index. As you probably agree, this does not make any sense. We have duplicated players on X axis and bars representing year and goals. This makes no sense, but we can see that it used all the index as the labels for the X axis, and each column has its own bar. Of course, we could tinker with Matplotlib, and we would get we want in the end, but productivity matters, and I prefer to quickly adjust my data to the expected format instead of trying to bypass it. Without surprise, if we take our wide DataFrame, this produces a nice plot of the goals scored in each year. Now it makes sense. So as you see, sometimes you really might have to change your data layout in order to achieve your goals quickly, so it's better to know your melting and pivoting methods, and expect next clip to give you more explanations on them.

Demo 2

Now when we have arrived at this wider table, why not try changing it back to the long format by using pandas' melt method. Before we do that, we have to reset the index, as the melt method doesn't really like to work with indexes. If we rest it, columns no longer contain exclusively years, so let's drop the misleading name. Having playerID as a column, we can start playing with melt. Apparently, this method has no mandatory arguments, so let's try it like that. Unfortunately, it's total nonsense, so you will need to learn a bit more about its signature. First argument we can pass is id_vars, which specifies which columns will be used as identifier variables. What's important, this does not include the year in our case, as we want it to become one. Here we specify that what columns are already identified in the records, and we indicate that we want to keep them. Let's try this out. Maybe a bit surprisingly it works. What happened here? What did pandas do? Let me explain it. In such scenario with melt, pandas takes all column's names that were not specified in id_vars arguments and put these values in the variable column. Then it also added a value column where it simply moves respective values. What is not desired are those variable and value names. I would prefer to have year and goals. It is easy to change that by passing var_name and value_name arguments, and the output looks nice. Another interesting thing that we can do here is to select a subset of columns to be used to fill the new variable column. In our case, we might want to have just 2001 and 2002's data and to drop 2003. This can be done simply by passing value_vars argument, by which we can explicitly specify subset of columns to use. Columns that are not included will be simply discarded when constructing the output DataFrame. This is all nice, and we have learned how to switch between different data formats. There are, however, some things that look not so straightforward. For example, pivot method produces something with index, but then if we want to melt it back, we have to reset the index as the melt method wants column instead of index. For me, it is a bit strange. I think we should explore limits of these methods and see if there are some more complicated cases in which they fail to produce the desired output. For instance, take such example. We have the same table as before, but we also take tmID into play, so this is another identifier variable. Here each player has only one team for one season, but could be more, so we can treat it as an identifier variable. Now here is the question. What if we would like each year's value to have its own column as before, but also keep both playerID and tmID as identifier variables? Previously, we specified playerID as the index argument. My only suggestion is to make index argument a list, add tmID there, and hope for the best. But without surprise, this fails. Pandas interprets this list as a list of index labels, which is completely not what I meant. For now we are doomed, and it looks like a big limitation of pandas. Is that what the biggest data analysis library for Python can do? Let's look for hints and try to do something similar. ______ tmID, let's add another measured variable, assists, and try to transform this DataFrame into wider format keeping not only goals, but also assists. What are your expectations? My first thought was that it should concatenate year values with variables so that we would get something like that. Apparently, it produces something like that, and it looks a bit mysterious. We still have year values, but they are duplicated, and above them we have goals and assists, so the columns are somehow divided into two parts, one corresponding to goals and one to assists. What is that? We can check that by inspecting the columns property of the DataFrame, and here we see something very new, object of type MultiIndex. This is a special structure that we will explore in the upcoming module, and in fact, we have to learn more about this to discover full capabilities of pandas when it comes to moving data between long and wide formats. The thing is that here, as well as in many different situations, pandas relies heavily on this MultiIndex structure, so it is not possible to take full control over your data without knowing how it works. In such case, I suggest we end our examples here and move on to the next module to see what this strange MultiIndex thing really is. I hope that so far you have grasped what wide and long data formats are about, learned to understand structure of your data, and to identify different types of variables.

Multi-level Indexes

Introduction

In the previous module, we hit a wall when pivoting our DataFrame, as we couldn't use two columns as id_vars argument in the pivot method. I have explained that this is because pandas prefers that we use multi-level indexes. In the upcoming videos, I will explain to you why such construct exists, how it exactly works, and what are the most important parts of the pandas API when it comes to dealing with MultiIndexes, as they are also called like that. Despite the fact that multi-level indexes do not exist in every data analyst's tool, there is a good reason for their presence in pandas. Some people, and it happened also to me, find them a bit confusing at the beginning, but I am sure that after this module you will be convinced that they have plenty of advantages, and you will start using them without any fear. To answer the why question, let's think about a regular index, which is a part of every DataFrame. It allows you to select your data in an optimized and convenient way by simply passing the index label. You may remember that when we had playerID in the master table we set it as the index, and this let us select rows play playerID, for example like that. Then you probably remember that we also have the scoring table, in which we didn't want to set the index, as we could have duplicate values. Whereas this is possible, this is not a good practice, and I wanted to skip it. The reason for these duplicates is that to find a single observation in the scoring table you have to also know the year, which is a second identifier variable. In our case, it is not a big deal since there are just a few years for each player. So after selecting the player, we can visually find each individual record. But it is easy to imagine some situation in which even after selecting by the first variable we still have lots of records. If we want to select a single record in pandas, even if we set index on playerID, we have to add a condition on year. This is not impossible to do, but first it is not the most convenient form, as we are chaining operations here, making it overly complex for a simple record selection, and second, it is not optimized for selection at all. Pandas developers have identified such problem long ago and tackled it with MultiIndexes. The idea of MultiIndexes is that several identifier variables can be used to create an index, and each of them creates an individual level in the MultiIndex. These multiple levels are what really differentiates MultiIndex from a regular one. As you can see, with each level the results can be narrowed down, ultimately nailing a single observation. This MultiIndex type is very naturals to pandas, and it is often expected and returned by some methods, people being perfect example of them, in which case an order to use multiple identifier variables they need to form levels of the MultiIndex. Don't worry. I'll show that in the demo. Of course, it is absolutely possible to have MultiIndex's columns index, as you might remember that we finished previous module with such case. Having our data organized with MultiIndex lets us select records that we need faster and with less code in a way that I believe is more intuitive. Now let me quickly introduce most basic functionalities that pandas has for working with MultiIndexes. One fundamental is used for creating a MultiIndex for the existing DataFrame. This is actually the method we know already, set_index. If we pass a list of column names to it, it will return our DataFrame with MultiIndex instead of a regular one. So here we have two columns, but if you have more identifier variables, you can of course pass them. Another situation in which MultiIndex is very useful is grouping, as we are able to group by its level values. So, for example, to get highest number of goals in a given season, assuming that we have set the indexes presented on the previous slide, we can just pass the level name or position to the groupby method, and this will do all the job. Working with groupby and MultiIndexes is really cool, and you can definitely expect some examples to be presented in the demo. One more thing that I love about MultiIndexes is how they allow you to slice your data, making many parts of your code more compact. Let's say we want our first player data for years between 1997 and 2000. If year was a column, this would be a very verbose condition. With MultiIndex, we can take advantage of index slicing. To do this, we first need an IndexSlice object. This is a convention in pandas, and once you create such ______, you can use it all over the place. This little idx thing lets you now do operations like that, where you slice arbitrary levels of the MultiIndex. If you want this year's range, but for all players, you can just select all of them. Personally, I love this feature as it saves me a lot of unnecessary typing and makes my code less error prone. One caveat that is important to mention is the that if you want to slice on the index values like I've just shown, it is necessary to have the values sorted. This is simply done by calling sort_index. Just do it once, and then you may be sure that the results of your slices are correct. In our case, the values are sorted any way because our input data was, but obviously it is not always the case. Now when you have an overview of what the MultiIndex is and how it works, I think it's a good moment to move to the demo, as this will better illustrate coolness of the MultiIndex, as well as will allow us to explore and discuss its properties on live object.

Demo

With our standard initial lines of code, we are ready to enter the world of MultiIndexes. For a starter, we will work with scoring DataFrame. This is how we keep it now. There is ordinary integer index, and all the information is kept in columns. Let's change it to use MultiIndex now by calling set_index method and passing our identifier variables just like I have shown on the slides. You can see that it produces this new special index, which contains both playerID and year. If we inspect the index attribute of our DataFrame, we can clearly see that it is of type MultiIndex and that it has an attribute called levels. We can check the length of this attribute to confirm that this is in fact 2. Actually, pandas provides a shorthand method to check how many levels a MultiIndex has. Digging deeper, we can see that the first element of levels is a regular index that contains IDs of the players. Unsurprisingly, next element is the year index. This shows that we are actually dealing with pandas' indexes, and MultiIndex is just another layer on top of it that enables us to work with multiple index instances for a single DataFrame. So we have the MultiIndex. Now let's take a look at what are some useful things we can do with it. Let's start with grouping, as I've said it's very convenient with the use of MultiIndex. To find a maximum number of goals for a specific year, we just have to specify a level. This is practically the same as for grouping with columns, so we are not winning a lot in this regard, but I just want to assure you that you are not losing this possibility when using MultiIndexes. Of course, now we don't see for which player is this result, but we can get index labels by using idxmax instead of max. Because we have two-level

MultiIndex, our labels are two-element tuples. If you would like to take a look now at full records, we can just use these labels for indexing with loc. This looks very convenient and should give you a hint that we can explicitly select records using tuples. So for instance, we can select records for Wayne Gretzky in 1982 in the following way. This is where you can see a real power of MultiIndex, as this makes things much, much simpler. This is already nice, but I've previously shown example with slicing, so let's do that. To prepare the ground, we have to first ______ IndexSlice object for convenience and also sort our index to make sure the tiers are ordered. In fact, for slicing, this is a must, as our index must be lexicographically sorted in order for slicing to work. How is this checked? MultiIndex has a method called is_lexsorted that returns a Boolean telling if the index labels are lexicographically sort or not. If we call it now, we see that it is, so we can proceed to slicing. We will then come back to see what happens when labels are unsorted. Let's now execute example from slides and find records for selected year range. This code indeed gets us records for selected tiers for all players. It is really great that we can do this so easily, and I encourage you to use this slicing technique with MultiIndexes, as this is very readable and fast. If that's not enough, we can also slice a playerID. One thing that is crucial to understand is that these slices work like Python slices. So we can, for instance, add a step and select every 2 years starting from 1997 to 2004. The difference is that in pandas the ______ index is inclusive. This is great, but what if you want to select years from 1997 to 2000 and then from 2004 to 2006? This is a bit more complex, but still very easy if you know the recipe. To achieve that, we need to learn a new MultiIndex method called get_locs. One of the things that get_locs does is to take our slice indexer shorthand wrapped in a tuple and return locations of the rows corresponding to our slices. Note that this array has only one dimension since even for MultiIndex integer locations are strictly positional. It means the levels are relevant only when using labels and loc. With iloc, each subsequent row is subsequent integer, disregarding index level. So now if we pass the result of the get_locs to iloc, we get the same result. It is a longer way to achieve what we have done before. The interesting part is that we can call get_locs for an arbitrary number of slices and then combine resulting arrays into one and pass it to iloc. This is not supported by pandas directly, as this is not an ultra-common use case, but we can write a quick helper function that given a list of slices will produce an array with index locations. We will need NumPy for that, as we are dealing with empty arrays directly. So our function takes a DataFrame with a MultiIndex as the first argument and then an iterable of slices. The only thing we have to do is to compute locations for every slice and then combine them using NumPy's concatenate function. We wrap the arrays to concatenate in a tuple since this function can take more arguments, but it's not relevant for us now. If we now apply this function to our DataFrame as two different slices, we will get a complete list of positions. Now we just call iloc again, and we get the result. As you can see, with a little custom code, we have performed a very advanced indexing operation for a MultiIndex. If it's not obvious what I did here, take your time to look again. This is not a trivial operation, but I believe it has a high learning value and can help you with many complex indexing problems. Let's come back to the topic of sorting the MultiIndex. I have told you that it has to be sorted in order for slicing to work, so let's see if it is indeed like this. Previously, we just called sort_index on our DataFrame, but in fact we can pass a level argument to it. So let's see what happens when you sort by year. It is a bit confusing, but what happened here is that the year's level took precedence over playerID, and records are now first sorted by year and then by playerID. If we call is_lexsorted, we get False. This is because pandas is checking the sorting from left to right, and in our case, the primary sorting column is the rightmost year. If we try to use the same slicing as before now, we will get an error, as pandas will not allow slicing in such a case. We can get the order back by sorting index without argument again, or alternatively, we can swap the index levels. This is something that is not very relevant in our case, but you will probably want to use it in your code from time to time. In order to do this, we have to call swaplevel's method. This will literally swap years with playerIDs, and now our index will be sorted back again.

Wide and Long Formats Revisited

You have already learned some really useful things that you can do with MultiIndex, but I want to remind you that our story with MultiIndex started in the previous module. We first saw MultiIndex automatically created by pandas when moving between wide and long data formats and realized that the knowledge of this structure is essential for doing those operations easily in pandas. So let's return to wide and long data formats and see how we can handle them knowing MultiIndexes already. To learn and practice this, I suggest to use team_splits table. To remind you, in this table we have year, tmID and name, and each column corresponds to a team statistic in a single month. This is not the right way to queue this data, as individual month values should be kept in a single column, not have a separate column dedicated for each of them. The thing is that the situation here is a bit more complicated. Column names are month names concatenated with the metric name, so we first have to find the way to separate them. It is a very good exercise, and now I will show you how to construct a MultiIndex separating these two things apart. First of all, I move everything that is not a metric value to the MultiIndex, so let's set one with year and team's name. TmID, in fact, is not giving us any more information than team name, and we won't be joining this table anymore, so I think we can safely drop it. We could have done this earlier, but I was not sure if we would need it or not. These operations leave us with a DataFrame that contains exclusively team's metrics as columns. Having this, it is easy to identify the patterns that the month is indicated by three first letters of the column names, and the rest is the metric name. So what I would do now is to extract these things to individual indexes. Let's stop for a moment here and explain it again. I have taken the existing columns index, and by performing map operations, I've obtained two indexes with the same length, one with information about the month and the second with information about the metrics. Now adding this to it, it's very easy to construct the hierarchical MultiIndexes that will incorporate the information from both of them. This is achieved with the use of MultiIndex function called from_arrays to which we pass our months in metrics. You may ask why it is called from_arrays and we are passing indexes? The answer is that index is such ubiquitous in pandas that there exists mechanisms that can deal with that implicitly. Now since the order is preserved and the MultiIndex has the length of the original one, we can simply replace the existing column index with the one newly created. And here it is. Now we have the information in a more structured way. The problem is that we are still in the wide format. I don't want the maps to correspond to columns, I want them as values in the row index, so we have to perform a melting operation that will move our data from this wide format to a longer one. When working with MultiIndexes, pandas provides a method that is dead simple to use. It is called stack and takes the level of the MultiIndex to use for melting. In this case, stacking means the same thing as melting, so it simply refers to converting the data to the longer format. Months are our first level of our column index, so they have index 0. And with this simple operation, month names are moved to our row index. This is really amazing. Just think how we started and how quickly we have reshaped our data with the use of MultiIndexes. One thing I still don't like is that we have years and months separated by team names and ID in our index. Fixing this is extremely easy thanks to swaplevel's method. We just specify which levels we want to swap, and it's done. Please remember that the levels are ordered from left to right. Apparently, I don't like this either. Now I've changed my mind, and I think that the best ordering for the index would be by team's name, year, and month. We can call swaplevels two times again, but it doesn't seem like the simplest solution. Luckily, pandas has another method for this called reorder_levels. We can pass the list with a new order based on the current one, and it will return the DataFrame with the desired index. Pretty nice, isn't it? The last thing missing here is the name for the month's level. We have names for teams and year level, so it would neater to have also one for months. It is as simple as accessing the name attribute of the specific level and setting it to what we want. Now our DataFrame looks really nice. To show you once again that we can do grouping very quickly with a MultiIndex, let's compute sums for all metrics for all teams aggregated by year. This is as simple as grouping by teams and year level and calling sum. We have moved from wide format to the long one, but what if you would like to do an inverse operation, maybe with multiple variables? Remember that pivot method could not help us in such case. Luckily, there exists an inverse operation to stack, which is called unstack. So this time, there is more consistency in naming. There can be many different shapes of the data that you may want to obtain, but for our example, let's assume that we want to take year's and month's information from rows to columns. We've unstacked this super easy as this method accepts arbitrary number of levels for unstacking. If we pass year and month, we'll get a new DataFrame with these levels moved from row to column index. We could also reorder levels here and fill the missing name for the index level, but I will leave it to you as an exercise. Last thing is to save our DataFrames for use in upcoming modules. I cannot stress enough how I like using pandas' MultiIndex since it allows us to keep our data in a very logical way and quickly perform all the common operations. It is honestly hard for me to believe that at the beginning I didn't like this construct and found it unintuitive. I hope that after watching this module you are convinced that MultiIndex is nothing you should fear, that it can simplify your workflow, and that it is a natural and idiomatic way of working with multiple identifier variables in pandas.

Handling Time Series

Introduction

In our data, we have some columns that are representing data information, but so far we have not taken closer look at them. I think it's a good time to do this since pandas has a lot to offer when it comes to handling such type of data. In this module, I will explain to you why we need special treatment of time series data, then present you some of the data types, methods, and functions that pandas provides for this domain, and in the end, we'll try out an experiment with this API in the demo. Time series data is the bread and butter of data analysis, as very often you have to work with data that contains temporal information. In our case, it may be a team's performance across seasons or players birthdates, but basically anything that can be indexed by points in time is a time series. One of the domains that usually deals a lot with time series data is finance, as you can probably easily imagine temporal ticks of currency's exchange rate or stock value. And since pandas was created in financial environment, it has excellent support for this type of data. But let's look at what the fuss is about and what problems arise when we lack proper time series data representation. Well let's take a look at our player's data ______ information related to the date of birth. Since these variables mark a point in time, we are definitely dealing with time series data. As you can see, each portion of our date is kept in different columns. This make it convenient, for example, if you would like to count how many players were born in the same month. We just group by year and month and get the desired result. We can also set these columns as levels of a MultiIndex and use them to select records by any of these date components. This seems convenient and sufficient representation of the data, but for time series data, we can often have more detailed questions. For example, how many people have been born in a single quarter? An aggregation by quarter is a very real-life scenario as, for example, many companies create quarterly financial reports. Now if I want to sync the use grouping, we would have to manually reindex the data by introducing quarter level, computing its value, and changing months' values. It is a real hassle that I would rather avoid, yet it is still doable. But imagine financial data, like euro to dollar exchange rate. This data may look like this with observation done in 15-second intervals. Then how should we keep this time information? We could again make levels of the MultiIndex out of the data components, but here we are at 15-seconds granularity, so this would be 6 levels, and imagine some more precise data. We could keep it as a string like it is presented now, but then any time you would want to extract some part of the date or group by something you need, you would have to apply string operations, which is very inconvenient, not to even mention the performance of such solution. We could also store it as an integer as a Unix Epoch, but this is really closely inconvenient if you want to do anything else than comparing the values. Intuitive grouping and aggregating by let's say hours is practically impossible in such case. The problem in these cases is that pandas is absolutely not aware that we are trying to represent a time series data. I think that it is clear now that there is a need for a new data type that could let us handle time series in an easy and performant way. This means being able to store it in readable format in one column, but be able to narrow operations to specific time you need, as well as to other subtract values. For example, take a date and add three days. And since the whole civilization is revolving around time, probably there are many use cases that someone would like to add three business days in a given country or to encode time zone information in the value for easy conversion. This makes it more and more complex, and dealing with such data just with the use of strings and numerical values would be very inconvenient. But I've told that in fact pandas has excellent support for it, and that's exactly where time-oriented parts of pandas come in to play. New data types that are time aware in the row has to perform many time-specific operations. For example, we mentioned aggregation by quarter. Now it's time to take a look at what exactly are these parts of pandas' API. This will be rather high level, but should give you a good overview, and you will definitely see more specific examples in the demo. Pandas exposes several data types that were crafted for datetime. First one that is a must know is a timestamp. Timestamp simply contains information about a single point in time and has multiple methods and attributes that simplify working with dates and times. It is aware of all the datetime components ranging from year to nanosecond, can perform conversions between time zones, recognize business days, leap years, etc. Practically, in most cases, this is equivalent to Python's datetime type from the datetime module. The simplest way to get a timestamp is to pass a string in one of the recognized formats to the timestamp constructor. Having this representation, we can then conveniently access certain properties, like year, or call utility methods, for instance, to get the name of the weekday. Another type is timedelta, which gives us relative notion of time. If we subtract one timestamp from another, it will produce a timedelta. So it cannot be used to determine a point in time alone, but for example, we can add it to some timestamp and get a new timestamp. Its maximum resolution is days, so we cannot have timedelta of 1 year, but rather need to resort to 365 days. Last data type that I believe is important to introduce now is the period. It represents, guess what, a period of some predefined resolution, for example, a month. We can call start_time and end_time on it to get the exact timestamps that bound this period. In that way, we can also test if some timestamp falls into the period. This was very quick, but we will go through all these data types in the demo. To complete this introduction, I would just like to mention that all of these types can be used to create an index, and pandas has separate types reserved for these indexes. Timestamps can produce DatetimeIndex; timedeltas, TimedeltaIndex; and periods, PeriodIndex. Now, let's go and check how these things work.

Timestamps

In this demo, we will mostly work with master and team_splits DataFrames. The first one will provide us a smooth start into the time-related parts of pandas, and with the second one, we will practice the concepts on a more complex dataset. To remind, master table has information about specific players indexed by their ID. Columns that are of interest for us now, namely birth year, birth month, and birthday, are so far separated and stored as numeric values. This is in fact very impractical, as we can transform them into datetime values. So the idea here is to take three of them and combine into one with time-aware data types. The basic function that is capable of converting other data types to our desired one is called to_datetime. It has a number of arguments that we will explore, but for our use case, we should take advantage of its capability to construct the columns from a dictionary. In the dictionary, we just have to specify mappings between time units and columns. This is a good moment to remind the assign method, which lets us create new column based on the function that produces its values. After creating it, the original columns are not necessary, so we can drop them. And now you can see that our DataFrame has got a new column. Let's take a closer look at it. There is a new dtype here called datetime64, which is a NumPy type designed for storing time information. If we select a single value from the column, its type is pandas' timestamp. This is the type that I've presented before, and it is the basis for any time series work in pandas. It has a lot of interesting methods and attributes, and to showcase some, let's try to print the timestamp in the custom format. Here we make use of two methods and two attributes to present the date in a human-readable format. In the same way that we extract year and day, we can obtain any time you need, ranging from year to nanosecond. But this is just a small part of what we can do with timestamps in Pandas. One of the other very useful things in today's global world is ability to easily move between time zones. By calling the tz_localize method, we can make our timestamps aware of the time zone. Then obtaining the time in the other time zone is just a matter of a single call to tz_convert. These things are really handy, as for example, you may want to take some timestamp data, produce somewhere else in the world, and have the hours displayed in your time zone. This happened to me many times, for example, when analyzing system logs from remote data center. So these things really come in handy sometimes. Note that we are working on individual timestamp values now. Can we apply such things to the whole column? Of course we can, and it is as easy as using dt attribute of the column. With it, we can call the same things, for example, fix the column to a specific time zone. This is also a good time to check how we can convert values between strings and timestamps, as this is an operation that is very often required. For example, your input data may contain datetime information in some custom format that has to be explicitly translated to pandas' timestamp. On the other hand, after you are done with your analysis, you may want to store the datetime information as a string in a format that suits your need the best. As we already have the timestamps, let's start with a letter case and see how we can convert it to string. It is as easy as accessing the strftime method, which takes a single argument, a string format that instructs how to dump your timestamps into a string. This accepts format symbols like capital Y for years, d for days, or a for the name of the day. It follows Python's datetime behavior, and documentation for this module should be a reference if you want to find all of the possible symbols. I have intentionally specified a ridiculous format to show you that indeed you can customize it as you want. Now let's use some sane format and see how we can convert from strings to timestamps. Actually, this format is so standard that pandas can infer it, and there is no need to specify it explicitly, but let's make it more cryptic as the real-world data comes in all strange and inconvenient formats, and you can never be sure what you will update for analysis. We can quickly mutate our strings with one of the pandas' string methods, replace. Now that we have two X's instead of hyphen, pandas has no idea what format this is, and it will communicate that by throwing an exception. In such case, we have to explicitly provide a format now. So it's worth noting that regardless of how the information is stored on the input, you can always parse it to a date. But as we know, data is often dirty, and it is important to take a look what will happen if some of the dates in our source series do not conform to the format. If we use the things just like before, we will end up with an error informing us that the date could not be parsed. If we want to parse the column any way, we have to pass the coerce argument. In such case, for any value that cannot be parsed, NaT will be returned, which stands for not a time. So is it just data type specific representation of missing values. Last option, which I rather try not to use, is to pass ignore argument. In that case, upon encountering a non-parsable column, padas will just return to original one without leaving any trace. As useful as it can be, it can also lead to errors in your data pipeline that are hard to trace, so use this carefully. Now before we explore other features of the datetime data type, I suggest we get rid of all the unnecessary information and to just leave what's really important. I would like to keep only the timestamps as the index and playerIDs as values. This is a very easy operation that we can perform with direct call to series constructor, specifying the values and what to use as the index. And the result is a series with a datetime index, an index composed of timestamps. We will see in a moment that this index type opens up new analytics possibilities for dates and times. Just one thing before. It is always good to have the values sorted when dealing with time-aware index. At the moment, we still have everything ordered by playerID, so let's fix that by simply calling sort_index. Going back a little, I have told you that MultiIndex solves some of the problems of data selection, but apparently, in case of timestamp, pandas can often smartly infer what you mean. Let's say that the requirement is to select only IDs for the players that were born in 1980. With MultiIndex, we would select just using appropriate level, and I would not expect datetime index to make things harder. Actually, it makes things simpler as we can just select by passing the year value. This looks a bit like magic, but the simple truth is that if pandas knows that it is dealing with time series data, 1980 is enough for a library to figure out that we are asking for a year. We can narrow down the result by passing a date in one of the recognized formats. So we can select specific months or even days. Basically, you just have to provide a string that pandas can pars into a timestamp. What is really nice, you can also use slicing with these. Note that in case of slicing both lower and upper limits are inclusive. Now that you have a basic understanding of how these timestamps work, we can move to the next video and check out other time-aware data types.

Other Data Types

Second data type that I mentioned in the presentation was timedelta. One way to get a timedelta is to find the difference between two timestamps just by using minus sign. As you see here, pandas uses timedelta to represent the fact that the second oldest player in our table was born 55 days after the first one. Having a timedelta, we can also add it to some timestamp and obtain a new timestamp. It is very important to understand this concept, as this allows for very flexible and powerful operations on dates. Actually, it does not only work for individual timestamps. We can subtract the timestamp from the whole index, and then we will obtain a timedelta index with all entries relative to the one we subtracted. Timedelta has also one interesting and useful attribute that timestamp does not have. It is called components and contains a tuple with all individual component's values. Accuracy of our timedelta is in days, but we can alter it simply by adding or subtracting some values. Let's add 5 hours and subtract 10 minutes. We now have more granular timedelta, and we can use it to see one more useful method of this class, namely round. If we want to get lower granularity, we can round to arbitrary number of time units, be it 10 days or 1 hour. You can see that timedeltas are really useful, and I'm sure you will use them many times when you will have to work with time series data. But let's go back to our birth dates. We still store them as a datetime index. This is really convenient, but not entirely precise. In each timestamp, we give information about hours, minutes, etc., but we actually know only on which day the player was born, so all these values of smaller time units are set to 0. If we set this data like this, someone in the future can use it and assume that since we have provided the exact timestamps, we knew that all the players were born on midnight, which is obviously very unlikely. To account for such situation, pandas has a period type, which lets us encode some specific time span instead of an exact time instant. In case of our index, it would be a better representation, and we can do the conversion by calling to_period on our series, passing also the desired frequency, which in our case is days. This produces a PeriodIndex, which is the type of period parameterized by the frequency, which is also stored as an index property. We can select a single element of the index in order to confirm that it is really of pandas' period type. Now, objects of the period type have two very important attributes, start_time and end_time. They return timestamps that specify the bounds of the period. This is useful, for example, for testing if a given timestamp falls into a period or node. And we can easily simulate both cases with the use of timedeltas so you can see how it works. Just to confirm, as you can see on the examples here, attributes and selection strategies for periods are practically the same as for timestamps, so we are not losing any of the convenience. After learning what exact data types we have at our disposal, it is a perfect time to move to the next video and learn some more advanced things that time-aware indexes let us perform.

Advanced Strategies

Time series data types have some special functionalities that greatly simplify working with them. One method that I would like you to really remember is resample. As the name implies, it is used for resampling, which simply means frequency conversion. So for example, we can upsample our series demand frequency. Interestingly, it will return something that is called resampler. One of the most powerful features of the resampler is aggregation. Think about it. We have data with an accuracy of one day, and now we want to upsample it to one month. We have to tell pandas how to aggregate all values from a single month into one record. For example, we can use count to tell how many players were born in a single month. So this is a bit like a groupby for timeseries data, as we are performing an aggregation here. Note that after applying the aggregation our index gets filled with all the possible periods, even if there are no records corresponding. This resampler is very flexible, as, for example, we can upsample to two-month periods. We could pass three months in order to obtain something close to quarterly aggregations, which I have mentioned a few times before, but, in fact, pandas understands quarters, so we can explicitly upsample to quarter. Isn't it fun, don't you think? Let's stop for a moment and take a closer look at the frequency. Q-DEC, what does it mean? As I told you, pandas was initially created for financial applications, and quarters are used extensively as a time reference in finance. DEC in this case stands obviously for December and indicates when does the fiscal year end. So in such case, first quarter start in January, which is the most common case. You can play with those quarters, but it is rather uncommon to have them starting in different months, so I don't think that it is super relevant to explore this in detail now. Just know that you can similarly work with weeks where the date that defines the start of the week is adjustable. This is cool and very flexible, but should give you a hint how complex can data manipulations become. Because there are endless possibilities for custom time periods in the offsets, I'm not going to dive too much into these topics, as such cases are usually not very generic. Just to give you a hint where to look for such functionalities, there is a whole module called offsets, which is created to handle such cases. It has a lot of custom types, which I don't want to present all because it would quickly get overly complex, but as an example that can be actually very useful, this module has a BDay type, which stands for business day. We can take, for example, a point from our index and add business days to it. As you see, I'm adding 7 business days, and we are getting a day that is 11 days forward. I am using timestamps since these offsets do not work great with periods. We could explore this more and more and explain why timestamps are required here, but this is definitely out of scope of this course, so let's continue with more important things. Another useful method, though not as spectacular as resample, is shift. It lets you literally shift your index by arbitrary number of periods specified as an integer. So we can shift by one day, which will effectively add one day to each index row. Same can be done for month or any other recognized frequency, but be aware that the shifting frequency must be always same as current index frequency. We need to specify the frequency anyway because if we called shift just with the integer argument, it will simply move our values forward without touching the index. Now, before we close the module, let's take a quick look at our team_splits table where we keep month and years as the levels of MultiIndex. How could we convert these levels to a single one with a date type? MultiIndex does not have a method that can do it alone, and honestly, when we want to work with period index, the best option is to drop MultiIndex and go for a regular period index. The more time you spend with pandas, the more you will recognize when to use which type of the index. Basically, I know already that I will use team's names just for grouping, so I know up front that there will be practically no benefit of MultiIndex for this specific use case; thus, I decided to drop it for this demonstration. So let's reset the index and construct periods from existing year and month columns. Here I'm using the most flexible method to manipulate DataFrames, apply, which lets us do practically anything we want, and because the lack of performance optimizations. In apply, we also need to specify axis since we may apply our operations row-wise or column-wise. In our case, it is obviously row-wise. So for each row I take year and month and construct a period. This produces a series, which I can assign as a new column to our DataFrame, doing it in place of the month. We can drop the year now and set our periods column as the index. This makes this DataFrame ready for more advanced operation, grouping by team name and resampling. The interesting thing, and the reason why I'm actually use this DataFrame now, is that we can change grouping and resampling. In that way, we can, for example, compute sums for each value for each team aggregated by quarters. This is a quite advanced operation and something that would not be easily achieved with regular data types and grouping; thus, I believe it is worth a little effort to prepare the DataFrame for this. Just like sums for quarter, we can compute any metric we want, for example, means, just like with regular groupby aggregations. We will see that in fact we can get even more insights into our data by using windows, which I will cover in the next module. I think that after saving our DataFrames in case we want to use them in the future, it is a good time to finish this module, as it has already been a lot of information. I hope that now you have understanding of the support that pandas has for date and time data and that you will be able to work with such data without problems in any relatively common scenario. Just remember that because handling dates often requires handling special cases, lots of nonuniformity, different time zones, or even different calendars, this domain can get very complex. Pandas offers a lot of functionality, and its datetime API could be a topic for a separate course itself. So despite the fact that I could have digged deeper into some topics, I believe that this was the right amount of knowledge for a starter. Of course, you may need to use parts that were not presented here, and if you ever find yourself in such situation, I strongly suggest you refer to pandas' documentation, especially looking for chapters related to date and time handling. I don't want to point you to an exact link now as things change, and this information could get quickly outdated, but this is usually easily discoverable and can save a lot of headache with corner cases. Now, depending on how you're feeling, take a break or jump right into the next module.

Getting More Insights into Your Data with Windows

What Is a Window?

Module about time series showed us how we can perform more advanced analytics, taking advantage of the fact that pandas can be time aware. In this module, we will take a look at another tool that we can use to get more info about our data, namely, windows. I will explain to you the concept by showing how we can interpret them, show how different types of them work, and what are some best strategies for working with them. What do I mean by window? Imagine again a table with currency's exchange rate. In this table, each subsequent row represents increment by 15 seconds and keeps a value of the exchange rate. Now window is in fact just a subset of rows of such data series, or a DataFrame. So for example, here we have a window of size two because it's spanning two rows. In this case, it is nothing more than a selection of two rows. Nothing stops us from computing some statistics for these window values, for example mean, and storing it on the side. Nothing difficult here. You may be thinking that you are missing some concept because it's too easy, but there is no hidden theory here. We just take two rows and compute some aggregation. Treat this state is our starting point. Based on this initial situation, I would like to discuss two types of windows, rolling and expanding. There exists more, but these are the most important ones. Let's start with rolling. In the rolling window case, given that we start in the situation that was presented before, we are actually able to move the window row by row and produce new values using the same function. If we proceed to the end, we will be able to get a new column with values for aggregation for each window. And that's what rolling windows are about. Our subset of rows is literally rolling through them, hence the name. Now that there is a missing value since if the window has a size of 2, at least two values are needed to compute the aggregated values; therefore, there is nothing of information of for the first value. This is a very powerful and widely used concept. For example, in financial analysis, most notably stock values, rolling mean is a very popular technical indicator. Later on in the demo, we will try out the rolling method on our data, but I hope that you get the concept. To replicate the presented operation in pandas, you just have to call rolling method with the desired size, in our case 2, and then call the aggregating method, in this case mean. This may resemble to you aggregate operations that we can perform on groups, and that's right, as here we just aggregate value for a single window instead of a single group. Let's take a look at another type, expanding windows. We will start from the beginning again. In case of rolling window, we are moving start and end of our window one row at a time. In case of an expanding window, the end of the window is also moving forward, but the beginning stays in the same place. So in this scenario, we are computing an accumulated value with each step, adding one more value to our window. The way to do this is to directly call expanding and then an aggregation method. Note that expanding does not take any argument as there is no size that we have to set. We just start with a window of size 1 and enlarge it until the series is finished. There exist some other types of windows supported by pandas or not, but I believe that these two are most widely used, and they will be of biggest value for you. So let's just stick to them for now and see them in action in the demo.

Demo

For these examples, I will use the scoring DataFrame and birthday series that we have prepared not long ago. After required intense module about time series, this demo should be smoother, as we are just going to explore two types of windows, and no surprises are waiting for you along the way. To start, I would like to take just one player's scoring data, let it be Wayne Gretzky's since he played in a lot of seasons, and therefore we will have several rows to have fun with. We end up with a DataFrame like that indexed by year values. They range from 1980 to 1998 without an array. Note that we have a duplicate entry for 1995 since he played in two teams in this year, but we don't need this information now, so let's sum these values by using groupby and leave only goals and assists from the original frame. Note that since we are grouping by the index values we have to pass the level argument, even is our index is not a MultiIndex. As a result, we get practically the same data with a difference only for 1995. Now we can compute a rolling mean for a window of size 3. So now a value for a year is in fact a mean of the original values for this in previous two years. What kind of information can it give to us? I told that rolling mean is widely used in financial analysis, but it is not related to this application. Its popularity comes from the fact that it lets us see a more general trend that would not be easily spotted by seeing only individual values as they can change significantly from one measurement to another without reflecting the mentioned strand. We can in fact very quickly do a basic plot. Here, it is clearly visible that the rolling mean line is in fact smoothed out version of the original values. We call the rolling method just with the desired size of the window, but it also takes some other arguments. One of them is min_periods, and we can use it to let pandas compute values for the first rows, even if it has less observation than the specified window size. If we set it to 2, the value will be computed for the second row with the use of just first two values. It's easy to guess that if we set min_periods to 1 there will be no missing values at all. Let's leave out the min_periods argument and check another one called center. What does it do? Let's call, again, rolling with the argument of 3. Note that I'm leaving out selection on goals to show you that the rolling calculations on the DataFrame work for all the columns. Here the result for the years 1980, 1981, and 1982 start in the row corresponding to the last window member, 1982. If we set the center to True, new aggregated values will be assigned to the label that is in the middle of the window. So the mean for the first three years will be kept in the second row. Be aware that now we have a missing value in the last row since the aggregated value that was previously there is now in the ______ row. What if the size of the window is even? Then the center row is assumed to be the one closer to the end of the window. This is not a super important functionality, but it's good to know that you can adjust this depending on the use case. Last interesting argument that rolling can take is called win_type, which is the type of the rolling window. What does this mean? Basically, now all the values that are inside the window are treated equally. So if a computer rolling sums with center set to True and size 3, in each row we will have a sum of its original value and the surrounding ones. Window type can change that. It is a kind of filter that changes the importance of the values depending on their location in the window. For example, there is a triangular window, which has a shortcut of triang. It's called triangular because it leaves the value of the center of the window at the original, but the closer to the edge of the window, the smaller the value's contribution. This relation is linear, therefore the triangular shape. So if we use this window on our DataFrame to compute sums, you can see that the sum for a window is in fact some of the central value and half of the sum of the edge values, which means that the contribution of these values to the final value has been diminished. There are other types than triangular, but I'm not going to discuss them for two reasons. First is that it can be already quite confusing how this works, and second, most of the time the special window types are using specialized applications, like signal processing, and I suspect that majority of the viewers do not come from the signal processing background. So I just wanted you to be aware that such mechanism exists, and if you're curious about other window types, pandas' rolling method documentation can lead you in the right direction. There was another aspect of the rolling operations we have just performed that I would like to discuss. In order to get windows of three-year size, we had to make sure that years in the index are increasing by one with each row and that there are now duplicates. Couldn't we integrate this with period's functionality if the index was a period index? In theory, it is possible to directly pass some frequencies, like 2 seconds, in order to have windows size fix some of them not on the row count. This is supported only for some of the frequencies, and in practice I prefer to fit my data into format in which I can use row offsets, not more abstract concepts. The problem here is that once you start to stack more and more complex features of pandas on top of each other, you will in the end spend more time debugging and looking for solutions to your problems than analyzing your data. Because of that, I suggest to first resample your data, which you actually did with the use of groupby, and then perform window calculations. This is just two lines of code, and I promise that it is on average much faster than trying to fit arguments other than integers into the rolling method. To take a more complex example, we may want to compute rolling sums by quarters. Let's take the birth_dates DataFrame, or more specifically, series, for this purpose. Birth_dates may not be the most interesting data, but the structure of this is really nice for the demonstration. So again, to do rolling sums by quarters, it is enough to first upsample the data to quarters and then apply the rolling window calculations, as now we are sure that each index value is just one subsequent quarter, so using rows is perfectly fine. We could do the same for weeks or whatever time period you want. There is nothing wrong in separating this flow into two stages, and there is no need to look for rolling windows based directly on frequencies. Remember how powerful resample method is, and use it when needed. That's all that I wanted to show you as far as the rolling windows are concerned. So let's take a look at expanding ones. Expanding windows are in fact simpler than the rolling ones because you do not specify the size, as they produce values from the start of the column. You can also pass min_period and center arguments, but nothing more than that. Expanding windows are useful when you would like to see how some value is changing with new observations coming in. For example, you may be planning your personal company's budget for a new year, and to do that, you would like to see how little the spending and income was changing in the previous year month by month, and this is a perfect example of an expanding window. In our case, we could want to see how the total goals and assists numbers were changing throughout Wayne Gretzky's career, and this is as simple as calling expanding and then calling sum on it. Actually, we can very quickly get a little plot so that you can have a better feel of what is happening here. It may also be interesting to call max. This plot gives us instant information that the most goal reached season for Wayne Gretzky happened relatively early in his career, but as far as assists are concerned, he was able to improve in following years. Now, such computations are very common, and for them pandas provides shortcuts. They are called cumsum and cummax, and the result is completely the same. But these are just for some very few specific cases. Expanding method is completely generic, and that means that you can pass any aggregation functions that you want. For example, I want to compute for each season what percentage of total career goals were scored by that season. To do this, I have to first compute the total goals in career, and then I can easily construct custom aggregation functions. The x argument here is in this case a series with values from the beginning to the current row. So I just sum all the values together so far, do a simple division, and as a result, I can see how Wayne Gretzky was getting closer to the total number of goals scored in his career. I think that for now it is enough, as there is not so much more to the expanding and rolling windows. I definitely encourage you to check pandas' documentation to see what predefined aggregation methods you can call on windows apart from sum, max, or min. Just last thing to be aware of. We did all the computations row-wise, but it is also possible to compute with windows on columns simply by passing axis arguments to the rolling or expanding method. Frankly, I have never used windows on columns, so I left this from the examples, but know the this is possible, and everything you've learned so far in this module applies there. I really hope that this module showed you how simple concept of windows can give you a lot of interesting insights into your data at the cost of just few lines of code. I am still often excited how effective data exploration tool it can be, and I am sure you'll benefit from it many times when analyzing your data.

Plotting and Presenting Your Data

Plotting Tools Introduction

We have finally arrived at the last module. In this one, we will focus on effective data presentation. You may know all the pandas methods and functions, but for your analysis to have any impact, you must be able to present your data and communicate results effectively, and this part of the course is to help. We'll first take a look at the plotting library that I chose for this module, and then I will present you basic concepts and code snippets that are essential to start creating your own plots. We'll close with the demo that will also include general hints about data visualization. As this is the last module of the course, some examples will also involve using the concepts from previous modules in slightly different scenarios in order to get our data into final shape for plotting and let you see some of the most important things learned in action again. So first question I'd like to answer right at the start is what tool we will use, as there are no plotting capabilities embedded directly into pandas. This means that there is always a need to call some third-party library. There exists several plotting libraries for Python. For example, when you call plot on the DataFrame, Matplotlib library is called underneath. This is the one that is probably the most comprehensive and most widely used; however, it is so powerful that it may be overwhelming sometimes, and for this and some other reasons, alternative libraries started to appear in the Python ecosystem. I would like to make plotting fun for you, not a frustration; therefore, I decided to introduce one library, which I have particularly enjoyed working with, namely, Bokeh. Why have I like it so much and why I think it's a good choice for you? There are several reasons for that. The API is simple. There is no abundance of methods that you have to browse in order to find one that will maybe do what you want. You work with simple but powerful primitives. It targets web browsers, which means that it can render interactive plots, which lets us use zoom tools, display additional information on mouse actions, or switch on and off selected data points, and it allows you to serve your plots as web applications, which is an amazing feature if you want to share your work and let people interact with it. It can also read pandas' DataFrames as a source of data for the plots, which can make plotting with pandas a breeze. With Bokeh, I have been amazed by how quickly I can draw appealing plots that can be also interactively used in a browser. Despite the lack of high-level methods that automagically create complex plots, Bokeh API was always very easy to grasp for me, and with simplest concepts I was able to create extremely cool plots. In the end, library choice is a matter of preference, but I assure you that Bokeh is fun, and since long time it is my preferred plotting library for most of the tasks. Also, the support for pandas' DataFrames, though basic, in most cases is what you really need. Also, Bokeh can export plots in different formats, but what is super important is that we can get interactive plots directly in the notebooks. Let me now guide you through the basics of the Bokeh library. In Bokeh, you create a figure by defining glyphs that should appear on it. There are a few glyphs available, the most frequently used being lines, circles, and bars. Nothing surprising here. How do you place such glyphs on a plot? Let's take a look at sample code. There are some standard imports and functions to render plots in the notebook, and you will see them the demo, but here just assume that every function in the object is in scope. You create a plot by calling the figure function. Having a figure, you may invoke methods that create glyphs. For example, to draw a line, you just need arrays that specify X and Y positions, and optionally, you can set the line width. The result is as follows. You can easily add more glyphs to the plot, for example, circles of various size. Note how easy it is to bind values of one array to the circle size. Of course, it needs some scaling, but for me, it's really fantastic how quickly we can do this. So this is basically the whole philosophy of Bokeh glyphs. We will see some others in the demo, but let's look now at how we can use our DataFrames in Bokeh plots, as this is surely of interest for us. So let's assume that we have such a DataFrame representing some completely made up employee salaries in some nonexistent company. We have employee's ID, number representing the grade in the company, years passed in the company, and their salary. This is not very interesting, but will illustrate how easily we can produce glyphs out of DataFrame values. Interoperability between Bokeh and pandas is achieved by leveraging Bokeh's column data source object, which we can create by passing a simple DataFrame. In that way, we transform DataFrame into an object that Bokeh can recognize. Now with this source, we can use column names to create glyphs. In this way, each row of our data will produce a circle with grade as the X-axis values, year of experience as Y-axis values, and size corresponding to salary. This produces a plot like this. Know that before generating it I divided the salary by some number to adjust the size, but this is not a complicated step. I must say this is one of the reasons I love Bokeh. From a DataFrame, I can generate a plot by simply telling which column should correspond to what. If this is not enough, I mentioned that another thing that is great about Bokeh is interactivity of the plots. The library has a very simple mechanism that lets you attach different interactive elements, that are called tools, to your plots. Some are default, like the zoom one, but you can also add some other, for example, very useful hover tool. Given the circles from the previous slide, we could also add it to that plot in order to display the ID of the employee when mouse arrives over the corresponding circle. The at sign is marking which column from the source to use. This is as simple as creating tools and calling method on the plot itself. And as we already have the source configured, we just have to specify that we want the ID here. As this is interactive, I will not show you this on the slides, as I prefer you to see it in action on the demo, to which we can actually move right now.

Demo 1

For starters, let's read the team_splits table with the period index set. Let's select only the portion of it, let's say for Anaheim Ducks, and use it for our plots. As you can see, this gives us single-team statistics for months. In my head, I have a plot that I would like to draw. On the X axis, I'd like to have a timeline for the whole history that we have for Anaheim Ducks, and on the Y axis count for wins and losses. Then, for each season, I'd like to have two lines, one representing wins and one representing losses in the season. I would like them both to present cumulative values for the season. Sounds complicated, and in fact it is not so straightforward, but I will show you that we can get up and running very quickly. This will of course require applying some of the concepts you already know before we get to draw our plot. First thing to observe is that the periods we have in the DataFrame diverge from reality. The year does not specify a year, but a season. So April 2006 is in fact April 2007, but just because the season started in 2006, we have 2006. Because of that, to have our index properly ordered, we have to add one year for all months in the first half of the year. Just before I do anything, let's compute sums for seasons so we can later validate if our changes give the data incorrect state. To add the year, we convert index values to timestamps in order to be able to add offsets and then simply map over them, adding year if the month happens to be before May. The choice of the exact month doesn't matter since we don't have any values for anything between April and September. As you see, after resorting, we have the index with correct ordering. Now the problem is that we have lost information about the seasons. How can we tell pandas that January of 2007 belongs to the same season as November 2006? Remember quarters where we were able to define the months being end of the year? We can do the same with years. If we upsample to annual frequency, we can also pass June as the year end to the resample method. In that way, we will obtain a resampler with this information encoded it in it making our year start in July and thus corresponding to seasons. To check that this is in fact correct, we can call sum on our resampler. If we go and check the original sums for seasons before adding year, the results are the same. So to sum up, we have both information about the seasons in correct ordering. Now for each season, we have to compute expanding sums of wins and losses. Unfortunately, we cannot call expanding directly on the resampler, but same as for windows, we can perform any computation we want on the resampler object. So for each x which correspond to a season, I first reset index. This is necessary for this to work since pandas otherwise refuses to perform expanding operation on datetime index. Then I just select interesting values and compute their expanding sums. As you see, the values are indeed expanding sums computed by seasons. The problem is that now our index is broken since we have MultiIndex with single timestamps and integers. This is, however, not a big deal since we have same number of rows as the original data, and the order is preserved, so we can simply replace the index. Last thing that is needed is to have missing values for the months that are without measurement; otherwise, on the plot we would less value of each season connected to the first value of the next season, which we don't want, and the missing values are necessary to fix that. If we want to get a uniform frequency, we can use asfreq method. It is a bit like resample, but it is not intended for performing aggregations and computations, simply just for making the index conform to the specified frequency with no values missing. Since we have first day of each month in our index, the frequency to which we want to align is MonthBegin. In that way, we get ticks for each month, and values for index entries that were not present before are set to missing. I am aware that this part may be quite dense, but these are probably the most advanced things you'll ever do with time series. And if you look at how little code we wrote, this is quite amazing. And believe me that with time and having all the examples from this course at your disposal, you'll develop good intuitions as far as all these advanced operations are concerned. Now, let's reset index to have everything in columns. With DataFrame in such shape, we can move to plotting. Bokeh requires a very quick setup for convenience using notebook. We need three functions, figure to create plots, output_notebook that we call instantly and only once in order to have our plots displayed in the notebook and also show which will be responsible for triggering plots rendering. We also import ColumnDataSource so that we can use our DataFrames. With this, it's times to plot, and it's really easy. We first create a source from our DataFrame. Next, we create a figure and inform Bokeh that we have temporal data by specifying datetime type for the X axis. After that, we draw two lines, green for wins and red for losses. Now that we only have to pass column names. And the last thing is to call show, which gives us our plot. The result is as expected, two lines for each season showing cumulative sums of wins and losses. Just a brief look at it, and we can see that the season 2006/2007 was definitely better for Anaheim Ducks than 2011/2012. This plot is definitely missing labels, legends, etc., but first, let's focus on the bar that appears on the right. This is default for Bokeh, and most of the things there are related to interactivity. These are in fact tools that you can disable or enable. At the moment, only one, pan tool, is selected. It allows us to move the plot area around. We can, for example, enable wheel zoom tool and scroll back and forth, or we can exchange the pan tool for the box tool, which is very useful. So I can zoom out a lot, for example, and then select area of interest back with the box tool. Also know that there is a save button for exporting your plots to files. This interactivity part is really great, as knowing that you are able to dynamically select specific parts of the plot may be really helpful when deciding how to present your data. Let's try to improve this plot a little bit. First thing that I'm missing is the legend. Simple wins and losses will do, and it's just a matter of adding the legend attribute to lines. Nothing special. But Bokeh's interactivity shows up also here. If we set the legend. click_policy to hide, now we will be able to select which lines we want to see on the plot. This is really intuitive and quick, another point that makes me really like Bokeh library. Obviously, the title and labels are missing. Let's leave labels for later since here the axes are self-explaining, but it would be good to have the title. To set it, we could pass the title argument to the figure function, but we can also directly access title attribute of the figure and customize it as we like. So we can, for example, set font size and alignment, which is enough for now. Let's now move to another plot I would like to show you, which will showcase more Bokeh features.

Demo 2

Before we start a new plot, I'd like you to observe one thing. The one that we have just generated is two dimensional, and each data point represents two values, one corresponding to X axis and another to Y axis. One of the very important aspects of creating engaging and informative plots is to take advantage of the space you have and present more dimensions of your data. Circle plot that I've shown on the slides is a good example of that. Apart from X and Y values, the size of the circles carries information about the value of the third variable, yet the plot is still two dimensional. This is important to understand, and our next plot will present more dimensions of data on a single plot. It will be more complex, but also much more comprehensive. I will use Wayne Gretzky's data again to present his performance across seasons. I will attempt to draw a plot, which will present us how many games Wayne played in each season and how effective he was by counting points per game. I also want to differentiate by team. This is already four dimensions, but don't worry. It can be done, as you can see in this conceptual plot. We will be using colored vertical bars, so we will need to do some computations to position them correctly. First, let's just leave some columns that will be necessary for us, year, tmID, games played, and Pts, which to remind you is a sum of goals and assists. We can also quickly calculate points per game statistic for each row and create a new column with those values. I suggest taking closer look at the tmID column. It is of the categorical type and contains 37 categories. The fact is that Wayne Gretzky did not play in 37 teams, and these categories were left after selecting rows from the original scoring table. In order to not have these redundant values showing up on the plot, we can take advantage of the remove_unused_categories method. Note that for categorical columns, all the methods related to the categorical value's handling are accessed through cat attribute, which is the same conventions for string operations, for example. The teams will be on the Y axis. How can we know where to place our bar so they correspond to the right team? This is easy to solve because all categorical variables have integers through which all the possible values are mapped. We can give these integers as a column by using the code attribute. In that way, we know exactly how tmIDs will map to the numerical values on the Y axis, and that allows us to have full control over our bar's position. Also, one thing, because Bokeh plot will not recognize the period type which we have now in the year column, we have to convert it to timestamp. Next step is for computing some constant values, namely minimum and maximum values of the variables that we want to present. I want to represent the games played with the height of the vertical bar and points per game with color. In order to choose the right value for both, we have to put this in the context of all values of the given variable. That's the need for min and max. Let's leave the color for a moment and focus on the vertical bars. In order to draw them, we need to specify their bottom and top position. We will have teams on the Y axis, so for each team we'll have one tick; therefore, the maximum height should be close to 1. Let's compute top and bottom positions for the bars and set them as columns. To get the height, we divide values of games played for a row by the maximum one. The bottom point for now will be the team code and top point will intuitively be bottom point plus height. That is enough to create a first draft of our plot. This time, we also specify y_range in order to get team IDs instead of row integers as the Y axis labels. We then specify vbar glyph, which represents vertical bar. Note that there is a mandatory argument with, but let's set it to 1 now and see what happens. Okay, this plot is not completely wrong. The heights of the bars correspond to the games played, and the bars themselves are more or less at the level of the correct team, but otherwise, this is far from good. Let's fix things one by one. I definitely want to increase font size of the labels, as they are barely visible. This is very simple as we just need to access X axis and Y axis attribute and set the desired size for the major labels. I think 12 points is okay for this plot. And now we have the teams on the Y axis, but the last two are not chronologically ordered. So we have data for New York Rangers, which was Wayne Gretzky's last team, below the single data point for St. Louis Blues. I prefer to have this the other way. How can we know which is the correct order, and how do we rearrange those values? If we have the data sorted by year, we can fetch unique category values, and in that way, we will have them in order they appear. Then it's just a matter of passing these values to reorder_categories method. Of course, we have the recompute our team codes and bar positions now. At the same time, we can also fix the width of the bars. Such thin ones do not look good. What are the units in which we specify this with? The X axis has a datetime type, so this is in nanoseconds. As each bar is for one year, let's convert, let's say, 320 days to milliseconds so they are relatively thick, but there is some spacing left between them. If we regenerate our plot, things start to look better. Things are now correctly ordered, and bars are wide enough. What is really harming the experience here is the fact that that bars are not centered around team labels. The worst looking is the St. Louis Blues bar, which is significantly out of place. What I suggest doing is to take lower bars' positions and move them up so that for each one the middle of the bar is at the label level. This gives us bars that are nicely positioned with respect to the team labels. One slight correction is that maybe I would like them all a bit lower so there is more vertical spacing, but this can be quickly fixed by multiplying the height by some constant factor. This looks much better, and I think the information about the games played is now displayed in a nice way. Now, let's add colors to represent points scored per game in the given season. To add this functionality, we have to use a color_mapper that Bokeh provides. I am choosing the most straightforward linear one, and in the Bokeh documentation, you can check what other types you can use. To construct such mapper, we have to provide a color palette and indicate what are the boundary values. In our case, these are minimums and maximums of the pts_per_game column. Now, to use this mapper in our plot, we have to pass additional argument to the vbar, which is called color. We are passing a dictionary, indicating which column we want to represent using colors, and also pass our color_mapper as transformer attribute. This gives us a color plot with color varying depending on the points per game value. The problem is that there is no legend indicating which color corresponds to which value. We can fix it by creating a color_bar from our color_mapper and adding it let's say on the right of our plot. This shows now which color corresponds to each value. One of the last things I want to show you is the hover tool since we still have not checked that functionality in action. If we want to display information about point per game and games played when the mouse appears over a bar, we have to create a HoverTool specifying tooltips, which tell Bokeh which columns' values to display and how to name them. Then it is enough to call add_tools and pass our newly created tool to this method. If we now hover over any of the bars, the two variables will be displayed. This features really makes Bokeh stand out, and I use it whenever I prepare plots that will be displayed in the browser. So I think that's enough. Labels and complete legends are obviously still missing, but my goal here was to show you how to create an advanced plot with Bokeh that includes more data dimensions than just two. This already took some time, and therefore I leave the exercise of adding the annotations to you, as I believe this is a much simpler task than what we have just done. To sum up, this module should have familiarized yourself with the Bokeh library, show you how using simple plotting primitives you can fairly easily compose complex plots, and made you aware of the importance of choosing the right visual representations in order to fit more data dimensions on a two-dimensional plot.

Course Wrap-up

With plotting module completed, we have finally reached the end of the course. I really hope that after watching it you feel that data analysis with pandas has much less mysteries for you now. To recap, you should now have no problems with working with multiple DataFrames, optimizing the format of your data, representing higher-dimensional data sets by using MultiIndexes, and covering any trend in time series data, applying window operations, and performing advanced interactive visualizations of your data. I am convinced that you will rarely encounter a data analysis task in which you will feel limited by tools, and with the lessons learned, you will quickly translate your data analysis knowledge into working code. I wish you all the best luck with all your future data work, and let pandas serve you well. Thank you, and good bye.

Course author

Paweł Kordek

Paweł is a software engineer passionate about knowledge sharing. He's especially focused on processing and exploring data sets (be it big or small) and is always searching for emerging tools that...

Course info

Level	Intermediate
Rating	★★★★★ (28)
My rating	★★★★★
Duration	2h 30m
Released	12 Jul 2018

Share course