Pandas Fundamentals

by Paweł Kordek

Start Course

Bookmark Add to Channel Download Course

Course Overview

Course Overview

Hi everyone. My name is Pawel Kordek, and welcome to my course, Pandas Fundamentals. Currently I am a software engineer at TomTom where I develop geography cross search functionality for the online APIs. Data scientist has been called the sexiest job of the 21st century. Regardless of this statement being true, digitalized data is everywhere, and skills that contain these never-ending streams of information are sought like never before, so why not do some data science yourself? In this course we are going to learn Pandas, a fantastic data analysis library for Python, which along the art programming languages is the most complete and popular open source programming tool for doing data science. We'll start this course by introducing core Pandas concepts, and then we'll cover topics that form the foundations of data analysis. This includes data input and output, indexing and filtering your data, working with groups, and creating plats. By the end of this course you will know how to manipulate your data in basic forms with the use of Python. Before beginning the course you should have a basic understanding of Python programming language, but no prior experience with any data analysis library is assumed. I hope you will join me on this journey to learn this amazing tool with the Pandas Fundamentals course at Pluralsight.

Getting Started

Course Introduction

Hi. My name is Pawel, and welcome to the Pandas Fundamentals course. In this course we will dive into the world of data analysis with Pandas library for Python. Regardless of your background, this course will help you in gaining data analytics skills that are extremely valuable in today's job market. This course is an introductory one, and assumes no prior knowledge of Pandas or any other data analysis library. You can expect introduction of basic Pandas data types, its data input and output capabilities, operations on groups, filtering and plotting. Minimal experience with Python is recommended. So, if you have ever felt like you could benefit from programmatic data analysis skills I invite you to join me and learn Pandas. This will be well invested time.

Overview

Hi, it's Pawel again. I'm very happy that you decided to start a Pandas Fundamentals course. What's in this module? First and foremost I will tell you why it might be a great idea for you to learn Pandas. Secondly, I will introduce you to the dataset that we will use for all the demos in this course, and finally, you will see how the basic data formats in Pandas work and behave.

Why Use Pandas?

So why would you be even interested in Pandas? I think that the best way to convince you might be to tell you a story of how I started using Pandas. At one of my previous jobs I had a task of reading large Excel files, performing numerous transformations on what I had read, and outputting the data into CSV format. I realized I needed to automate this in a nice way. I was looking at different solutions for automating this process, trying some others alongside, but in the end Pandas happened to be the one that was a pleasure to work with. I used it successfully for this task, and not only has my work become much more pleasant, but it saved me a lot of time. I have since been challenged with other data problems, and most of the time Pandas is still the best way to go for me. This Excel to CSV automation is just one of many existing use cases for Pandas. You can start very simple, but may end up doing complex visualizations, applying different statistical maintenance to your dataset. Basically, it's professional data analysis, and Pandas will do, but since it's Python, when Pandas is ever not enough you can always take

advantage of other libraries from the enormous Python ecosystem without changing the technology. You may wonder what exactly makes Pandas such a great tool for working with data? First, it gives us an intuitive table-like data format that I will explain soon. It also has a large number of data transformation methods available just out of the box, and last, but not least, it offers powerful data visualization possibilities.

The Dataset and Basic Objects

To learn data analysis with Pandas we will need some data to play with. Even for learning purposes, I love to work with public datasets that represent something real. Luckily, there are lots of them available nowadays. For this course I decided to use the Tate gallery collection metadata. This dataset describes around 70, 000 artworks from the Tate Collection. It is distributed via GitHub in JSON and CSV formats. We will use it for all the demos, except in this module, since the examples will be very introductory, but be ready for some real-world data later on. The time has come to take a first look at Pandas basic objects, as well as a few tiny code snippets that will show you how we can start using Pandas in our program. Let's go and don't get scared. We will start with two basic lines of Python code. These slides tell Python interpreter that we need Numpy and Pandas libraries. We could really ask them in anyway, but pd and np are typically used shortcuts. Why do we even import Numpy? There are several routines in Numpy that simplify working in Pandas, and you will see that very soon. Pandas and Numpy add several new data types, and there are three of them that are very good to be familiar with, since the very beginning of your adventure with Pandas. First of all, Numpy array and ndarray. These are two sibling data types that Pandas uses heavily under the hood, and you will also use them explicitly sometimes, as we will see in a moment. Nd stands for n-dimensional, which means it can have arbitrary number of dimensions. Then there are two fundamentals Pandas types. First is Series that represents dataset with a single column. Second one is a table like DataFrame format that represents multi column data. Remember that because they are everywhere in Pandas. Let's now take a look at a few examples of code snippets that present how we can bring these basic data types to life. At this point, I just want you to get familiar with them, and we will delve more deeply in the demo. First, let's see how we can create a Numpy array with three random elements. There is no use of Pandas in this example, but again, Pandas likes Numpy, and it interacts with it in a natural way. To do that we import Numpy in an idiomatic way, and then we create an array. Questions may arise. Can't we do that with Python's standard library? We can, but generating an array with random numbers is so much simpler in Numpy that there is no good reason to not use it. Such arrays can be instantly used to create a Pandas series that will contain three random

numbers, and don't worry, we'll practice this in the demo. The last short piece of code shows how we can create a DataFrame. Here we take a Numpy array that has three rows and two columns, and as a result we get the DataFrame with the same dimensions. To conclude this introduction of the new data types, let's make a simplified visualization. If you take a real Numpy array we just have a structure similar to basic Python lists. Then, if we put this data in a series object we get row labels as a bonus, and for the DataFrame column labels are also there.

Demo

The time has come to start our first coding session with Pandas. We'll create some series, a DataFrame, and see what's happening there, so you can get a better sense of concepts that were introduced in the previous clips. We will start with the Numpy array. First, we have to import the library, and in the next step create Numpy_array with three random numbers, the thing we have already seen in the previous clip. We can check what we have just created by checking the variable explorer in this Spyder IDE. I will often refer to tis tool, since it is a very nice way to visualize objects that you create. In this case, we can see that we have just created an array of items with type float64. This type column refers to the type of the items inside array. To make sure that we are dealing with Numpy objects we can try typing type, and the name of our variable. These arrays elements can be accessed just as in the case of any regular Python array. Now, let's import Pandas, and tell Python to create series using our existing array. Now that Spyder understands that it is dealing with series objects let's take a look inside. Do you see any difference? We still have one column of data, but instead of numbered labels, like for a regular array, we see something that is called index. Let's see what happens if it tried to access the first element of our series. Wait, it's the same as for a Numpy array, so why introduce these kind of special labels if the result is the same? Here is the first hint on how awesome Pandas is. These labels can be anything. Integers that start with 0 are the default, but we could specify our own index explicitly by passing in argument to the series constructor. We can try this out by specifying some strings, for example. Look at that. We have a series with the same data, but labels that we have defined ourselves. This 0 above is irrelevant because series can't have more than 1 column. Now you may have started to suspect, and you are correct, that we can index now using our labels, but what's great, we can still use integer-based indexing. Please know that we do not have to use variable explorer all the time. We can just type variable name in the console, and we'll get a nice output. Now what are these labels? We can inspect series index just like any property in Python. What we see now is a string representation of a Pandas index object that contains an array of our labels, and a dtype of these labels, where dtype stands for the data type. Objects in

Pandas represents anything that is not a number. That's a simplification, but that's okay for our use cases. So, as we see, series is basically some array-like data that is labeled by its index. It's cool what we have, but I already said that Pandas has awesome table-like DataFrame format, so now I'll show you what do we do when we want to have more columns than just one. Let's start with creating a random array with two columns this time. If we now want to retrieve some specific element we have to pass two indexes, row and column nuance. Now, for example, I'm getting item from the first row and the second column. In the same way we have moved from one dimensional array to series, we will now move from two dimensional array to a DataFrame. We do this similarly, but instead of series we create DataFrame. We will use df as a shortcut for a DataFrame, which is a widely used convention. For series we could access elements just like for a Numpy array. Will this work for a DataFrame? Unfortunately not, as we are getting an error when you try to do this. This complicates things a little, but don't worry. We will discuss indexing in one of the upcoming modules, and you will understand an embrace the way in which Pandas lets use index DataFrames. Labels for columns are kept in the same format as for rows. Just instead of specifying index property, we need to specify columns property, so if we have two columns we can send this property to two element collection, and we will end up with fresh new labels. If we now execute such code, we'll get second column of our DataFrame, and since it is only one column it is no longer a DataFrame, it is a series. We will end this demo in this place. Take your time to digest the material, but don't worry, we will spend the rest of the course working with series and with DataFrames, so you will ultimately be in full control over your data. See you in the next module.

Exploring Pandas Data Input Capabilities

Overview

Hi. Welcome to the next module in which we'll focus on Exploring Pandas Data Input Capabilities. What can you expect? We will first see what data formats Pandas can simply take and load into a DataFrame in one go. Then we will explore our dataset by reading the CSV version, and checking how we can take control over the formatting on the fly. To conclude, we'll try to achieve a similar result with the richer JSON version of the Tate Collection dataset. I believe these two approaches

will make you more fluent with Pandas input API and expose you to two different real life scenarios.

Pandas-compatible Data Formats

The data we may want to analyze in our scripts can be stored in various data sources. What can Pandas deal with? The library, at the highest level, can handle three types of both input and output data. These are, text, I will come back to this in a moment, since we keep Tate data in the text format. Binary, this form has become most useful when you need to work with other software formats. This allows data interchanged between Pandas, and for example, Excel, SaaS, and Stata software. Our use case for some of the binary formats is when you need to optimize I/O performance or you will use the files only from within the Python script, which we will also take a look at. Relational database format. Pandas can read data from all major relational databases, so if you have some data stored there you can read it with SQL query no problem with that. So something of the biggest interest for us, popular text formats. First of them, CSV, Comma Separated Values. This is a very popular format that we will explore later in this module. It's good to know that the separator does not always have to be a comma. We can tell Pandas that, for example, our data is separated by tab or any other delimiter, and it will understand. JSON. Widely used on the web, hierarchical format in which we can nest objects, as opposed to flat formats like CSV. Apart from that, you can read data from HTML tables, which comes in very, very handy sometimes. This was a very brief tour around Pandas input capabilities, but don't forget that whenever some built-in method does not suit your use case out of the box you can write a little custom Python code that will load your data into Python objects, like lists, tuples, dictionaries, which Pandas supports. For the simplest examples recall Numpy examples from the previous module.

Tate Collection Metadata: Take One

We have some background, so now we can start to think how we combined our Tate dataset. First, we will try to read CSV data. In this file metadata corresponding to all Tate artworks is stored in one file. For the sake of better understanding, it will be good to take a quick look inside. Please remember that the whole Tate Collection metadata is bundled with the demos for this course. First row of this file describes columns names. Each subsequent row represents single artwork. We have some IDs, artist name, title of the piece, and more that we will see later. CSV is a first class formatting Pandas, and the API is super simple. To get the data read into DataFrame we

just call the read_csv function. I think it's a perfect time to jump into this module's first demo, in which we will load the artwork data CSV file, inspect the produced DataFrame, and then read it again, but trying to avoid reading redundant data. Let's dive into CSV file processing. As I explained, Pandas provides a very component interface for reading these type of files. Since there will be more code than in the previous demo I will use Editor with my scripts more often than the plain IPython console. In the Spyder IDE anytime I want to execute a line or selection from the editor in the console I just press F9. First basic thing to know is where our CSV file is located. If you have unpacked the demos for the course in a random m2 folder your path will be the same as mine. After importing Pandas and knowing where the CSV is we can just call read_csv, but at the beginning I will try to take a look at the data sample to decide what I need instead of reading the whole file into memory. That's why I also specify nrows argument that will cause only first five rows to be read. This will give us good overview of how the data looks like. Now we are able to see what columns do we have, and what kind of data they contain. The first thing we can notice is that there is an index implicitly generated by Pandas, but we can also see that there's already an id column in our data. It is a good idea to use that one instead of creating something new. To take specific column from the CSV file as an index we can provide an argument called index_col. Now we are reusing already existing id. Next step is to decide which columns we want to keep, and discard others. First column that would be good to have is the artist column. We can try leaving only the artist column to see how this works. To limit the columns that are being read we have to specify another argument, usecols. This has to be a list of column labels. For the moment, we want to keep only one column, so we just passed artist and id. We have to keep the id column that we want to use as an index, otherwise, we will get an error. As you can see, we now get a small DataFrame with one column. It's worth knowing that instead of labels we can use columns positions. It would be 0 and 2 in our case, and the result would be the same. Just keep in mind that we cannot mix numbers with labels, so for example, 0 and artist would not work. Okay, we have this one column DataFrame, but we need more columns to have some fun with our data. I suggest we create a special array, COLS_TO_USE, which will keep our code cleaner. These are column names that I have selected as useful for the rest of the course. Apart from id and artist, we also give it title, medium, year, acquisitionYear, height, width, and units. Now I also dropped the nrows argument, so we can read everything. This reads the DataFrame, but issues a warning. It is basically caused by the fact that in numeric columns like width we have some trash data. For example, strings, and Pandas does not know what to do. You don't have to worry about it now. We will deal with that in the next module when we will want to use these numeric columns. Let's take a quick look at this DataFrame, since it will be a base from which we will derive all our data analysis tasks. We have unique id, artist name, title of the artwork, medium, year of creation, year

of acquisition, width, height, and units of width and height. I am sure that this will be sufficient for starting our adventure with Pandas. So basically no time we are done with reading the CSV file. One last thing before we move on. We will want to use this data in the future, so how we can simply keep it somewhere on disk to be easily loaded after. If we only want to use it in the Python scripts the simplest way is to use pickle, which is a native Python format for serialization. If you are not familiar with the term serialization, in our case, it just means writing and reading Python objects to and from a disk. To save this DataFrame with pickle we can just use Pandas method, df. to_pickle. It is now saved under the specified path, which I have chosen to be the demos root directory.

Tate Collection Metadata: Take Two

Now we will work with Tate Collection metadata, but in JSON format. In JSON the dataset is divided into folders with files of which each one represents a single artwork, so every file is like a row in a CSV file, but it can contain nested objects and represents more information in a structured way. For example, instead of a single artist column we have a list of contributors. I have mentioned that Pandas provides a convenient way to interact with JSON files. However, we have each data point in a separate file, and Pandas does not provide any way to specify just folder, and then read everything recursively. In such cases, when there is no easy way to read the data directly, due to its complexity, we can easily make a workaround by reading the files in Python and preprocessing them into a format easily digestible by Pandas. This can be, for instance, a Python dictionary or a list of titles. For such occasions, when we want to construct a DataFrame directly from the objects we have in memory Pandas provides several methods starting with from, with suffix depending on what objects we have. You will see this in action in a moment. We are moving directly to the demo. In this one we will find JSON files in sub folders, preprocess them for use with Pandas, and try to match our DataFrame generated from the CSV file. As discussed, our JSON data is scattered among many files. They are nested in several subfolders in the ARTWORKS folder, and each file represents a single artwork. We will need to process each of them one by one, and since reading almost 70, 000 files it's time consuming we will read only the first 5 from each subfolder, since we already have the data we need saved in pickle format, and it won't change anything in terms of learning outcomes. Pandas has several methods starting with from, but the one I often find especially convenient is the from_records method. It takes a list of tuples in which each tuple represents one record in the DataFrame, so you need to make sure that the ordering tuples and the link is always the same. To make an example, let's create a toy DataFrame using this method. I create a list, which could look like a simplified menu of any coffee

shop. In the record I have a coffee type and a price. Two items will be enough to demonstrate what we are dealing with. If I pass this list to the from_records method I instantly get a DataFrame with two columns. There are no custom column labels, but we can easily fix this by passing columns argument. That was a simple example, and now it's time to move to our problem, and read the JSON files. The strategy I propose is as follows. We will release files one by one into JSON objects. From each we will pick necessary data, which we will then put in a tuple, and append to one global root list of records. When we finish looping over all files we will be able to materialize our DataFrame. First thing we have to do is to take a look at one of the JSON files and check what we will need to pick. For example, we can see that information about the artist in one string is stored under the key, all_artists. I have checked other keys that can be used to mimic what we had in CSV, and these are the ones. There are some nested objects in this JSON, but we don't need them. Now we need some logic to process a single JSON file and return relevant data as a tuple. Naturally, I created a function for this. It takes file_path and list of keys to extract as arguments. What happens inside? We first load the files content into a JSON object. Then we create an empty list that is representing our record. For every key from the past list we extract the data and append to the record list. After we have all the data we can convert a list to a tuple and return it. We can try a simple example by specifying a path to the first file in collection, and calling our function with this path. As it is visible here, the returned tuple contains all the necessary information. Now let's collect more data by iterating over multiple files, and generating DataFrame from the gathered records. I also decided to put this logic in a function. It takes only the case to extract as an argument. To start traversing directories we need to specify the root directory, and it's done as the first thing inside a function. Again, if you are using demos provided, and they're in the empty folder, you will have the same path as me. Next, we can write a code that will traverse directories. At the beginning we specify artwork sleeves that will keep all records tuples. Then Python's os. walk function takes care of taking us to the JSON files. The only thing we do with a single file is to call a previously defined function, and append the return tuple to our global list. Break instruction makes sure that we visit only the first file in each directory. After we are done with processing files we can do the most important thing, create the DataFrame. We have to call from_records method of the DataFrame class, passing our master list, specifying column names, and the column to be used as the index. The last line simply returns the DataFrame. Well let's make this function available in our session, and call it to see what is created. It takes a while to complete, so you can now realize why I decided to read only the first file from each folder. You can now see that our data generally matches these extracted from the CSV. We have column names and the index in place, so everything is correct. In that way, you should have learned and understood how Pandas DataFrames can be created from different input formats,

and I hope it will help you in the future when dealing with some special cases. Let me just remind, that the from_records method is especially useful in many cases. That's all for this module. See you in the next one where we'll actually analyze the data that we have set from the CSV file.

Indexing and Filtering

Overview

Hi. In this module we'll focus on some other essential components of a data scientist tool book, namely indexing and filtering. Being able to quickly translate your ideas into code and select the right data means more time spent on getting knowledge about your data instead of trying to figure out how to get the desired output for a specific operation. What can you expect from this module? First of all, we will see what are some typical questions that involve data selection and filtering. In the second, and the longest part, I will present you what are the best strategies for data selection, and more importantly, you will gain understanding of the methods that Pandas provides. We will also see that our input data is not always clean and that we often have to deal with some unexpected cases. In the last part we'll recap on what are the best ways to select data in Pandas and what you should do to avoid surprising results and errors. Let's now look at the task I've prepared, which will loosely guide us for this module. The task consists simply of three quantities that I would like to extract from out data. These are how many artists in total we have in collection, how many artworks painted by Francis Bacon are there in the collection, and what's the artwork with the biggest area?

The Basics

Let's now dive into Pandas data selection starting with simple examples. The most basic thing, which we have briefly seen in the first module, is column selection. To recap, if we want to select a single column from our DataFrame we just need to pass its name in square brackets. Here you can see our DataFrame containing all the data, but in a simplified visualization. Using square brackets on it will return series like this. Please remember that I will use df whenever I will want to refer to our complete DataFrame. Important thing to remember is the fact that the index of this series is exactly the same as one of the source DataFrame. Instead of one column name we can pass a list of column names, which will in turn return a DataFrame, since we have more than one column now. There is also one way to select a column in Pandas that you should be aware of, but

generally avoid using it. Given our artist column selection we can also write it like this, and that will work, but know that this can have unexpected results sometimes, and I would discourage you from using this convention in your code.

Demo 1

With this knowledge we can start our first demo in which we will see how many distinct artists we have in our dataset. We will of course start by importing Pandas and reading our DataFrame that we previously exported into pickle format. Since we are interested only in artists here let's just create a single series called artist. Now we have to see how many unique values there are inside. Luckily, Pandas exposes a function for series that is called unique, which we can call like this, passing a series name as an argument. It's simple as that. This returns an ndarray with names of all artists without duplicates. To get the count of distinct artists we can call Python's len function, just like for any regular array. Now we see that we have 3336 unique artists in total. That was easy wasn't it?

Filtering

Now it's time to take a look at data filtering in Pandas. We are selecting columns, but what if we want to select only some of the rows that match specific conditions? With help comes one of the very important features of Pandas. We can do a logical test on the series. This means that we can write things like the following. We compare a series, in this case artist column, with a specific value. This works the same way as for all Python objects, but we effectively compare all the values in one go. What's the result? Given our DataFrame the result would be such a series, which is a Boolean series with the same name as our source series, artist in this case. It contains true in the row in which the condition was fully filled and false otherwise.

Demo 2

Let's practice now and check how many artworks by Francis Bacon we have in the dataset. I am in the same Python session as before, so I have our DataFrame in memory. Now I will write our comparison expression. As a result, like I have promised, we get a Boolean series that tells us if the artist for a specific row is Francis Bacon or not. One way to get our desired information is to check how many truthy values we have. To help us Pandas exposes a very convenient method called value_counts, which we can call directly on this series, and we can instantly see that there are exactly 50 artworks by Francis Bacon in the collection. This return object is a series itself that

takes all the unique values from this source series and counts them. Using this approach, we could also do this with our Boolean expression. It is possible to call value_counts on the artist column of the DataFrame. Then we get a series level by artist names with number of times they occur in the column. So if we select the Bacon's label now we will get 50 too. Is one approach better than the other? Absolutely not, and it is good to be aware of the fact that sometimes different ways to achieve the same result exist.

Indexing Done the Right Way

At this point you would probably be ready to perform many operations without a need to resort to other Pandas functions and methods. At some point, however, your code could start to look a bit clumsy. You could be also getting some strange errors and warnings, and you would start to search for some consistent way to perform all the data selection tasks in Pandas. To save you such problems, I will now guide you through the path of the proper data selection. By proper I mean consistent, with expected results, the one that does not surprise you, and one that does not force you to guess what is going on, and why are you getting what you are getting. This whole part will revolve around two special attributes that DataFrames have, loc and iloc. These allow you to crunch your data in an easy and consistent way, loc by labels, and iloc by position. You may be confused at this point. What's the exact difference between labels and positions? So let's take a look at our DataFrame. Labels are what is kept inside Pandas index objects, so our literal labels are just row names that we have set as the index when we are reading the data from the file. In the same way, we have column labels, which are just column names and were inferred from the files header. Apart from this, each row or column label has its position in the index. These positions start from 0. This means that you can select your data using these positions, just like you index, for example, Python lists, and that's what iloc is for. I hope you are getting a sense of what you can expect. Let's see how things work for loc. Using loc requires us to put our desired labels in square brackets. In the first place, we will be specifying row indexer, and in the next place, column indexer. A simple example may look like this. Here we are telling Pandas we want a value from row with label 1035 and the column artist, so this one would return a single value, but loc is much more powerful. We can, for example, put our Boolean expression in the row indexer. It will infer labels from the true values returned by this expression, and effectively return DataFrame only with rows in which Francis Bacon is the artist. This colon in the column indexer says that we want all the columns. Now let's take a look at the iloc example. Its usage is the same as for loc, but we are not allowed to use labels, we have to provide positions instead. One example of iloc usage would be something like this. For the rows I am specifying a slice of 200 rows, and for the

columns I am passing a list of three positions. This would result in a 200x3 DataFrame. We can also use the wildcard column. In this case, we would get all the rows.

Demo 3

Now you should have a basic idea how these methods of indexing work, so let's try them out in a demo. In this coding session we will practice what you have just seen, look for the biggest artwork in the collection, and see that there are common problems that can appear in data analysis. Again, I am in the same IPython session as previously. Before we start looking for the biggest artwork, let's warm up and replace some examples of loc and iloc. For example, we can try selecting a value of the artist column for a row labeled 1035. It works like a charm, and we get Robert Blake as a result. It also happens that I know that the row with label 1035 is the first row in our collection, and artist is the first column, so we can reproduce this with iloc just by passing two 0s. We can also try getting all the columns for the first row. Pandas will then return a series with this rows data, and as a last warmup exercise, let's check slicing. For example, to get first two rows and first two columns. As you can see, loc and iloc work and they are a very accessible interface. Now we can get back to the task of finding the biggest artwork in the dataset. By biggest I mean the one with the biggest area. We do not have an area column, but we have width and height, so we can just multiply them. We can use regular automatic operators for a series, so we can just try multiplying height column by the width column. Unfortunately, we are getting an error. Why is that? If we check what are the types of height and width columns it is object in both cases. As I briefly mentioned this in previous modules, this is the most general datatype in Pandas used for text and basically anything that is not a number. Since the data in these columns is not recognized as a number, but as an object, Pandas is not able to perform multiplication, but we have numbers in these columns, and when we were reading the data from the file Pandas should have inferred the datatype and used some numeric datatype for these columns, but the problem is that our input data is dirty. To check what is wrong I suggest taking one of these columns and sorting it using Pandas built-in sort_values method. Let's also add head to the mix, since we just want to peek into the data. As you can see, we indeed have some strings instead of numbers. Unfortunately, this is very common in data analysis that the input data is not perfect, and it often requires cleaning. In our case, to be able to perform multiplication we need to convert the column to a numeric format, and translate these rubbish values into something that will tell Pandas that we are missing data. Since missing data is a very common case, Pandas handles it out of the box by using not a number value. As you can see, we already have some not a number values in our DataFrame. They originate from all the empty fields in our input data. Again, we want to convert

the column into some numeric format, changing all the string values into not a number. How can we do this? Pandas provides a very convenient method, to_numeric, which will attempt to convert the column to some numeric type, so let's try that. Again, we end up with an error. Pandas could not convert to a numeric value. Luckily, Pandas is smart, and we can tell that whenever it fails to convert a value to a number we want this value to be changed to not a number. This requires passing additional arguments, which is errors='coerce', which means literally, on error please coerce the value to not a number. Hooray. This one works, and we end up with a series of a numeric dtype, float64. Since we now know how to clean our data we would like to replace our old width column with a fresh new one. We have been selecting data from the DataFrame, but how can we write to it to replace the column? Nothing difficult, but we can use the loc attribute. To write into the width column we need to select it like usual and assign new value, and we have clean numeric values in our width column. Just to confirm that this works, dtype is float64. Now we can repeat this operation for the height column, and we should be ready to multiply. Let's see if it works now. It does. We don't have units here, but by using value_counts on the units column we can see that only millimeters are present. Having this data, it would be useful to add it as a new column to our DataFrame with the name area. Pandas makes it painfully simple. We can just call the assign method on our DataFrame, specifying new column name and a method to compute it. This returns a DataFrame with the new column, so if you want it to take effect on our DataFrame we need to assign it. We have dealt with a lot of problems, and we are very close to answering the question, what is the biggest artwork in our dataset? We just have to take our new column and call the built-in max method on it. So the max value is something around 13 square meters, which seems very large for an artwork. Let's investigate what is that. We can easily find the index of this row by using idxmax instead of max. This returns a row label, so we can quickly use loc to display information about this artwork. Table and Four Chairs, aluminum, plastic, it looks like it is some sculpture or installation, which would explain why it is so big. We have answered the principle question, and I think now it is a good time to finish this demo. It has been a lot, but I assure you that you now know a large part of all the things that a data analyst should know about Pandas indexing and data selection.

Dos and Don'ts

In this module we have explored the most important parts of Pandas indexing capabilities. It's a good moment to sum up what are the best strategies for indexing and filtering in Pandas, and what should be avoided. Whenever I work with Pandas I always still to one main rule. Use loc and iloc. These methods are the most reliable ones, and they will not give you unexpected results. It is

a little more typing, but in return you get their solidity. On the other hand, I must admit I do use shorthand methods sometimes, but in very few cases here or there, either when I want to retrieve a single column or a few of them from the DataFrame or inside loc when using expressions for indexing. These are perfectly okay, but otherwise, stick with our good friends, loc and iloc.

Operations on Groups

Overview

Hi. In this module you'll learn another amazing Pandas feature. It's grouping capabilities and API that allows us to perform operations on these groups. We will go through it step by step. First, I will explain the motivation for REST APIs by showing several problems that may rely on this type of API. Next, I will introduce methods that allow us to group Pandas DataFrames based on column values, show you how it is handled by the library, and how to start using groups in the most naïve form. After that you will learn how to leverage Pandas built-in methods to speed up your workflow with groups.

Types of Operations on Groups

In this clip we'll look at what are some typical data analysis problems that may require grouped operationals. In the following part you will learn how Pandas can deal with them. I will not give any dry definitions of what operations on groups are. You will quickly grasp it from the examples. To start with, take for instance acquisitionYear of each artwork. Well let's say we would like to know what was the first acquired piece of art in the whole collection. We could simply check this by calling min method on the column with acquisition year like this. Now I can change this question. What was the first acquired artwork for each artist? To answer this we would need to somehow split our data and call min for each artist's data separately. Such operation in which we group data by some value, in our case, artist, and return one value for each group is an aggregate operation, and this is a family of operations that is used very often in data analysis. You may also come across such kind of problem. We have some values in the medium column, but in some places the entry in the CSV file was empty, so we ended up with not a number values. If we have missing data we usually want to do something about it. One way to deal with this is to simply remove such records, like we did in the previous module, but then we are losing data, which is often previous. Another approach is to fill values using statistics. One of the simplest ways to fill

this missing value is to take the most common value that appears in the medium column for a given artist, and put it there. It makes sense if most of John Walker's artworks in the Tate Collections are screen prints on paper, then we can guess that the one missing the medium value is also a screen print on paper. In this case, we would expect the DataFrame to look like this afterwards. This is another type of operations on group. We take our DataFrame, split it into group, perform some computations separately on each of them, and then return a DataFrame with the same size just with some answers changed based on the computation applied. This type of processing is simply called transformation. The last example I would like to introduce is a group dropping task. Leaving alone artists, we also have here the title column. As you can see, in the DataFrame there are some titles that repeat, like these things. It would be interesting to take a look at the data, but only for which items for which a title repeats at least once. So if we group the data by title we would like to keep all the columns, but drop the rows for which the title is not duplicated. This kind of operation where we keep some groups of our data and discard others based on some condition applied for each group is called filtering. To sum up, we have just seen three use cases that make use of different types of operations on group; aggregation, transformation, and filtering. In the next clips you will learn how we can call them using Pandas.

Iterating Over Groups

I have presented some of the situations in which we will want to use Pandas grouping capabilities. Now you are probably wondering how we can actually bring it all to life, so here we go. If we want to organize our DataFrame into groups based on a single column value we simply have to call the groupby method on it, specifying which column we want to use. This returns an object of a new type, which is called DataFrame groupby. Yes, it looks weird, but that's the name. This object basically holds the information about the data stored in our DataFrame, as well as about the groups that we requested. All the aggregation, transformation, and filtering operations will be called directly on this type of object. This object has also an interesting property. We can iterate over it. It generates a name, which is always a value from the column path and a DataFrame, which is a subset of our complete data corresponding to the name value. When we loop over such groupby object we can treat this smaller DataFrame as the master one retrieving any kind of information about the specific group. It is time to see how this looks applied to a real world problem in the demo. In this one we will iterate over artist groups, find the first acquired piece for each artist, and fill some missing values. I suggest we create a smaller data offering to get a better idea of the things we do. I have chosen a small slice to use for some of the examples here. To make sure that we will be working on the full copy of the data, I also invoke the copy method.

We can see two artists repeat, and there is one with a single row, so you will be able to clearly see how grouping works. First, let's call the groupby method with artist column name as an argument. We can now confirm that the type of this object is a special Pandas class, DataFrame groupby. We cannot visualize it, as it is a Pandas internal abstract representation, but let's try iterating through it. As I explained, this will yield names in specific groups one by one. I will use break, so that we can inspect the first group. We can see in the DataFrame that indeed it just gives us the relevant part of the master DataFrame as a new smaller DataFrame. As I discussed previously, we may want to learn when the first piece for a given artist was acquired. It can be simply done by printing minimal acquisitionYear value for each group. See? How simple is that. I also mentioned other kind of operation, transformation. With our example we would want to fill the missing values in the medium column with the most frequent value for a given artist. This is a more complicated case, but we can cope with that. We can see that our preselected data is okay, and no medium is missing, but let's pretend it is. This is also an occasion to show you another cool feature of working with Pandas in the Spyder IDE. We can edit our DataFrame when we open it using Variable Explorer. To change some value to not a number we have to just right-click on the cell, convert to float, and type nan inside. I suggest adding one missing value for an artist with many entries, but also one for the artist for which we have only a single row. We will be filling missing values with some precomputed ones, but if they're only missing values for a specific group what will happen? We will handle this case in a second, but I also want to bring your attention to the fact that it is a very good practice to check if your code works for some corner cases, not only for perfectly preprocessed data. Next thing we have to do is derive the actual function that will fill the values using the DataFrame provided inside our loop. In fact, it has to take a series as an argument, as we will want to pass a specific column, medium in our case. To determine which is the most frequent value count we can call the known value_counts method. Here we handle the case of having nan value for all entries for an artist, so in such a case we have to keep the data unaltered. If there are only nan values we cannot infer anything, and the data has to stay the way it is. Then we can check what is the actual most frequent medium, fill missing values with it, and return a freshly created series. For this to take effect on our complete DataFrame we have to iterate over our artist groups after meeting for each DataFrame, and then compose a single DataFrame from these partial ones using concat function. This produces the expected DataFrame, and you can see that the value was filled where it was supposed to be filled, and for the data for which we only had nan nothing changed. It may look like we have done a good job here, but in fact, it is a very primitive way of working with groups in Pandas. From time to time you have to use loops, but you will soon see that Pandas has some special methods that enormously simplify working with groups.

Built-in Methods

We saw that we can analyze data on a per group basis using loop, but there is much better and more concise way of doing these kind of things. Let's start where we finished our last demo, and analyze the transformation case again. We have defined a function that fills our data, and then we have to apply it in every loop iteration, store the results somewhere, etcetera, etcetera. You will be happy to know that Pandas has a fantastic shortcut for this. Given our fill values function, we can just do the following to transform the DataFrame. We group by artist and select medium, since this is what we want to transform, and then we just call transform method on our groupby object, passing our custom function as an argument. It returns fill minimums, so we can simply assign it to our original DataFrame. That is so much simpler and so much fun. Let's take a look at our second case. Finding minimal values for group. We first group and select the acquisition here, since this is the column we want the minimum for. Then we can use agg method for the groupby object, which allows us to pass any function that takes series as an input, and returns one value. We can even write it in a more concise way by using min, since Pandas exposes some aggregate functions directly. Similarly, we have such methods for filtering. Remember that we wanted to look at the duplicated titles only? We can group by titles, create a custom condition function that returns true if the length of the series is greater than one, and finally, we just need to apply the filter method with our condition path. This looks easy, and it is easy, as we will run these examples in no time in the upcoming demo. In this demo we will answer our questions properly, use methods that Pandas cache for the groupby objects exactly in the way explained in the previous clip. This is a continuation of the previous demo. We will just see how things are done in a simpler way. We have finished with this piece of code, which used our function, fill_values, to fill holes in the medium column. Now I can show you that using Pandas methods we can entirely skip the custom code we have written to apply these functions group by group. Instead, we just call transform method, and everything is automagically done. The example above shows you what is the logic behind, and now you see how to quickly implement this operation. In the same way we can move to the aggregate function. As you can see on these slides, getting a minimum pair of group values for group is painfully easy. We can simply call agg method, passing the name of the function that we want to use. Here we could as well use Python's built-in min function, but when using Pandas it is always better to use Numpy functions, as you may also get some performance improvement for free, but in this case we have the Pandas min shortcut, which is the best way to go. Now it's time for the last operation, the one we have not yet done before. I'm talking about a filtering operation. We need it when we want to get rid of certain groups based on a specific criteria. In our case, the criterion for group to be kept for the title is that it appears at least two

times in the dataset. We define our condition as a lambda function, and then we pass it to the filter method. Looking at the produced DataFrame we can see that this works and the titles repeat. To sum up, in this module you have seen how you can take advantage of Pandas DataFrame groupby objects and use them to perform aggregate, transform, and filter operations. It is important to remember that whenever possible you should use the built-in method, but there can always happen a case when you will need to resort to iteration for groups, and it is good to be aware of that. Thank you, and see you in the next module in which we'll learn how to output our Pandas DataFrames into different file formats.

Outputting Data

Overview

Hi. In this module we'll focus on learning how to quickly output Pandas DataFrames to different types of files. You have already learned that there are lots of different methods that you can apply to your DataFrame, and you also know that you may instantly view your DataFrames in the Spyder IDE using Variable Explorer, but ultimately this is not always enough, as there are many situations in which you may want to export your data to some other format. We'll look at three different cases with three different formats involved. First, imagine that you want to report your work to a customer, manager, or anyone else, and put your data in some easily accessible format like Excel. You may also want to share your previous results across your company by putting them in a relational database or you may be using Pandas server-side and may want to serve your data to the user by a web API, so you'll probably need it in a JSON format. Luckily, Pandas provides easy solutions for all these use cases, and we'll cover them one by one in this module.

Demo

We'll move directly to the demo in which we'll work with an xls as the output format. In this module we will also use a smaller DataFrame for some examples. Let's start a new IPython session and load the dataset into memory. Without needless talk let's dump this shrunk dataset to an Excel file. To do this we just call to_excel method of the DataFrame, and specify path to our output file. Now let's see what we have produced. It looks very similar to what we have seen in the variable explorer, but here it's in Excel. I have used this method successfully many times, but it is also good to know that there are several tweaks we can do to control how the data is sent to

the file. First thing that is often desired, we want no index, as it is often of no interest to the reader, so this is the first thing we want to control, and as usual, we can control it by passing additional arguments. In this case, it is index=False. Now we have an excel with our index. Another thing that is often handy is to only include preselected columns. For example, in the final report one would like to have only artist, title, and year columns, so we can control this without changing our DataFrame, but just by providing a list of columns ticket it instantly produces an Excel file that looks like that. This is nice, and it is often sufficient to use only this functionality, but and you may ask, what if I want to put multiple DataFrames in different Excel sheets, but in the same file? What if I want to apply some colors? Apparently, Pandas allows to do this. First, let's consider a case when we want to save the smaller data frame in the first sheet as a preview, and the original one in the second sheet. To do this we have to do some more work, but it is still very easy. We need to first create an ExcelWrite object. Then we call to_excel on each of the DataFrames passing a writer instead of a file name, and finally, save. This all takes a moment, since we are dealing with lots of rows, but as you see, we get exactly what we wanted. The next thing that I would like to show you for the Excel files is coloring with conditional formatting. This is not part of the Pandas library, rather of the underlying xlsxwriter, but we can easily apply some custom rules to our data frames. Let's try it out on artist value_counts To apply custom formatting we have to access xlsxwriter objects by accessing a sheet attribute. With this we can style the sheet. If you don't understand what is happening exactly don't worry. Achieving what you need in this case obviously requires working with the xlsxwriter documentation. We have to specify range to which we want to apply our formatting, and also specify exact parameters. Here I have chosen 2_color_scale with some tweaked parameters to decrease the impact of the extreme values. Take your time to go through it. The point of this example is to just give you a general idea of how this works, and let you know that this is possible. If we open the workbook now we can see we have row counts for each artist with colors ranging from light yellow for the largest values to deep orange for the smallest ones.

SQL Demo

Next, let's take a look at interface that Pandas provides for relational databases. To keep the example simple I will use SQLite, which is using simply a single file as a database. Thus, there is no need to installing a database system on your local machine. To work with SQLite databases we have to import SQLite module and create a connection that will also create the SQLite file for us. This connection is used by Pandas to determine where to put our DataFrame. With this connection opened it is only needed to call to_sql method with the name of the table in which

our data should be stored, and pass the connection as the argument. I can use now any free SQLite browsing tool to check that the data indeed has been loaded into the database. If you are using some real database like Postgres or MySQL, then you need to create a real connection first using sqlalchemy, which is a widely used SQL toolkit for Python. I am not going to run it now, but in case you need to do this there is commented out code that shows how to do this for Postgres database.

JSON Demo

The last format I would like to show you is the very popular JSON format. Pandas allows us to save data frames of the JSON files, also allowing us to specify how exactly we want it presented. We simply type, to_json, and our DataFrame ends up as a JSON file. Each column is a separate top level key, and inside it it contains indexes as keys and corresponding values. This is the default, but your needs may vary, so you are free to change this format by specifying additional argument, orient. For example, there is one called table that will output a schema first, and then a list of record like objects. There are few other orients available, which you can choose depending on your needs.

Plotting

Overview

Hi. In this last module we'll focus on one of the most important parts of effective data analysis, namely, visualization. You may be producing great tables of results, but as they say, a picture is worth 1000 words, and nothing can beat a great plot. That's why we will close the Pandas Fundamentals course with this topic. We will only touch the top of the plotting iceberg here, but it should be enough for you to get a basic understanding of how you can plot your data with Pandas. We will start with a brief introduction to Pandas plotting mechanism by creating the simplest possible plot. In the next step we will learn how to customize it for the desired appearance, and how to save our results to files.

Introductions

Let's start with some basics. Pandas provides a very straightforward plotting API. We can simply call plot method on our DataFrame and the plot will appear. One should be, however, aware that this is just a wrapper around the Matplotlib library, which does all the hard work, so if you ever need to fine-tune your plots and Pandas plotting documentation is not enough it means that you have to search in the Matplotlibs docs, but don't worry, I will show you how to interact with Matplotlib in an easy way.

Simplest Plot

Let's get our hands dirty. I will begin as previously by reading the whole DataFrame into memory. I think the simplest thing we can do is to group our data by the acquisition year, and count how many artwork were acquired in the given year. Let's take a look. Now we simply have a series that is indexed by the year and has acquisitions counts as values. We can instantly get a plot by calling the method I have mentioned before. We have just created the simplest possible plot with Pandas, yet it is very useful. For example, anyone looking at it can instantly tell that there was a single year in which the majority of all the acquisitions happened. That was easy, but there are a few things that I don't like about it. We have this acquisitionYear name, which looks bad, as well as we have no label on the vertical access. I would also like to increase the horizontal access granularity, so that we have shorter intervals for ticks than 25 years. Other problem is that this peak makes smaller differences barely visible, so maybe we need different scaling on vertical access, and where is the title? Let's move on to the next clip and see how we can deal with these problems.

Leveraging Matplotlib

Since the underlying Matplotlib library is very powerful, we can tweak practically anything we want. I don't want to spend too much time explaining Matplotlib in details, but I would like to show you some core concepts with which you will be able to achieve a lot and understand well what is going on. The master object in Matplotlib is called a figure. Imagine this as the final product of your data presentation. It is a kind of container for concrete plots. It can contain one subplot, also a grid of several of them, for example, four. What I have just called a subplot in Matplotlib is called an Axes object. So first you define a figure, and then just put some Axes subplots inside. In this demo we will work with just one subplot, but understanding these simple concepts will allow you to get more control over your figures, and will save you some time in the future. Enough theory for now. Let's continue plotting. The whole explanation in the previous clip

was required to take advantage of all the power that Matplotlib gives. Let's start with importing the Matplotlib's pyplot module, which provides many utility methods and functions. I also import rcParams, which combined with the update method is a way to tweak some runtime parameters of Matplotlib. It is not mandatory, but I want to tweak two parameters, so that everything fits nicely on the plot, and so that the title of our plot will not overlap with the plot itself. One other thing before we get down to the business. You must have noticed that when we called plot before our figure automagically appeared. This was due to the support for Matplotlib, that is enabled by default in Spyder. While this is usually desired, I suggest we disable it now to make sure you understand what is happening. Instead of relying on the IDE we'll call Matplotlib API directly to display our figures. Just be aware that you have to restart Spyder for this change to take effect. Okay, let's create the master figure using pyplot function, and then add a subplot. One, one, one means that we'll have a grid of one row, one column, and this subplot is the first one. In our case, it is the only one, so it will take the whole space. Then if we want to plot our series on this already existing object we need to pass ax argument to the plot method specifying our subplot. You may guess that ax stands for axes. Nothing appears, since we have disabled IDE support, but we can just call the show method and the plot will appear. We have just created what we had before, but with three more lines of code. In exchange, we get a lot more control over our data, as we will see in a second.

Plot Customization and Exporting

First thing we can set are the names for the axis. Let's make them nice. I will specify Acquisition Year as the x label, and Artworks Acquired as the y label. To set the desired names I only have to call set_xlabel and set_ylabel methods. Now let's increase granularity of our ticks on the horizontal axis. We could explicitly define which years we want to appear as ticks, but we can just specify how many ticks we want to distribute. On our subplot we have to call locator_params method specifying quantity of the ticks, plus also specifying axis if we want it to take effect on the x-axis only. Let's adjust the value, so we get ticks every five years. Okay, they are very squeezed now. It would be good to rotate them. We could use Matplotlib's method again, but here we can also pass an argument to our plot function where the argument will specify the rotation of the x-axis ticks. Let's go for 45 degrees. Now it looks much nicer. We have improved many things, but I am still not satisfied with the plot. We have this huge peak which stands out so badly that the context is lost for other parts of the plot. For example, what was the trend in the last 50 years? It is possible to see that, but it requires very close inspection. This can be fixed by adding log scale on the axis. It simply requires passing another argument to our plot method,

logy=True. I think now it is much more informative, don't you think? And we can improve its verbosity by also passing grid=True argument, which will display the grid. In that way we can identify more easily how many artworks were acquired in the given year. The last things I don't like is first, the lack of the title, and the font sizes, which are too small, but we can quickly fix that. We have to create an object called font date, which will specify our fonts. I have created two of them, one for labels and one for the title. Labels will be dark red, and title will be dark blue. Then we need to select the title in the same way as we have specified axis labels, and also pass arguments that will apply our desired font settings. For x labels I also add some extra spacing. Now I can say that this is a fairly good looking plot, so maybe it's time to save it to a file, so that we can share it with someone. First, popular format we can use is png, which is a raster format. In order to save our plot as a png file we simply have to call savefig and pass finally with a png extension as the argument, and this is the resulting file. That's so simple. Saving our plot. png is often sufficient, but sometimes we want it to be simple to zoom in without losing quality or easily embed the graph on the web. In that case, we additionally have to pass format SVG to the savefig function. You can then use, for example, the web browser to see that indeed we have our plot available as vector graphics.

Course Summary

Same as with plots, be aware that there is much more about Pandas and the whole Python data science ecosystem than you have seen in this course, but I hope that during this course you have already started to feel that you have a powerful tool under your fingertips, and that you can start doing data notices and exploration with ease. You may be wondering where to go from now, and there are many paths possible. There's one thing I have learned that cannot be emphasized enough, start using Pandas right away, as with every new tool, it takes time to integrate it properly into your workflow, but I promise that it pays off as you get more and more control over your data with every single use. Even if you don't feel very confident with it yet, get your hands dirty. There are little situations in which your Pandas knowledge can be applied, even if you are not a data scientist, because from time to time any kind of job requires processing of data that finally lands in various reports and on diagrams, and believe me, Pandas can be a great help in such tasks. I also strongly advise you to visit Pandas website where you can get general information about the library, as well as access the documentation. This is the first place to consult whenever you have any problem and seek new solutions. Not only will we find API referenced there, but also I could look at numerous examples of how to use different parts of the API. As far as books are concerned, I have two suggestions. One is called Python for Data

Analysis, and it is written by Wes McKinney, the creator of Pandas, which I think speaks for itself. Second, is an option if you would like to also learn or refresh basic statistics knowledge while practicing Pandas. There is a free statistics textbook released by the fantastic OpenIntro initiative. The book itself has nothing to do with Python and Pandas, but you can follow along trying to solve selected exercises using Pandas. This is a great idea because you practice two things at once, Pandas and statistics, a complete data science learning experience. I believe that the sources I have just mentioned are more than enough for you for now to develop your Pandas skills. Whether you will decide to learn more Pandas or not, I hope you enjoyed the course, and that you will benefit from it in the future. Good luck.

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	Beginner
Rating	★★★★½ (113)
My rating	★★★★★
Duration	1h 20m
Released	10 Oct 2017

Share course