**MIT xPRO Data Engineering Certificate**

**Pandas**



The pandas *library* can handle data that comes in many formats. In this mini-lesson, you will further explore the input and output functionalities in pandas to handle files with different extensions.

**What Is Pandas?**

Pandas is a powerful *library* written for the Python programming language. Arguably, pandas is the preferred tool for data manipulation and analysis in the Python data science community.

As you have learned, it is possible to define data as a pandas *series* or, as you will see in the next video, as a pandas *dataframe* within the *library*.

However, you will often be asked to import or export external data in different formats by using the pandas IO tool.

**What Is the Pandas IO tool?**

The pandas IO tool is a collection of reading and writing *functions* that allow the users to import and export data saved in various formats to and from their program.

For example, suppose your manager asks you to analyze some data saved in HTML format. As a data scientist, you would then load the data in your Python program using the appropriate pandas *function* to read HTML files and perform your analysis.

**Reading Data**

The pandas *functions* for reading a file are all named using the pattern read\_<file\_extension>(), where <file\_extension> denotes the type of the file you are trying to read.

For example, as you have seen in the last video, the read\_json() *function* is used to import JSON files.

As pandas is a versatile program, it can handle files with different extensions. In the table below, the *functions* used to read the most common file types are summarized.

**Functions for Common File Types**

| [read\_json()](https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html#io-json-reader) | Read JSON files |
| --- | --- |
| [read\_csv()](https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html#io-read-csv-table) | Read .csv files |
| [read\_html()](https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html#io-read-html) | Read HTML files |
| [read\_excel()](https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html#io-excel-reader) | Read Excel files |
| [read\_sql()](https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html#io-sql) | Read SQL files |
| [pandas IO tool](https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html) | The pandas IO tool is a collection of reading and writing *functions* that allow the users to import and export data saved in various formats to and from their program. |

**Writing Data**

After reading, analyzing, and manipulating your data, you may want to export your data to a file for future use.

Depending on the file format you want to use, pandas offers a *series* of writing *functions* conveniently named with the pattern to\_<file\_extension>(), where <file\_extension> denotes the type of the file you are trying to save.

For example, as you have seen in the last video, the to\_json() *function* is used to export JSON files.

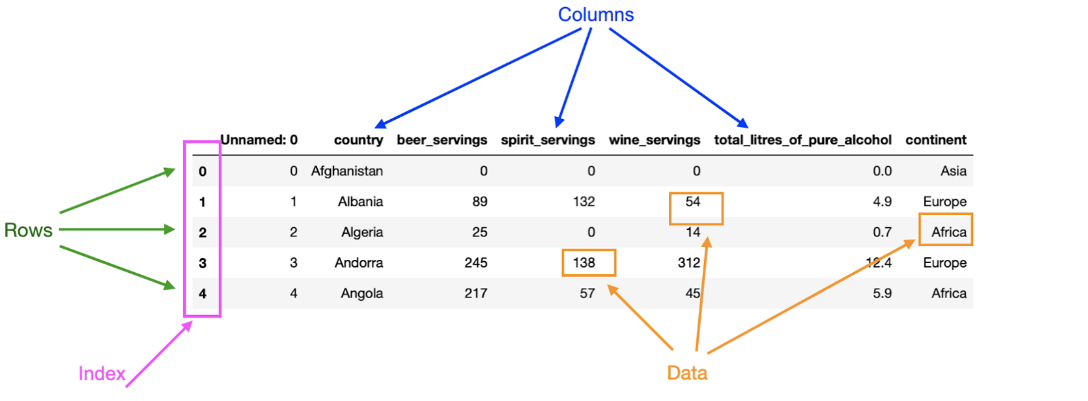
The table below lists the *functions* to export data to the most common file extensions.

| [to\_json()](https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html#io-json-writer) | Write JSON files |
| --- | --- |
| [to\_csv()](https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html#io-store-in-csv) | Write .csv files |
| [to\_html()](https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html#io-html) | Write HTML files |
| [to\_excel()](https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html#io-excel-writer) | Write Excel files |
| [to\_sql()](https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html#io-sql) | Write SQL files |

It is important to keep in mind that pandas can handle many more file types. Here is a comprehensive list of the [IO pandas *functions*.](https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html)

**What Is a *Dataframe*?**

A pandas *dataframe* is a two-dimensional data structure where the data is organized in rows and columns.

Pandas *dataframes* have four principal components: rows, columns, *index,* and data.

Given some data, each row represents an observation for that particular set of data. For example, in the image above, the first row contains the drinks consumption for Afghanistan, whereas the second row refers to drinks consumption data for Albania.

In a *dataframe*, columns can be seen as a label for each measurement taken. In the image above, each of the columns describes what data is stored in them; for example, the column “continent” contains information about which particular continent a country is in.

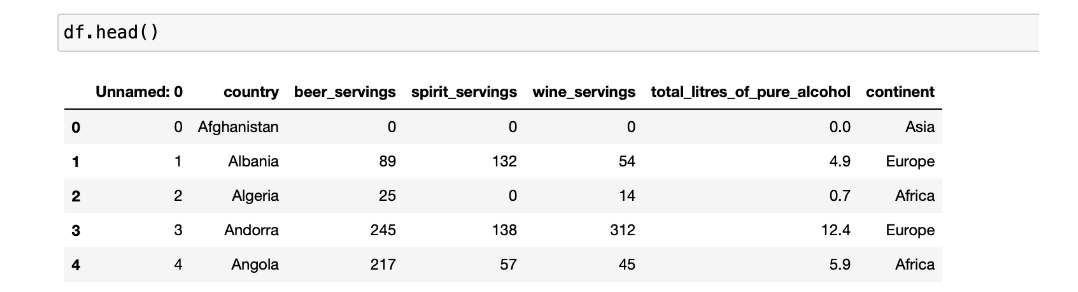
The *index* defines the location, or address, of each data point in the *dataframe*. Therefore, the *index* can be used to access data in a *dataframe*. In the image above, the first column contains the *index* of the *dataframe*. Columns also have an *index*: the column name. Therefore, the entry “Afghanistan” in the image above has row *index* 0 and column *index* (or label) “country.”

Finally, the data is simply the information stored in the *dataframe*. One particular thing to consider about *dataframes* is that they can contain different types of data. However, a column in a *dataframe* can only have one data type. Observe the *dataframe* in the image above. You can ascertain that the data across the *dataframe* is of different types: *integers* and *floats.* However, the data stored in a single column of the *dataframe* is always going to be of one data type. For example, all the entries in the column “total\_litres\_of\_pure\_alcohol” are of type *float.*

**Exploring a *Dataframe***

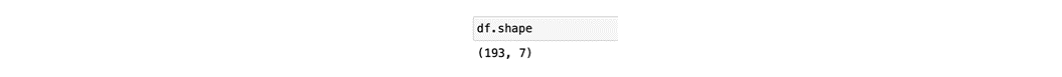
Once you have defined or imported your *dataframe* in your program, it is important that you become familiar with your data. Pandas offer a range of *functions* to facilitate this.

For example, suppose that you have read and saved the *dataframe* in the image below in your code as df. Often, one of the first things you would like to do is visualize the first few rows of your *dataframe* to see what type of data it contains. This can be achieved by using thehead() *function*, like this:



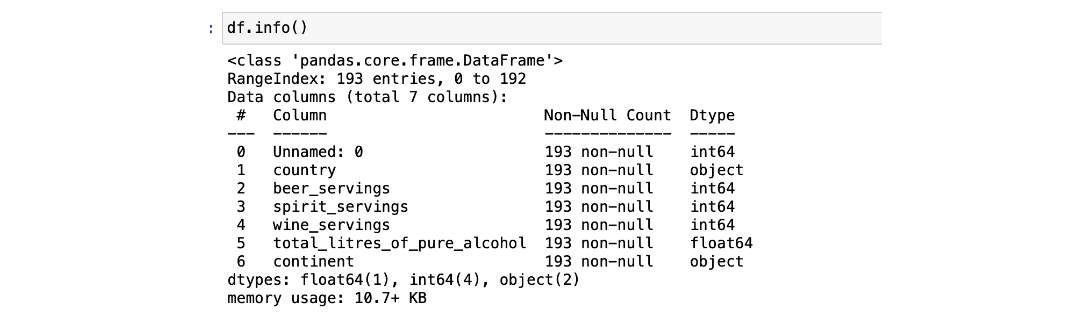
By default, the .head() *function* returns the first five rows of the *dataframe*. This number can be adjusted by passing the desired number of rows as an *integer* to the *function*.

To know the dimension of your *dataframe* — i.e., to get the total number of rows and columns — you can use the .shape *attribute*, like this:



You can see that the command returns a *tuple* containing the number of rows (193) and columns (7) in the *dataframe*.

To display the data types in each column of your *dataframe*, you can use the .info() *function*, like this:



This *function* returns a list of all the columns in your data set and the type of data that each column contains. Here, you can see the data types int64, float64, and object. Int64 and float64 are used to describe *integers* and *floats*, respectively. The object data type is used for columns that pandas doesn’t recognize as any other specific type. It means that all of the values in the column are *strings*.

*Data cleaning* is the process of removing extraneous data. This can include removing any data that may not appear useful or removing any areas that are missing data, either by filling in the missing information or by deleting the data entirely.

*One-hot encoding* is a technique used to transform categorical (non-numerical) data into binary values. This technique is fundamental when you want to apply any machine learning algorithm to your data.

**What Is Data Cleaning?**

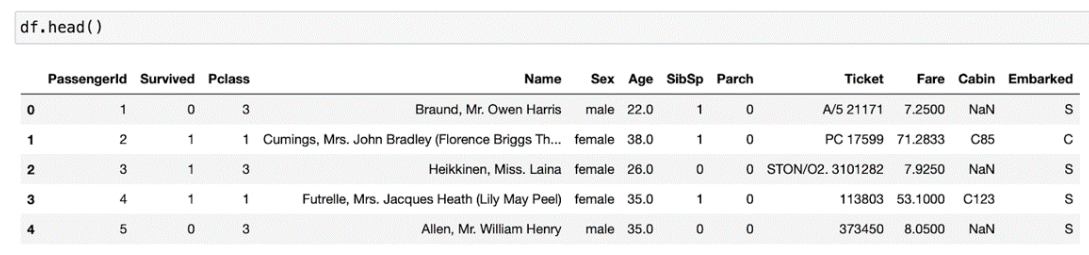
Data cleaning is the process of detecting and dealing with inaccurate, erroneous, or missing data.

This process can include dropping and modifying columns in your *dataframe*, modifying the *index*, and dealing with missing data.

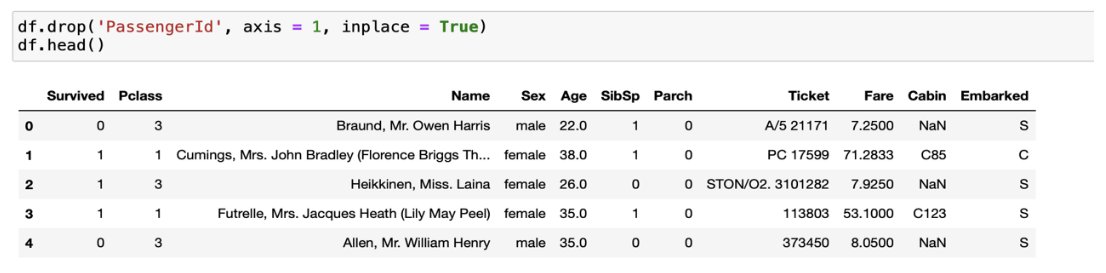
**Dropping Columns in a *Dataframe***

Dropping a column is the action of eliminating a column in a *dataframe*. You may want to drop a column for different reasons. For example, a column can be redundant or not useful for your analysis.

Consider the *dataframe* below, which contains data about the passengers who were on the Titanic:



You can see that the column “PassengerId” is used to number each passenger, but it doesn’t contain any useful information about their trip. Therefore, it makes sense to simply drop the column to make your *dataframe* simpler to use. To drop a column, you can use the drop() *function*, like this:



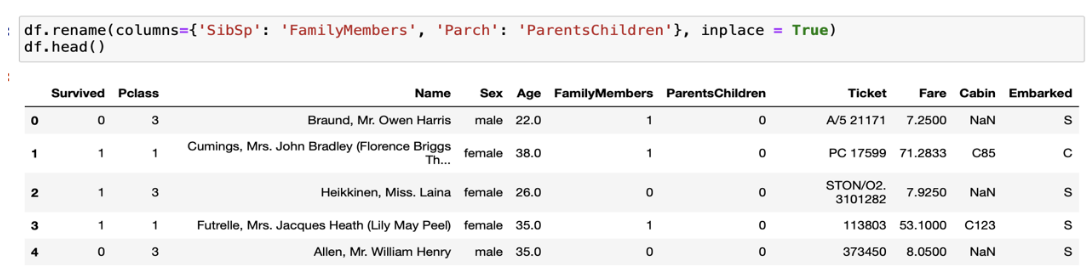
In the code above, the *argument* axis = 1 denotes that you want to drop the column with the label “PassengerId.” Setting the *argument* axis equal to 0 tells pandas that you want to drop a row instead. Finally, the argument inplace = True tells your program that you don’t want to make a copy of the modified *dataframe*.

**Modifying Columns**

When cleaning data, you may also want to pay attention to the column names and entries and think about whether you want to modify anything to make your analysis more practical.

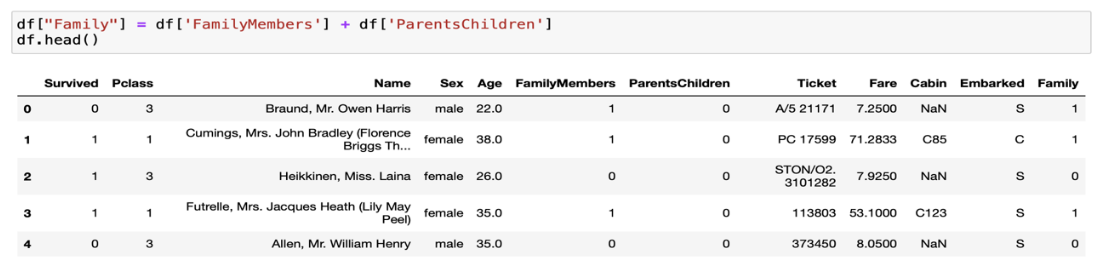
Consider the *dataframe* above again. The column “SibSp” describes the number of siblings and/or spouses aboard the Titanic. Similarly, the column “Parch” contains the number of parents and children aboard. You may deduce that these column names are not very descriptive and would prefer to change the titles of the columns to something more meaningful.

To change the name of a column in a *dataframe*, you can use the .rename() *function*, like this:



Sometimes, it’s also useful to modify the column values in a *dataframe*. This can include adding columns or converting their values to a more convenient format.

For example, in the *dataframe* above, it can be convenient to group the columns “FamilyMembers” and “ParentsChildren'' in a new column called “Family” that captures the total number of people traveling with a certain passenger. Observe the code below:

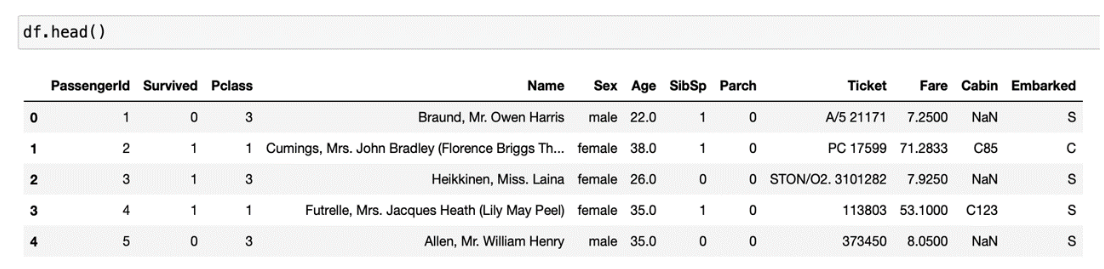


You can see that a new column “Family” has been created in the *dataframe*. The next logical step at this point is to drop the columns “FamilyMembers” and “ParentsChildren'' as they are redundant.

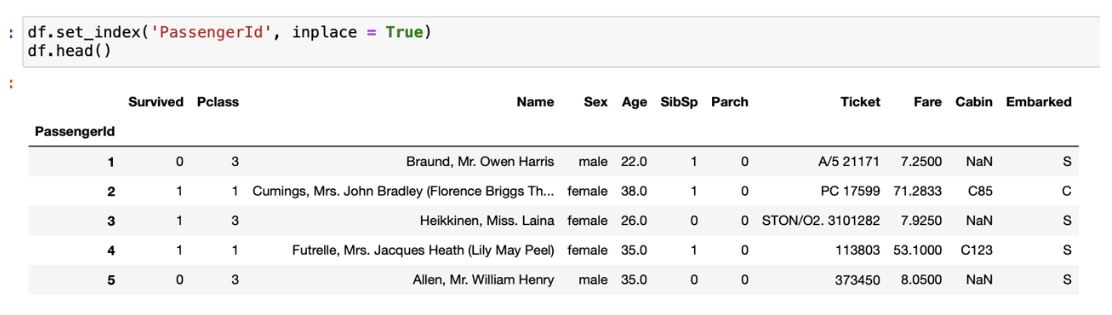
**Modifying the *Index* in a *Dataframe***

One can decide to modify the *index* column in a *dataframe* either by choosing an existing column as the new *index* or by choosing new values.

The set\_index() *function* is used to set the DataFrame *index* using existing columns. For example, in the *dataframe*:

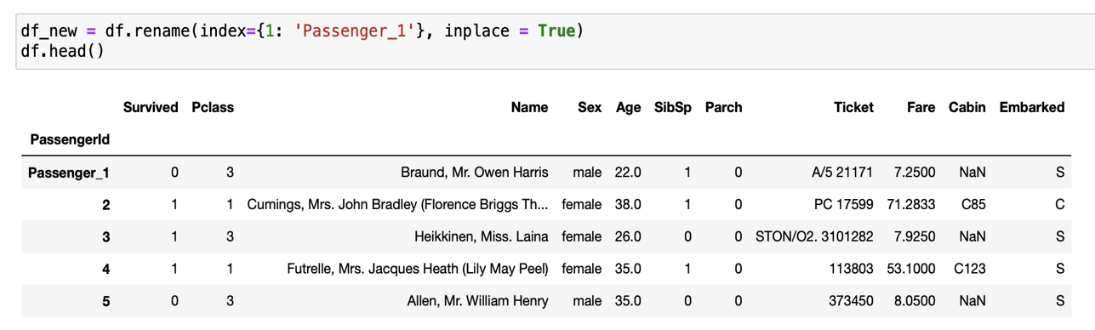


The df.set\_index('PassengerId', inplace = True) command returns:



Where you can see that now the new *index* is given by the column “PassengerId.”

You can also use the rename() *function* to change the *index*. Observe the code below:



You can see that the *index* 1 has been changed to “Passenger 1.”

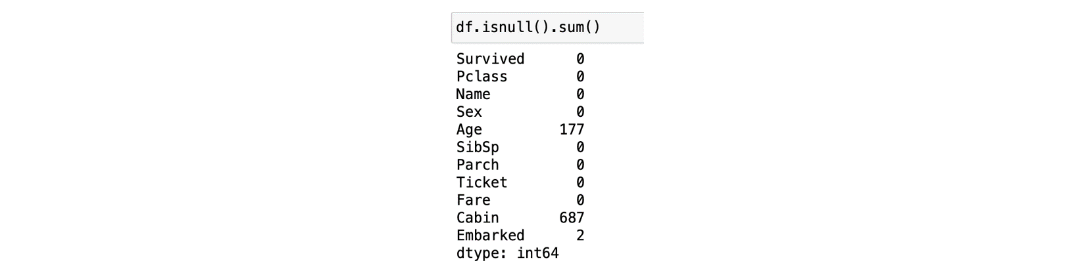
**Dealing with Missing Data**

Missing data is denoted in a *dataframe* with the notation NaN for “Not a Number.”

As a data scientist, you will often come across *dataframes* that have missing data. It is important that you understand why the data is missing and that you deal with it accordingly.

The first step to deal with missing data is to detect whether any value in your *dataframe* is missing and then to locate it.

The number of missing values in a *dataframe* can be found by using the is\_null() *function* followed by sum() . Given the *dataframe* df above, you get:



Therefore, you observe that you only have missing values in the “Age,” “Cabin,” and “Embarked” columns.

The second step when working with missing data is to decide how to deal with it. Some of the most common techniques to manage missing data are:

* Drop missing data.
  + This technique simply removes the missing data from a *dataframe*. All the missing data in the *dataframe* can be deleted at once by using the dropna() *function*. Although using this *function* may look convenient, you must pay attention to the consequences. In fact, if your *dataframe* has a lot of missing values, using this *function* may not be beneficial, as you may end up with very little useful data left.
  + Another strategy for deleting NaNs is to directly drop the columns or the rows where these values occur. If a column has a lot of missing data (like the “Cabin” column above), it’s often a good idea to drop the column altogether. If a column does not have many missing values, then one option would be to drop the rows in the *dataframe* that contain those missing values. Both of these actions can be achieved with thedrop() *function*.
* Impute data.
  + Imputing data is the technique of filling missing data with appropriate values. This technique is usually employed when there is not much data missing and when it is obvious which values to use to replace the NaNs. The pandas *function* that allows you to accomplish this is fillna() . Choosing a value to replace the missing values can be tricky. Some of the most common techniques include replacing a value with the most frequent one in that particular column or using the median value.

**Common *Functions* for Cleaning Data**

| [drop()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.drop.html) | *Function* to drop a column or to drop rows that contain missing values. |
| --- | --- |
| [rename()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.rename.html?highlight=rename#pandas.DataFrame.rename) | *Function* to change the name of a column or *index* in a *dataframe*. |
| [set\_index()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.set_index.html?highlight=set_index#pandas.DataFrame.set_index) | *Function* used to set a *dataframe* *index* using existing columns. |
| [dropna()](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.dropna.html) | *Function* to delete all of the missing data in a *dataframe* at once. |
| [isnull()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.isnull.html) | *Function* to detect missing values for an object. |
| [sum()](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.sum.html) | *Function* that returns the sum of the specified values. |
| [fillna()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.fillna.html) | *Function* used to fill missing data (NaNs) with appropriate data. |

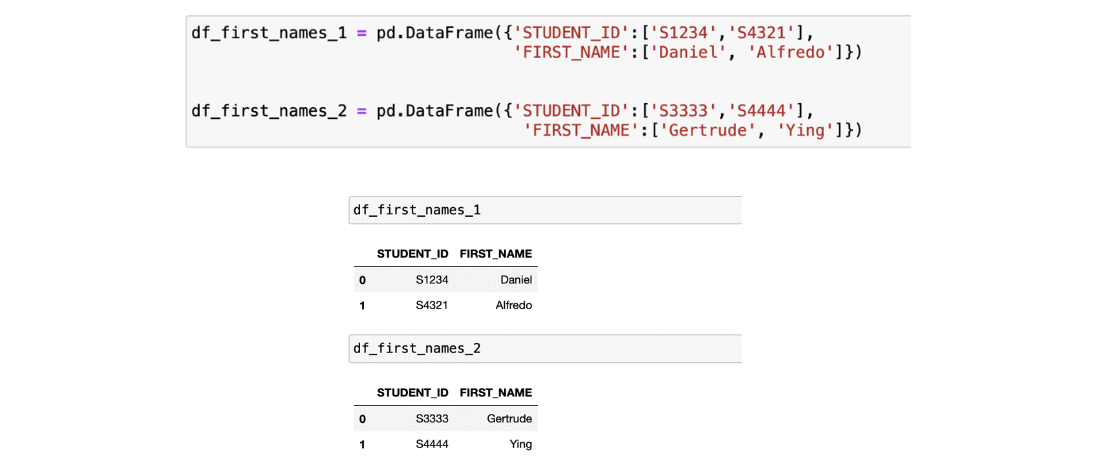
**Why Combine *Dataframes*?**

In many real-world applications, data is presented to you in different files. Therefore, it’s important that you know how to combine data sets into a single *dataframe* to analyze your data.

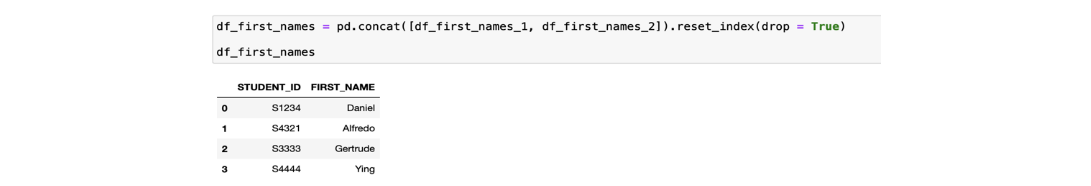
Depending on your needs, pandas provides two ways of combining *dataframes*: the union and the *join*.

**Union**

With the union, you can append the columns of one *dataframe* to another. The union can be achieved by using the.concat() *function*.

Suppose you have two *dataframes*, df\_first\_names\_1 and df\_first\_names\_2, defined as below: 

Since the columns in df\_first\_names\_1 and df\_first\_names\_2 are the same, you can concatenate them vertically to create a new *dataframe* by using the .concat() *function*:



In the code above, you have used the *function* reset\_index() so that the *index* in the resulting *dataframe* is in ascending order. Note that in order to perform a union between *dataframes*, the columns in all the original d*ataframes* must match.

***Join***

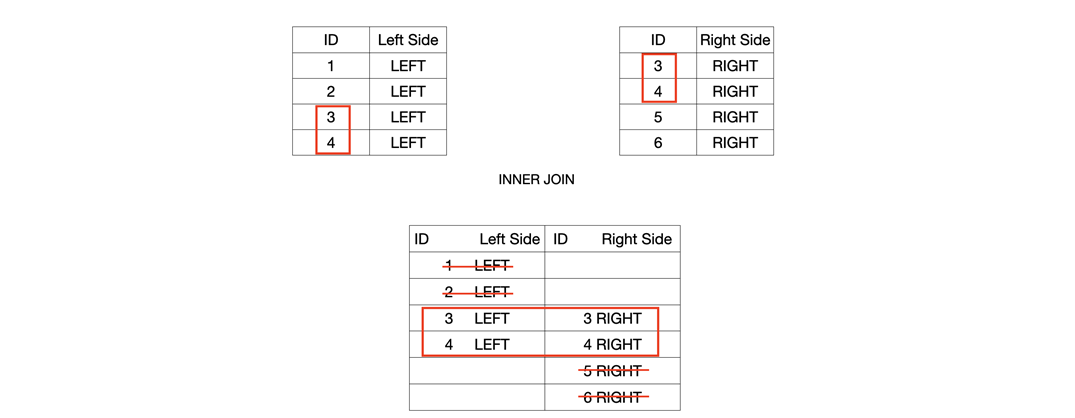
Sometimes, you may want to combine columns in different *dataframes* that contain common values. This technique is called *joining*, and it can be performed in pandas by using the .merge() *function*.

When performing a *join*, the columns containing the common values are called “*join* keys.”

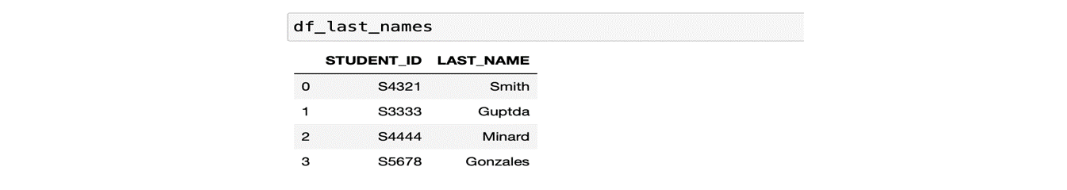
There are four different types of *joins*: inner, outer, left, and right. Next, you will review each one of them.

**Inner *Join***

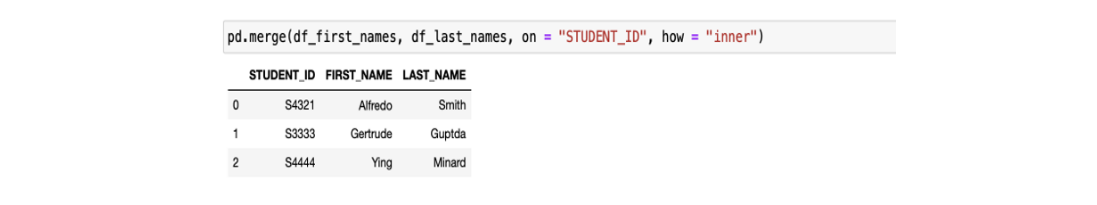
An inner *join* combines two *dataframes* based on a *join* key, and it returns a *dataframe* that only contains the rows that have matching values in both of the original *dataframes*.



As an example, consider again the df\_first\_names *dataframe* created above when you performed the union and the df\_last\_names *dataframe* defined below:



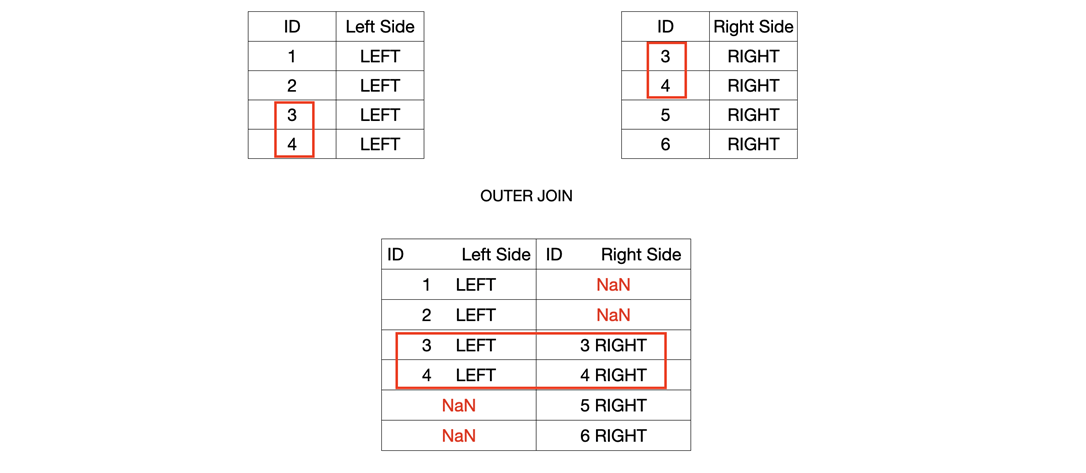
If you perform an inner *join* between df\_first\_names and df\_last\_names , choosing “STUDENT\_ID” as a *join* key, the resulting *dataframe* will only have the rows where the entries in “STUDENT\_ID” match in both of the original *dataframes*. Observe the code below:



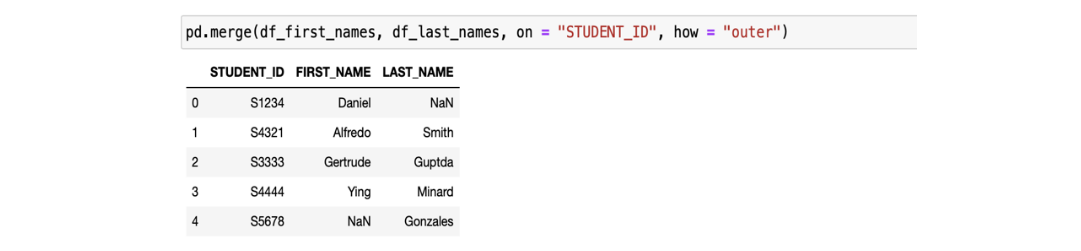
You can see that the rows for the students with first name “Daniel” and last name “Gonzales” have been deleted because the corresponding “STUDENT\_ID” key is not present in both *dataframes*.

**Outer *Join***

An outer *join* combines two *dataframes* based on a *join* key and returns a *dataframe* that contains all the rows in the original *dataframes*. The resulting *dataframe* will contain `NaN`, where data is missing in one of the *dataframes*.



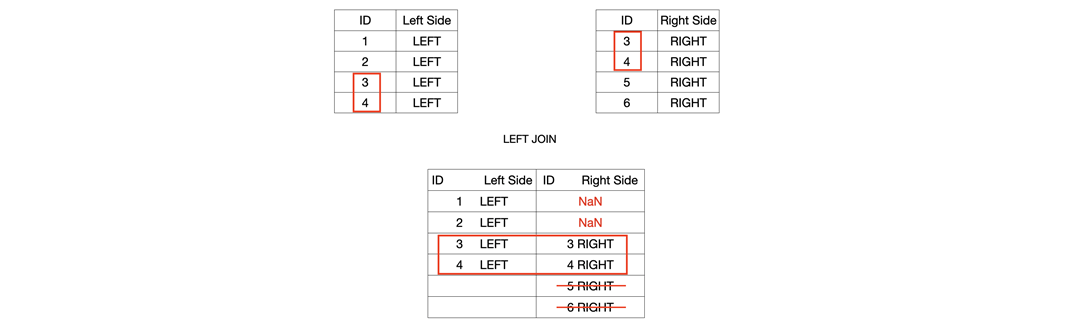
As an example, consider the df\_first\_names and df\_last\_names *dataframes* again. Performing an outer *join* on “STUDENT\_ID” will give the following result:



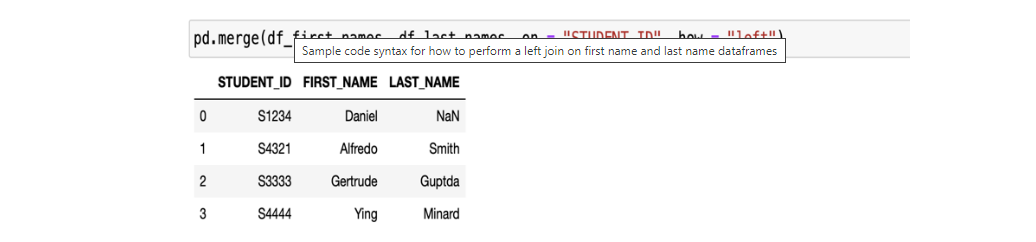
You can see that now all the rows from the original *dataframes* are present, and that missing values have been automatically filled with NaNs.

**Left *Join***

A left *join* discards rows from the right *dataframe* that do not have values for the *join* key(s) in the left *dataframe*.



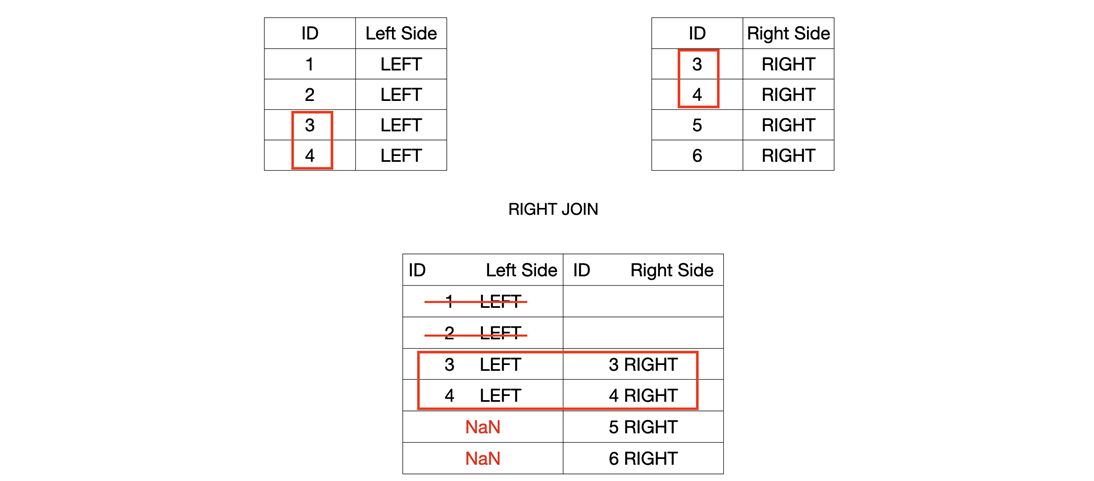
If you use the df\_first\_names and df\_last\_names *dataframes*, performing a left *join* on “STUDENT\_ID” will give the following result:



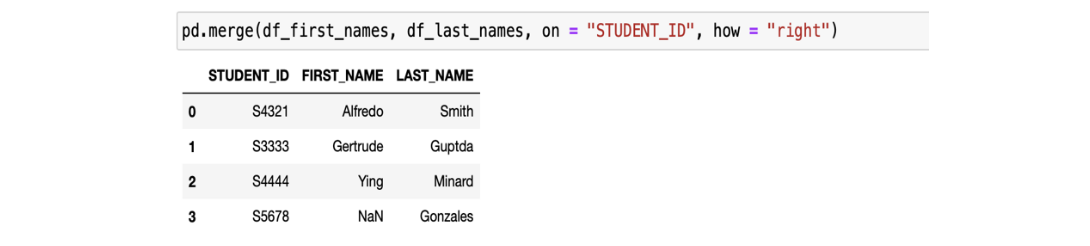
You can observe that the student with first name “Daniel” in the left *dataframe* is kept by the *join*, and his last name has been filled with a NaN entry. On the other hand, the entry in the right *dataframe* with last name equal to “Gonzales” has been dropped.

**Right *Join***

A right *join* discards rows from the left *dataframe* that do not have values for the *join* key(s) in the right *dataframe*.



Using the df\_first\_names and df\_last\_names *dataframes*, performing a right *join* on “STUDENT\_ID” will give the following result:



You can observe that the student with the first name “Daniel” in the left *dataframe* has been dropped. On the other hand, the entry in the right *dataframe* with last name equal to “Gonzales” is kept, and the entry corresponding to his first name has been filled with a NaN value.