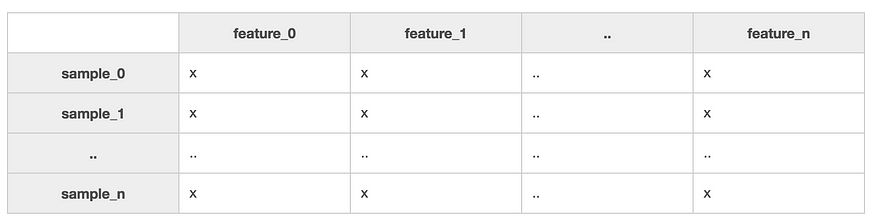
**Data Analysis with Python**

**Introduction**

This post is regarding Data analysis using Python language and I will walk you through examples of using high performance data libraries in python. Let’s begin

**Data Collection**

To analyze data using Python, your data needs to be organized into matrix of **samples** and **features**:



Whenever you collect data, any given feature will fall into one of two types:

**Continuous Features**

In the case of continuous features, there exist a measurable difference between possible feature values. Feature values usually are also a subset of all real numbers:

* Quantity
* Time
* Price
* Temperature

**Categorical Features**

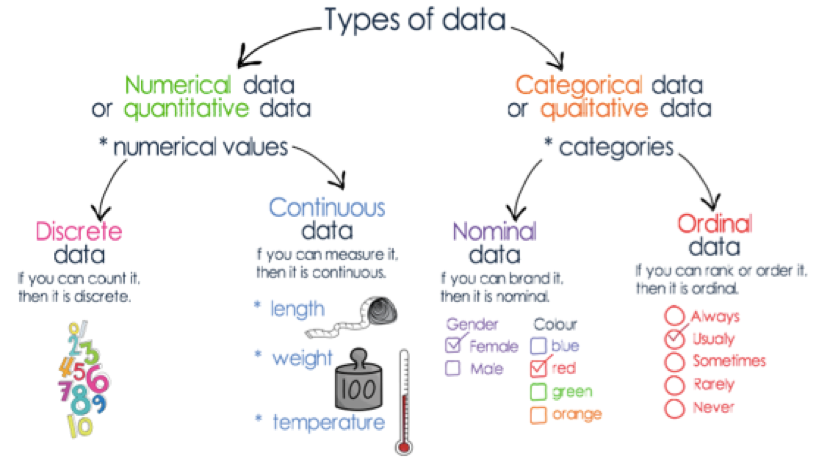
With categorical features, there is a specified number of discrete, possible feature values. These values may or may not have ordering to them. If they do have a natural ordering, they are called ordinal categorical features. Otherwise if there is no intrinsic ordering, they are called nominal categorical features.

*Nominal*

* Gender
* Colors
* Movies

*Ordinal*

* Small-Medium-Large
* 1–10 Years Old, 11–20 Years Old, 30–40 Years Old
* Happy, Neutral, Sad



**Data Loading**

Once you’ve collected your data, the next step is learning how to manipulate it efficiently.

*Pandas:*

*Pandas* is one of the most vital and actively developed high-performance data analysis libraries for Python, and you’ll be using it for all your data input, output, and manipulation needs. Pandas is built on top of another library NumPy. There are two data structures in Pandas you need to know how to work with. The first is the *series* object, a one-dimensional labeled array that represents a single column in your dataset.

import pandas as pd

*Pandas Series*:

Having all elements share the same units and data type make give you the ability to apply series-wide operations. Because of this, Pandas series must be homogeneous. They’re capable of storing any Python data type (integers, strings, floating point numbers, objects, etc.), but all the elements in a series **must** be of the same data type. The second structure you need to work with is a collection of series called a *dataframe*.

*Pandas Dataframe*:

To manipulate a dataset, you first need to load it into a dataframe. Different people prefer alternative methods of storing their data, so Pandas tries to make loading data easy no matter how it’s stored. Here are some methods for loading data:

from sqlalchemy import create\_engine  
engine = create\_engine('sqlite:///:memory:')  
  
sql\_dataframe = pd.read\_sql\_table('my\_table', engine, columns=['ColA', 'ColB'])  
xls\_dataframe = pd.read\_excel('my\_dataset.xlsx', 'Sheet1', na\_values=['NA', '?'])json\_dataframe = pd.read\_json('my\_dataset.json', orient='columns')csv\_dataframe = pd.read\_csv('my\_dataset.csv', sep=',')  
table\_dataframe= pd.read\_html('http://page.com/with/table.html')[0]

Note the return type of .read\_html(), it is a *Python list* of dataframes, one per HTML table found on the webpage.

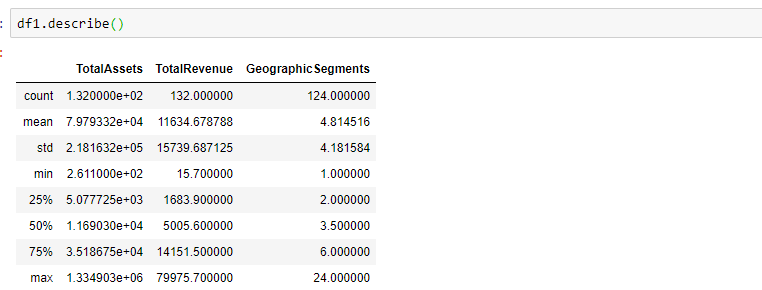
Also make sure you understand **fully** what the following parameters do:

* sep
* delimiter
* header
* names
* index\_col
* skipinitialspace
* skiprows
* na\_values
* thousands
* decimal

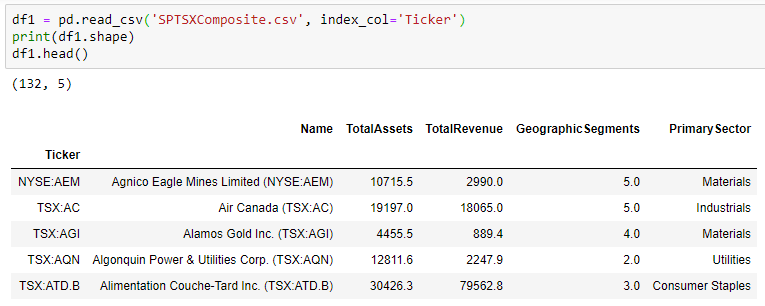
Many students new to data science run into problems because they rush through the mundane, data analysis portion of their work in their eagerness to get to the more exciting machine learning portion. But if they make mistakes here, for example by not knowing how to use index\_col to strip out id’s while reading their datasets with the .read\_csv() method, once they apply machine learning to their data, all of their findings are wrong.

**Data Summary**

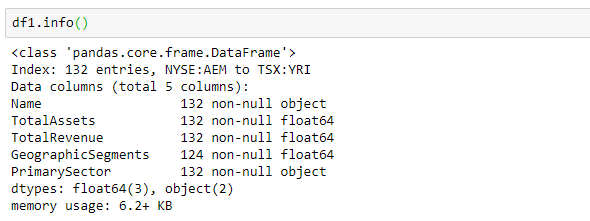
To see a descriptive statistical summary of your dataframe’s numeric columns using .describe():



To get a quick peek at your data by selecting its top or bottom few rows using .head() and .tail(). By default, it will show 5 records. Also, to get a quick peak at number of rows and columns for a dataframe, use .shape() method.



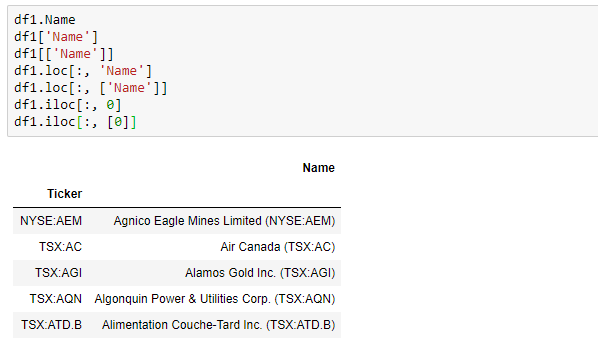
When you load up a dataframe, it’s always a good idea to see what data type Pandas assigned each column and how many entries are there. This is can be done using info() method. This is also very useful to check if there is any null value in any column.



**Slicin’ and Dicin’**

*Column Indexing*

A dataframe is essentially one or more series which have been ‘stitched’ together into a new data type. Pandas exposes *many* equivalent methods for slicing out those underlying series. You can slice by location, the way you would normally index into a regular Python list. You can slice by label, the way you would normally index into a Python dictionary. And like NumPy arrays, you can also index by boolean masks. Below are the different slicing methods to see the content of first column.



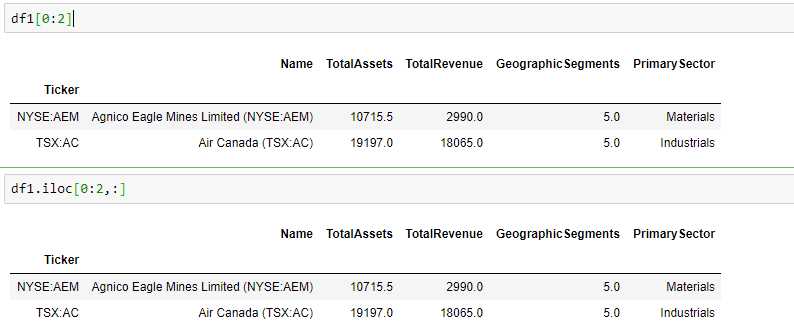
By using the column name in the code, it’s very easy to discern what is being pulled, and you don’t have to worry about the order of the columns. Doing this lookup of first matching the column name before slicing the column index is marginally slower than directly accessing the column by index. The .loc[] method selects by column label, .iloc[] selects by column index. Method .ix[] can be used as well whenever you want to use a hybrid approach of either, however, it is being deprecated.

Please be aware that if you use double brackets in syntax, even if you only specify a single column, the data type that you’ll get back is a *dataframe* as opposed to a *series*. So, the advantage of this list is that you can access more than one column.



*Row Indexing*

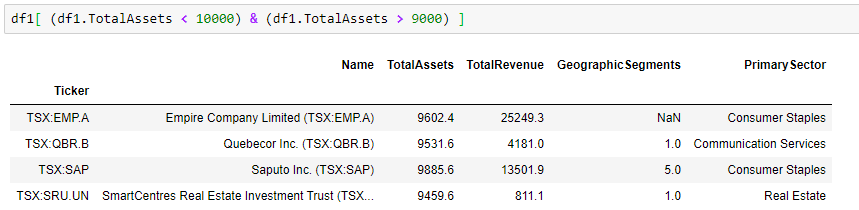
You can use any of the .loc[], .iloc[] methods to do selection by row, noting that the expected order is [row\_indexer, column\_indexer]:



The last important difference is that .loc[] and .ix[] are *inclusive* of the range of values selected, where .iloc[] is non-inclusive. In that sense, df.loc[0:1, :] would select the first *two* rows, but only the first row would be returned using df.iloc[0:1, :].

*Boolean Indexing*

Your dataframes and series can also be indexed with a *boolean operation* — a dataframe or series with the same dimensions as the one you are selecting from, but with every value either being set to True or False. You can create a new boolean series either by manually specifying the values, or by using a conditional. To index with your boolean series, simply feed it back into your regular series with using the [] bracket-selection syntax. You can further combine multiple boolean indexing conditionals together using the bit-wise logical operators | and &.



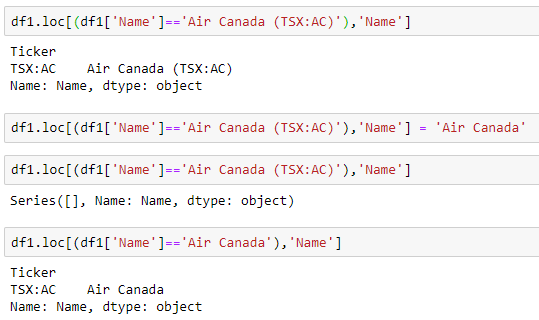
This is a bit counter-intuitive, as most people initially assume Pandas would support the regular, Python boolean operators or and and. The reason regular Python boolean operators cannot be used to combine Pandas boolean conditionals is because doing so causes ambiguity. There are two ways the following incorrect statement can be interpreted:

1. If the evaluation the statement (df1.TotalAssets < 10000) or the evaluation the statement (df1.TotalAssets > 9000) results in anything besides the False, then select all records in the dataset.
2. Select all columns belonging to rows in the dataset where either of the following statements are true: (df1.TotalAssets < 10000) or (df1.TotalAssets > 9000).

Option 2 is the desired functionality, but to avoid this ambiguity entirely, Pandas overloads bit-wise operators on its dataframe and series objects. Be **sure** to encapsulate each conditional in parenthesis to make this work.

*Writing to a Slice*

Something handy that you can do with a dataframe or series is write into a slice:



Take precaution while doing this, as you may encounter issues with non-homogeneous dataframes. It is far safer, and generally makes more sense, to do this sort of operation on a per column basis rather than across your entire dataframe.

In the next post, I would like to talk about Features representation, Features extraction, Features selection and Features importance.