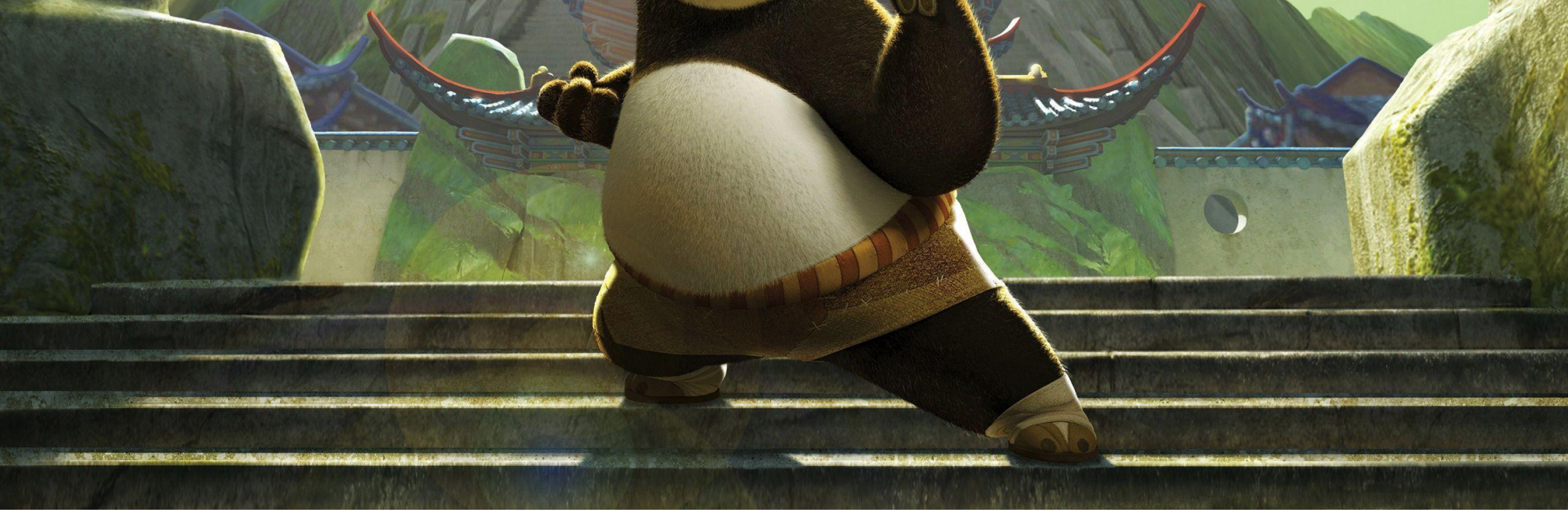
The Mastery of Pandas - I

Become a Master of one of the most used Python tools for Data Analysis

Jaime Zornoza Follow

Oct 15, 2019 · 12 min read ★



Most of you reading this post will have probably heard about **Pandas** before, and have probably used it in many projects, as **it is one of the most used Python Libraries for Data Science.**

This is the first post out of a series of 3 posts that have the intention of making you **comfortable with pandas and even teaching you some of its more advanced functions**. In this post we will see some theoretical insights into why Pandas is so useful and also cover its most basic use cases. In the following posts we will dig deeper into more complex endeavours.

Lastly, before we start, here you have some additional resources to skyrocket your Machine Learning career:

```
Awesome Machine Learning Resources:

- For learning resources go to How to Learn Machine Learning!
- For professional resources (jobs, events, skill tests) go to
AIgents.co — A career community for Data Scientists & Machine
Learning Engineers.
```

Lets get to it!

What is Pandas?

Pandas is a **software library** written for the Python programming language that is used mainly **for data manipulation and analysis**.

In a nutshell, Pandas is like excel for Python, with tables (which in pandas are called **DataFrames**), rows and columns (which in pandas are called **Series**), and many functionalities that make it an awesome library for processing and data inspection and manipulation. Over these post we will see many of these functionalities.

Pandas relies on Numpy and Matplotlib, two of the other main libraries that should be present in the toolkit of any Data Scientist, and as we will see there are many pandas functions which are naturally derived from them.

But first, lets see why we should use pandas.

Why and where should we use Pandas?

The answer to the *where* is pretty simple: If you are facing any project that involves data, you are going to want to use Pandas.

If you are creating a web page, or building a game, or writing an algorithm to manipulate a robotic arm, then you should be fine without it. However, if you are tackling any project where you have to visualise, analyse or do any kind of operations with data, do not hesitate: do it with Pandas.

The answer to *why* is also pretty straight forward: as I mentioned previously, pandas is built on top of other Python libraries, and part of it its implemented in C, giving it an extra speed boost. The following quote comes from Pandas documentation, which can be found here.

> *"**fast**, **flexible**, and expressive data structures designed to make working with "relational" or "labeled" data both **easy** and **intuitive**."*

These characteristics mean that we don't have to spend so much time waiting for operations on the data, and therefore can spend more time analysing it, understanding it and building models.

Lets see what it can do to better understand all this.

The Data: The famous Titanic

As the goal of this post is to explore the different capabilities of Pandas, we will not be using a complex data set, but rather a very simple one, that does however contain different kinds of features and some complications which we will have to solve through the the appropriate use of pandas.

You can find the data here: for this post we will only be using the training data of the Titanic Data set.

Pandas Yellow belt: Reading the data and first insights

Lets start our journey with the easiest tasks: reading the data and getting the first information out of it

The first thing we should do, once we have downloaded or collected some data is to **read such data into a pandas DataFrame**. This is one of the main Pandas objects, along with the Series, and like I mentioned before, it resembles a table with columns and rows. Before, as always, we should import the library.

```
import pandas as pd
```

```
df = pd.read_csv("train.csv")
```

The method ***read_csv,*** reads data from a csv file that is located in the same directory as the script or notebook that we are running the code from. **Pandas also has functions for reading from Excel sheets, HTML documents, or SQL databases** (although there are other tools that are better for reading from databases)

Its is kind of a **convention in pandas** to call these DataFrames by the name **df** (short for dataframe) for explaining purposes, so do not get confused if you keep seeing the variable name over and over again in tutorials. However, in a specific project, its probably best to give each dataframe a name that is intuitive and has some sort of relation with the data that is being treated.

We can check out the first *n* rows of our dataframe using the ***head* method**. There is also a ***tail*** method to look at the last *n*. By default if no *n* is given to these methods they return the first 5 or last 5 instances.

```
df.head()
# notice how no n is given, to get the first 10 items it would be:
# df.head(10)
```

Using the head method without a parameter returns the following block:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

Return of the head command on the Titanic Dataset

Sometimes however, we don't have a neat csv formatted document, like in the previous case. Consider the following example, in which we have **no names for the columns and a | separator instead of the usual comma**.

```
Peter|Peterson|182|81
James|Jameson|175|92
Martha|Mathews|168|55
Bruce|Wintersteen| 179|76
Charles|Xavier|170|68
Alexander|Aymes|183|78
Rebecca|Rebing|175|62
```

Example of an unconventionally stored csv document

Is there anything we can do? Of course, as I told you before Pandas is a very powerful tool, which can help us deal with these kind of issues. To solve this problem, we will use the same ***read_csv*** function, but we will give

it two additional parameters: the ***sep*** parameter, which lets us choose a specific separator, and the ***names*** parameter, which lets us choose the names of the columns for our data. Lets see the code.

```
df2 = pd.read_csv("data.csv", sep = "|", names = ["Name", "Surname",
"Height", "Weight"])
```

The following image shows the returns of the head method on the dataframe as we add more parameters to the *read_csv* function.

On the left we see what we would get if we use the method **without *sep* or *names***: just one column with no name and everything bundled up. In the **middle** we can see what happens when we use the *sep* parameter: we get neatly separated columns, but as there is no name for them in our .csv file, the first instance of data (the first person) gets treated as if it was the column names. **On the right** we can see what happens once we include both parameters: we get nicely separated columns with their corresponding name.

Using the “sep” and “names” parameters on the read_csv function

After successfully reading our data and creating our dataframe, we can start getting some information out of it with two simple methods:

- ***info:*** the *info* method returns the number of rows in the dataframe, the number of columns, the name of each column of the dataframe along with the number of non-null values of such column, and the data type of each column. There are no relevant parameters that can enhance this method, apart from increasing or decreasing the outputted information.

```
#Titanic DataFrame
df.info()
#Other Data Dataframe
df2.info()
```

Results of using the info method on both dataframes

As we can see here, on the Titanic Dataframe we have 891 rows (passengers) and 12 data columns. We can also see how for the Age and Cabin columns, we don't have 891 non-null values, which means that we will find some empty or troublesome instances.

- ***describe:*** the *describe* method returns some useful statistics about the numeric data in the dataframe, like the mean, standard deviation, maximum and minimum values, and some percentiles.

```
#Titanic DataFrame
df.describe()
```

Information returned by the describe method of the Titanic DataFrame

We can use an ***include* parameter** to get information about the non numerical attributes of our dataframe, however, often the statistics calculated for these attributes are not very useful. These attributes are stored as an object data type.

```
#Titanic DataFrame
include_list = ['object', 'float64', 'int64']
df.describe(include = include_list)
```


Result of the describe method on the Titanic Dataframe including all data types

That is it! We have seen the most basic methods about Pandas, and achieved out Yellow belt! Lets take on some more complex methods and continue learning to see what further insights we can get using this fantastic library!

Pandas Green belt: Accessing specific records

The next step after getting this global view of our data is **learning how to access specific records of our dataframe**. Like a python list, pandas dataframes can be sliced, using exactly the same notation as for the lists. So if we want to select the first 10 rows of our dataframe, we could do something like:

```
#Getting the first 10 rows of our dataframe
df[0:10]
```

First 10 rows of data

As we can see the **index of our dataframe (on the far left of the previous figure) is a normal integer**, which is what happens by default if we don't specify a different one when reading or creating the dataframe. We can access items with a specific index value using the ***.loc*** method of our dataframe, like so:

```
#Accesing the data with an index value of 1
df.loc[1]
```

Accessing the data with the index value of 1

We can also use the ***iloc*** method for **accessing data with a certain *index position*,** like in the following example where we select the same passenger but using its index position within the dataframe:

```
#Accessing the data with an index position of 1 (starting from 0)
df.iloc[1]
```

Accessing the data with an index position of 1

Whats the difference then? you might ask. Simple. Imagine we had a dataframe that instead of numerically ordered index values, had something else. I will teach you how to change the index of our dataframe in one of the following parts of this guide, but for the moment imagine our dataframe is the following, with passenger names as indexes and not integers.

Same dataframe but using the names of the passenger as indexes

In this case, to access the information of the second passenger (index position 1) **the values passed to the loc and iloc functions are completely different**. Lets start with iloc. Again, if we pass a 1 to this

function, we will get the information about the record of the dataframe with index position 1, which is the second element of the dataframe:

```
#Accessing the data with an index position of 1 (starting from 0)
df.iloc[1]
```


Accessing the element of the dataframe with index position 1

Now however, when using the loc method, we would have to pass the value of the index of the record that we want to access. In this case, as the values of the indexes are the names of the passengers, we would have to pass the name of the second passenger of our dataframe.

```
#Accessing the data with a certain index value
df.loc['Cumings, Mrs. John Bradley (Florence Briggs Thayer)']
```


Accessing the same data using loc

For accessing specific columns its as easy as:

```
#Accessing the 'Sex' column of our dataframe
df['Sex']
```

Accessing the sex column of our dataframe

Take into account that this is using our second dataframe (the one with the names as indexes). *loc, iloc* and column selection can be combined to access only a specific field, or a group of fields (columns) and records (rows).

```
#Accessing the first 5 records of the Sex column
df.iloc[0:5]['Sex']
```


Accesing the first 5 records of the Sex column

Using the *loc* command, we can select the specific rows (by index value) and columns (by name) that we want to access:

```
#Getting the columns Survived, Pclass and Sex from two records
df.loc[['Braund, Mr. Owen Harris','Cumings, Mrs. John Bradley
(Florence Briggs Thayer)'],['Survived', 'Pclass', 'Sex']]
```


Getting the columns Survived, Pclass and Sex from two records

Awesome! Now we know how to access the different elements of our dataframes! Lets carry on to the next belt and keep learning!

Pandas Purple Belt: Indexes, Columns and Drop

Okay, lets go back to our **initial dataframe**, (the one with integer numbers as indexes). **Here is the head**:

Return of the head command on the Titanic Dataset

We can **see the names of the columns of our dataframe** using the following block of code, which would print the name of each column of our dataframe.

```
#Seeing the names of the columns of our dataframe:
cols = df.columns
for item in cols:
    print(item)
```

We can eliminate certain columns of our datarame using the ***drop* method**, for which we have to declare the **axis as 1** to specify that it is a column we want to remove.

```
#Eliminating the columns Parch and SibSp
df = df.drop(['Parch', 'SibSp'], axis = 1)
```

Now, if we print the columns of our dataframe again, we will see that these two we just removed are no longer there:

We can also see the values of the indexes of our dataframe, using the following code:

```
#Seeing the values of the indexes of our dataframe:
df.index.values
```

In our case, this would return an array with integers from 0 to 890, as in our dataframe the indexes are just a sequence of numbers. Remember, however, how in the example in the previous belt we changed the indexes to be the

names of each passenger? This can be done using the ***setIndex*** method and **specifying which column we want to use as the index.**

```
#Setting the indexes of our dataframe to be the names of the
passengers:
df = df.set_index("Name")
```

Now, if we check the values of the indexes, or we will get the names of the passengers:

```
#Seeing the values of the indexes of our dataframe:
df.index.values
```


The first 10 new indexes of our dataframe

Awesome! Lets carry on to the orange belt and progress one step further towards our pandas mastery.

Orange belt: unique, nunique, and value_counts

Using the ***unique*** and ***nunique*** methods we can **see the distinct values** of certain columns of our dataframe (*unique*) or the number or **count of these distinct values** (*nunique*). Lets check out for example the different cabins that our passengers boarded. For this, first we have to select the column we want to inspect (Cabin in this case) and then use the *unique* method:

```
#Seeing the values different cabins that our passengers boarded:
df['Cabin'].unique()
```

If we do not want the values of the column (the different cabin names in this case), **but rather the number of different values** (the number of different cabins) we have to use the ***nunique*** method, which in this case is 147 different values (cabins):

```
#Seeing the number of different cabins
df['Cabin'].nunique()
```

Another very useful method is the ***value_counts*** **method**, which shows us the **different values of a certain column of our dataframe**, and also how many data points in our dataset have such value. In the following example we use the value_counts method to see the distribution of sex in our dataset.

```
#Seeing how many passengers are male and female
df['Sex'].value_counts()
```

As you can imagine, this method is **specially useful when we have discrete values in our dataframe column**, and not so much when these values are continuous. If we used the value_counts method on a column with a numerical value from 0 to 100000 for example, we wouldn't get much useful information.

That's it! We have obtained our Orange belt, and now are fully ready to take on harder data science tasks!

Conclusion

We have seen what Pandas is, and some of its most basic uses. In the following posts we will see more complex functionalities and dig deeper into the workings of this fantastic library!

To check it out **follow me on Medium**, and stay tuned!

Edit: **Check out part number two of The Mastery of Pandas here**.

For further resources on Machine Learning and Data Science check out the following repository: **How to Learn Machine Learning**! For career resources (jobs, events, skill tests) go to **AIgents.co — A career community for Data Scientists & Machine Learning Engineers**.

That is all, I hope you liked the post. Feel Free to follow me on Twitter at **@jaimezorno**. Also, you can take a look at my other posts on Data Science and Machine Learning **here**. Have a good read!

Data Science | Machine Learning | Artificial Intelligence | Analytics | Data