1. Creating, Reading, and Writing

1. Getting Started

In [1]:
```
import pandas as pd
```

1. Creating data : There are two core objects in pandas : the DataFrame and the Series.

   1) DataFrame :
   A DataFrame is a table. It contains an array of individual entries, each of which has a certain value. Each entry correspond to a row and a column. DataFrame entries are not limited to integers.
   We are using the pd.DataFrame() constructor to generate these DataFrame objects. The syntax for declaring a new one is a dictionary whose keys are the column names, and whose values are a list of entries. This is the standard way of constructing a new DataFrame, and the one you are most likely to encounter. The list of row labels used in a DataFrame is known as an index. We can assign values to it by using an index parameter in our constructor :

   2) Series :
   A Seris, by contrast, is a sequence of data values. If a DataFrame is a table, a Series is a list. And in fact you can Create one with nothing more than a list :
   A Series is, in essence, a single column of a DataFrame. So you can assign column values to the Series the same way as before, using an index parameter. However, a Series does not have a column name, it only has one overall name :

In [4]:
```
pd.DataFrame({'BoB' : ['I liked it', 'It was awful.'], 'Sue' : ['Pretty good.', 'Bland.']})

pd.Series([30, 35, 40], index = ['2015 Sales', '2016 Sales', '2017 Sales'], name = 'Product A')
```

Out[4]:
```
2015 Sales    30
2016 Sales    35
2017 Sales    40
Name: Product A, dtype: int64
```

1. Reading data files :
   Being able to create a DataFrame or Series by hand is handy. But, most of the time, We won't actually be creating our own data by hand. We'll use the pd.read_csv() function to read the data into a DataFrame.
   - df.shape : we can use the shape attribute to check how large the resulting DataFrame is
   - df.head() : We can examine the contents of the resultant DataFrame which grabs the first five rows

2. Indexing, Selecting & Assigning

1. Native accesors :
   Native Python objects provide good ways of indexing data. Pandas carries all of these over, which helps make it easy to start with. In Python, We can access the property of an objects by accessing it as an attribute. A book objects, for example, might have a title property, which we can acess by calling 'book.title'. Columns in a pandas DataFrame work in much the same way.
   If we have a Python dictionray, We can access its values using the indexing [] operator. 'book['title']'.
   Doesn't a pandas Series look kind of like a fancy dictionary? It pretty much is, so it's no surprise that, to drilll down to a single specific value, we need only use the indexing operator [] once more :

1. Indexing in pandas :
   The indexing operator and attribute selection are nice because they work just like they do in the rest of the Python ecosystem. As a novice, this makes them easy to pick up and use. Hoever, pandas has its own accesor operators, loc and iloc.
   - iloc : The first is index-based selection : selecting data based on its numerical position in the data. : revies.iloc[:, 0]
   - loc : The secodn paradigm for attribute selection is the one followed by the loc operator : label-based selection. loc uses the information in teh indices to do its work.

1. Manipulating the index :
   Label-based selection derives its power from the labels in the index. Critically, the index we use is not immutable. We can mainpulate the index in any way we see fit.
   - set_index() :

1. Conditional selection :
   We can choose data with conditions. When we make operation, this operation produced a Series of True/False booleans based on the column of each record. This result can then be used inside of loc to select the relevant data.
   - isin() : lets you select data whose valeu is in a list of valeus.
   - isnull() : let you highlight value whic are empty.
   - notnull() : companion to isnull()

3. Summary Functions and Maps

1. Summary functions :
   Pandas provides many simple "summary functions" which restructure the data in some useful way.
   - describe() : This method generates a high-level summary of the attributes of the given column. It is type-aware, meaning that its output changes based on the data type of the input. The output above inly makes sense for numerical data
   - unique() : To see a list of unique values
   - value_counts() : To see a list of unique valeus and how often they occur in the dataset.

1. Maps :
   A map in a term, borrowed from mathmeatics, for a function that takes one set of values and "maps" them to another set of values. In data science we often have a need for creating new represntations from existing data, or for transforming data from the format it is in now to teh format that we want it to be in later.
   - map() : use with lambda x : x ~. The function you pass to map() should expect a single value from the Series, and return a transformed version of that value.
   - apply() : It is the equivalent metod if we want to transform a whole DataFrame by calling a custom method on each row

4. Grouping and Sorting

1. Groupwise analysis :
   - groupby() : Created a group of reviews which allotted the same point values to the given wines. Then for each of these groups, we grabbed the points() column and counted how many times it appeared.
   - value_counts() : groupby()['col'].count() | groupby().size()
   - groupby().appy(lambda df : condition) : We can use directly apply() method, and we can then manipulate the data in any way we see fit.
   - groupby().agg() : Another groupby() method worth mentioning is agg(), which lets us run a bunch of different functions on our DataFrame simulatneously.

1. Multi-indexes :
   In all of the xamples we've seen thus far we've been working with DataFrfame or Series objects with a single-label index. grouby() is slightly different in the fact that, depending on the operation we run, it will sometimes result in what is called a multi-index. Multi-indices have several methods for dealing with their tiered structure which are absent for single-level indices. They are require two levels of labels to retreive a Value. Dealing wiht multi-index output is a common "gotcha" for users new to pandas.
   - reset_index() : In general the multi-index method we will use most often is the one for converting back to a regular index.

1. Sorting :
   - sort_values() : to get data in the order wnat it in we can sort it ourselves.

5. Data types and Missing values

1. Dtypes :
   The data type for a column in a DataFrame or a Series is known as the dtype.
   - dtype : you can use the dtype property to grab the type of a specific column.
   - dtypes : property turns the dtype of every column in the DataFrame
   - astype() : It's possible to convert a column of one type into another.

1. Missing data :
   Entries missing values are given the value 'NaN', short for 'Not a Number'.
   - is.null() : Find index of NaN values.
   - fillna() : Replacing missing values is a common operation.

6. Renaming and Combining

1. Renaming :
   The first function we'll introduce here is rename(), which lets us change index names and/or column names
   - rename(columns = {'col' : 'newcol'})

1. Combining :
   When performing operations on a dataset, we will sometimes need to combine different DataFrames and/or Series in non-trival ways. Pandas has three core method for doing this. In order of increasing complexity, these are concat(), join(), and merge(). Most of what merge() can do can also be done more simply with join(), so we will omit it and focus on the focus on the first two functions here.
   - concat() : The simplest combining method is concat(). Given a list of elements, this function will smush those elements togheter along an axis.
   - join() : lets us combine different DataFrame objects which have an index in common.
   - merge()