PANDAS

What's Pandas?

Pandas tool for data cleaning, transforming, and analyzing.

For example, Pandas will extract the data from that CSV into a DataFrame — a table — then let us do things like:

- Calculate statistics and answer questions about the data, like
    - What's the average, median, max, or min of each column?
    - Does column A correlate with column B?
    - What does the distribution of data in column C look like?
- Clean the data by doing things like removing missing values and filtering rows or columns by some criteria
- Visualize the data with Matplotlib. Plot bars, lines, histograms, bubbles, and more.
- Store the cleaned, transformed data back into a CSV, other file or database

Pandas is built on top of the NumPy package.

Data in pandas is used to feed statistical analysis in SciPy, plotting functions from Matplotlib, and machine learning algorithms in Scikit-learn.

Jupyter Notebook is environment for using pandas to do data exploration and modeling.

Install and import

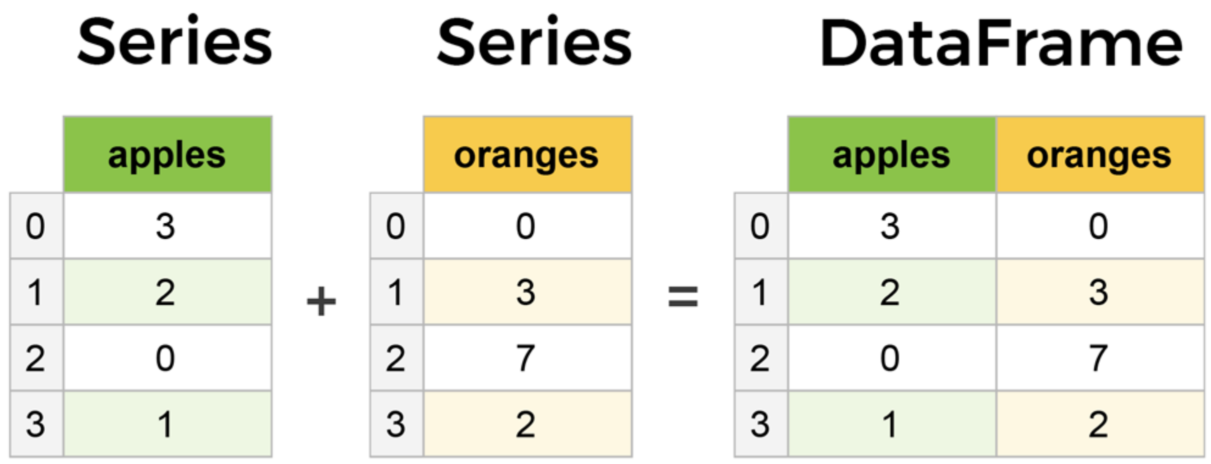
`pip install pandas`

```
import pandas as pd
```

Core components of pandas: Series and DataFrames

The primary two components of pandas are the `Series` and `DataFrame`.

A `Series` is a column, and a `DataFrame` is a multi-dimensional table made up of a collection of Series.



Series

	apples
0	3
1	2
2	0
3	1

+

Series

	oranges
0	0
1	3
2	7
3	2

=

DataFrame

	apples	oranges
0	3	0
1	2	3
2	0	7
3	1	2

Creating DataFrames from scratch (with dictionary)

import pandas as pd #I am importing pandas as pd

from pandas import Series, DataFrame

```
data = {
    'apples': [3, 2, 0, 1],
    'oranges': [0, 3, 7, 2]
}
```

And then pass it to the pandas DataFrame constructor:

```
purchases = pd.DataFrame(data)
purchases
```

Each *(key, value)* item in `data` corresponds to a *column* in the resulting DataFrame.

The Index of this DataFrame was given to us on creation as the numbers 0-3, but we could also create our own when we initialize the DataFrame.

```
purchases = pd.DataFrame(data, index=['Murthy', 'Raju', 'Kiran', 'Dravid'])
purchases
```

OUT:

	apples	oranges
Murthy	3	0
Raju	2	3
Kiran	0	7
Dravid	1	2

So now we could locate a customer's order by using their name:

```
purchases.loc['Murthy']
```

OUT:

```
apples     3
oranges    0
Name: Murthy, dtype: int64
```

```
import pandas as pd
import numpy as np  # numpy allows to work with array/matrix with numbers
df = pd.DataFrame(np.random.rand(100,5))
df.head() # it will print top 5 records  (df.tail())

import json
import requests
response = requests.get("https://jsonplaceholder.typicode.com/todos")
todos = json.loads(response.text)
json.dumps(todos, indent=2)
```

Reading data from CSVs

```
df = pd.read_csv('purchases.csv')
df
```

CSVs don't have indexes like our DataFrames, so all we need to do is just designate the `index_col` when reading:

```
df = pd.read_csv('purchases.csv', index_col=0)
df
```

Here we're setting the index to be column zero.

Reading data from JSON

```
df = pd.read_json('purchases.json')
df
```

Reading data from a SQL database

```
pip install pysqlite3
```

`sqlite3` is used to create a connection to a database which we can then use to generate a DataFrame through a `SELECT` query.

```
import sqlite3
con = sqlite3.connect("database.db")
df = pd.read_sql_query("SELECT * FROM purchases", con)
df
```

Converting back to a CSV, JSON, or SQL

```
df.to_csv('new_purchases.csv')

df.to_json('new_purchases.json')

df.to_sql('new_purchases', con)
```

DataFrame operations

```
movies_df = pd.read_csv("Movie-Data.csv", index_col="Title")
```

Viewing data

```
movies_df.head()
```

`.head()` outputs the first five rows of wer DataFrame by default, but we could also pass a number as well: `movies_df.head(10)` would output the top ten rows

To see the last five rows use `.tail()`. `tail()` also accepts a number, and in this case we printing the bottom two rows.:

```
movies_df.tail(2)
```

Typically when we load in a dataset, we like to view the first five or so rows to see what's under the hood. Here we can see the names of each column, the index, and examples of values in each row.

We'll notice that the index in our DataFrame is the *Title* column, which we can tell by how the word *Title* is slightly lower than the rest of the columns.

Getting info about wer data

```
movies_df.info()
movies_df.shape
```

Handling duplicates

```
temp_df = movies_df.append(movies_df)

temp_df.shape
```

OUT:

```
(2000, 11)
```

Using `append()` will return a copy without affecting the original DataFrame. We are capturing this copy in `temp` so we aren't working with the real data.
Notice call `.shape` quickly proves our DataFrame rows have doubled.
Now we can try dropping duplicates:

```
temp_df = temp_df.drop_duplicates()

temp_df.shape
```

OUT:

```
(1000, 11)
```

Using `inplace=True` will modify the DataFrame object in place:

```
temp_df.drop_duplicates(inplace=True)
```

Now our `temp_df` *will* have the transformed data automatically.

Another important argument for `drop_duplicates()` is `keep`, which has three possible options:

- `first`: (default) Drop duplicates except for the first occurrence.
- `last`: Drop duplicates except for the last occurrence.
- `False`: Drop all duplicates.

`keep` will drop all duplicates. If two rows are the same then both will be dropped.

```
temp_df = movies_df.append(movies_df)  # make a new copy

temp_df.drop_duplicates(inplace=True, keep=False)

temp_df.shape
```

OUT:

```
(0, 11)
```

Column cleanup

Here's how to print the column names of our dataset:

```
movies_df.columns
```

OUT:

```
Index(['Rank', 'Genre', 'Description', 'Director', 'Actors', 'Year',
       'Runtime (Minutes)', 'Rating', 'Votes', 'Revenue (Millions)',
       'Metascore'],
      dtype='object')
```

Rename columns:

```
movies_df.rename(columns={
        'Runtime (Minutes)': 'Runtime',
        'Revenue (Millions)': 'Revenue_millions'
    }, inplace=True)
```

```
movies_df.columns
```

OUT:

```
Index(['Rank', 'Genre', 'Description', 'Director', 'Actors', 'Year', 'Runtime',
       'Rating', 'Votes', 'Revenue_millions', 'Metascore'],
      dtype='object')
```

How to work with missing values

1. Get rid of rows or columns with nulls
2. Replace nulls with non-null values, a technique known as imputation

Let's calculate to total number of nulls in each column of our dataset. The first step is to check which cells in our DataFrame are null:

```
movies_df.isnull()
```

`isnull()` returns a DataFrame where each cell is either True or False depending on that cell's null status.

To count the number of nulls in each column we use an aggregate function for summing:

```
movies_df.isnull().sum()
```

OUT:

```
rank                0
genre               0
description         0
director            0
actors              0
year                0
runtime             0
rating              0
votes               0
revenue_millions  128
metascore          64
dtype: int64
```

`.isnull()` just by iteself isn't very useful, and is usually used in conjunction with other methods, like `sum()`.

We can see now that our data has 128 missing values for `revenue_millions` and 64 missing values for `metascore`.

Removing null values

Data Scientists and Analysts regularly face the dilemma of dropping or imputing null values, and is a decision that requires intimate knowledge of wer data and its context. Overall, removing null data is only suggested if we have a small amount of missing data.

Remove nulls is pretty simple:

```
movies_df.dropna()
```

This operation will delete any row with at least a single null value, but it will return a new DataFrame without altering the original one. We could specify `inplace=True` in this method as well.

So in the case of our dataset, this operation would remove 128 rows where `revenue_millions` is null and 64 rows where `metascore` is null.

Other than just dropping rows, we can also drop columns with null values by setting `axis=1`:

```
movies_df.dropna(axis=1)
```

In our dataset, this operation would drop the `revenue_millions` and `metascore` columns

Imputation

Imputation is a conventional feature engineering technique used to keep valuable data that have null values.

There may be instances where dropping every row with a null value removes too big a chunk from wer dataset, so instead we can impute that null with another value, usually the mean or the median of that column.

```
revenue = movies_df['revenue_millions']
```

Using square brackets is the general way we select columns in a DataFrame.

```
revenue.head()
```

OUT:

```
Title
Guardians of the Galaxy    333.13
Prometheus                 126.46
Split                      138.12
Sing                       270.32
Suicide Squad              325.02
Name: revenue_millions, dtype: float64
```

Slightly different formatting than a DataFrame, but we still have our `Title` index.

We'll impute the missing values of revenue using the mean. Here's the mean value:

```
revenue_mean = revenue.mean()

revenue_mean
```

OUT:

```
82.95637614678897
```

With the mean, let's fill the nulls using `fillna()`:

```
revenue.fillna(revenue_mean, inplace=True)
```

We have now replaced all nulls in `revenue` with the mean of the column. Notice that by using `inplace=True` we have actually affected the original `movies_df`:

```
movies_df.isnull().sum()
```

OUT:

```
rank                0
genre               0
description         0
director            0
actors              0
year                0
runtime             0
rating              0
votes               0
revenue_millions    0
metascore          64
dtype: int64
```

Understanding variables

Using `describe()` on an entire DataFrame we can get a summary of the distribution of continuous variables:

```
movies_df.describe()
```

`.describe()` can also be used on a categorical variable to get the count of rows, unique count of categories, top category, and freq of top category:

```
movies_df['genre'].describe()
```

OUT:

```
count                        1000
unique                        207
top       Action,Adventure,Sci-Fi
freq                           50
Name: genre, dtype: object
```

This tells us that the genre column has 207 unique values, the top value is Action/Adventure/Sci-Fi, which shows up 50 times (freq).

`.value_counts()` can tell us the frequency of all values in a column:

```
movies_df['genre'].value_counts().head(10)
```

```
OUT:
Action,Adventure,Sci-Fi       50
Drama                         48
Comedy,Drama,Romance          35
Comedy                        32
Drama,Romance                 31
Action,Adventure,Fantasy      27
Comedy,Drama                  27
Animation,Adventure,Comedy    27
Comedy,Romance                26
Crime,Drama,Thriller          24
Name: genre, dtype: int64
```

Relationships between continuous variables
By using the correlation method `.corr()` we can generate the relationship between each continuous variable:

```
movies_df.corr()
```

Correlation tables are a numerical representation of the bivariate relationships in the dataset.

Positive numbers indicate a positive correlation — one goes up the other goes up — and negative numbers represent an inverse correlation — one goes up the other goes down. 1.0 indicates a perfect correlation.

So looking in the first row, first column we see `rank` has a perfect correlation with itself, which is obvious. On the other hand, the correlation between `votes` and `revenue_millions` is 0.6. A little more interesting.

DataFrame slicing, selecting, extracting

By column

```
genre_col = movies_df['genre']

type(genre_col)
```

OUT:

```
pandas.core.series.Series
```

This will return a *Series*. To extract a column as a *DataFrame*, need to pass a list of column names. In our case that's just a single column:

```
genre_col = movies_df[['genre']]

type(genre_col)
pandas.core.frame.DataFrame
```

Since it's just a list, adding another column name is easy:

```
subset = movies_df[['genre', 'rating']]
subset.head()
```

By rows

For rows, we have two options:

- `.loc` - locates by name
- `.iloc` - locates by numerical index

```
prom = movies_df.loc["Prometheus"]

prom
```

OUT:

```
rank                                                 2
genre                              Adventure,Mystery,Sci-Fi
description     Following clues to the origin of mankind, a te...
```

```
director                                   Ridley Scott
actors          Noomi Rapace, Logan Marshall-Green, Michael Fa...
year                                               2012
runtime                                             124
rating                                                7
votes                                            485820
revenue_millions                                 126.46
metascore                                            65
Name: Prometheus, dtype: object
```

On the other hand, with `iloc` we give it the numerical index of Prometheus:

```
prom = movies_df.iloc[1]
```

`loc` and `iloc` can be thought of as similar to Python `list` slicing. To show this even further, let's select multiple rows.
In Python, just slice with brackets like `example_list[1:4]`. It's works the same way in pandas:

```
movie_subset = movies_df.loc['Prometheus':'Sing']

movie_subset = movies_df.iloc[1:4]

movie_subset
```

Conditional selections

```
condition = (movies_df['director'] == "Ridley Scott")

condition.head()
```

To return the rows where that condition is True we have to pass this operation into the DataFrame:

```
movies_df[movies_df['director'] == "Ridley Scott"]
```

We can get used to looking at these conditionals by reading it like:

Select `movies_df` where `movies_df` director equals Ridley Scott.
Let's look at conditional selections using numerical values by filtering the DataFrame by ratings:

```
movies_df[movies_df['rating'] >= 8.6].head(3)
```

We can make some richer conditionals by using logical operators `|` for "or" and `&` for "and".

Let's filter the the DataFrame to show only movies by Christopher Nolan OR Ridley Scott:

```
movies_df[(movies_df['director'] == 'Christopher Nolan') | (movies_df['director'] ==
'Ridley Scott')].head()
```

We need to make sure to group evaluations with parentheses so Python knows how to evaluate the conditional.

Using the `isin()` method we could make this more concise though:

```
movies_df[movies_df['director'].isin(['Christopher Nolan', 'Ridley Scott'])].head()
```

Let's say we want all movies that were released between 2005 and 2010, have a rating above 8.0, but made below the 25th percentile in revenue.

```
movies_df[
    ((movies_df['year'] >= 2005) & (movies_df['year'] <= 2010))
    & (movies_df['rating'] > 8.0)
    & (movies_df['revenue_millions'] < movies_df['revenue_millions'].quantile(0.25))
]
```

Applying functions

It is possible to iterate over a DataFrame or Series as we would with a list, but doing so — especially on large datasets — is very slow.

An efficient alternative is to `apply()` a function to the dataset. For example, we could use a function to convert movies with an 8.0 or greater to a string value of "good" and the rest to "bad" and use this transformed values to create a new column.

First we would create a function that, when given a rating, determines if it's good or bad:

```
def rating_function(x):
    if x >= 8.0:
        return "good"
    else:
        return "bad"
```

Now we want to send the entire rating column through this function, which is what `apply()` does:

```
movies_df["rating_category"] = movies_df["rating"].apply(rating_function)
movies_df.head(2)
```

The `.apply()` method passes every value in the `rating` column through the `rating_function` and then returns a new Series. This Series is then assigned to a new column called `rating_category`.

We can also use anonymous functions as well. This lambda function achieves the same result as `rating_function`:

```
movies_df["rating_category"] = movies_df["rating"].apply(lambda x: 'good' if x >= 8.0 else
'bad')

movies_df.head(2)
```

Overall, using `apply()` will be much faster than iterating manually over rows because pandas is utilizing vectorization.

Vectorization: a style of computer programming where operations are applied to whole arrays instead of individual elements —Wikipedia

A good example of high usage of `apply()` is during natural language processing (NLP) work. We'll need to apply all sorts of text cleaning functions to strings to prepare for machine learning.

Plotting

pandas allows to integrates with Matplotlib, so we get the ability to plot directly off DataFrames and Series.

`pip install matplotlib`

```
import matplotlib.pyplot as plt
plt.rcParams.update({'font.size': 20, 'figure.figsize': (10, 8)})
# set font and plot size to be larger
```

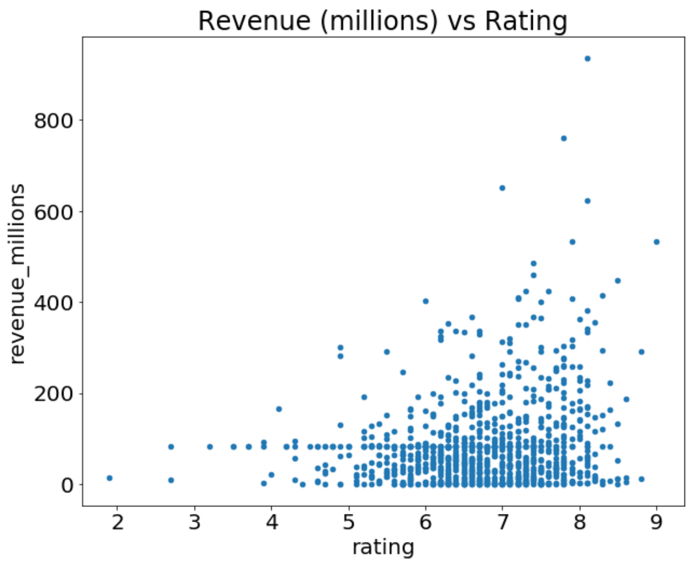
Plotting Tip

For categorical variables utilize Bar Charts* and Boxplots.

For continuous variables utilize Histograms, Scatterplots, Line graphs, and Boxplots.

Let's plot the relationship between ratings and revenue. All we need to do is call `.plot()` on `movies_df` with some info about how to construct the plot:

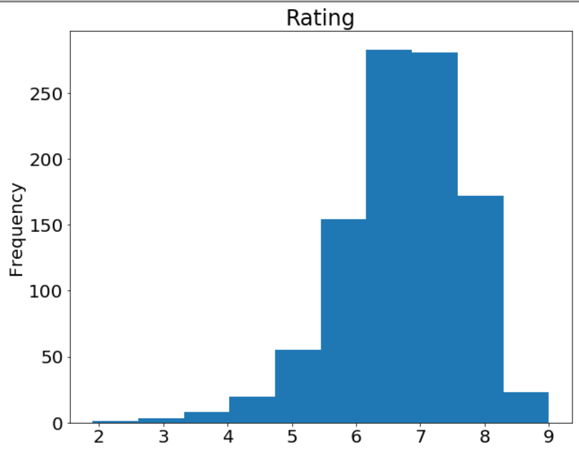
```
movies_df.plot(kind='scatter', x='rating', y='revenue_millions', title='Revenue (millions)
vs Rating');
```



What's with the semicolon? It's not a syntax error, just a way to hide the `<matplotlib.axes._subplots.AxesSubplot at 0x26613b5cc18>` output when plotting in Jupyter notebooks.

If we want to plot a simple Histogram based on a single column, we can call plot on a column:

```
movies_df['rating'].plot(kind='hist', title='Rating');
```



there's a graphical representation of the interquartile range, called the Boxplot. Let's recall what `describe()` gives us on the ratings column:

```
movies_df['rating'].describe()
```

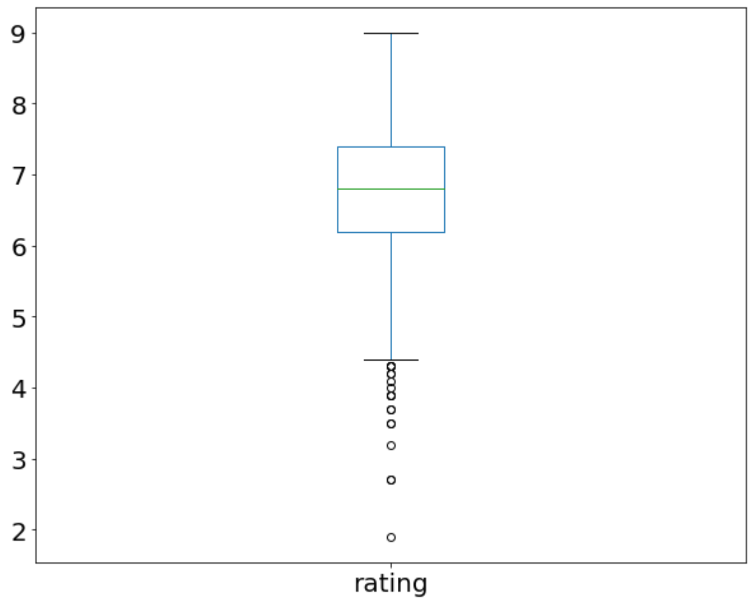
OUT:

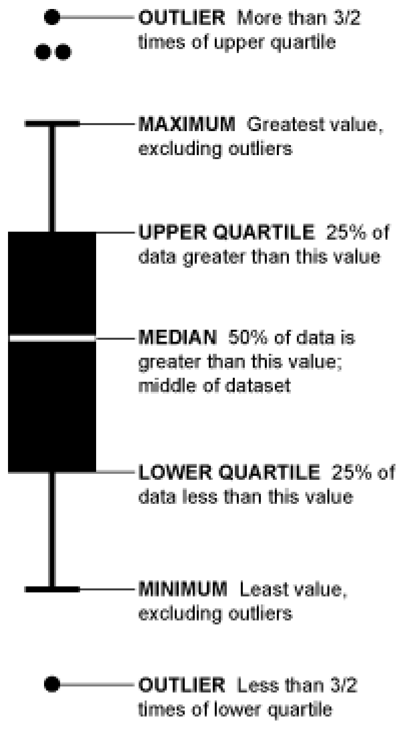
```
count    1000.000000
mean        6.723200
std         0.945429
min         1.900000
25%         6.200000
50%         6.800000
75%         7.400000
max         9.000000
Name: rating, dtype: float64
```

Using a Boxplot we can visualize this data:

```
movies_df['rating'].plot(kind="box");
```

RESULT:




OUTLIER More than 3/2 times of upper quartile
MAXIMUM Greatest value, excluding outliers
UPPER QUARTILE 25% of data greater than this value
MEDIAN 50% of data is greater than this value; middle of dataset
LOWER QUARTILE 25% of data less than this value
MINIMUM Least value, excluding outliers
OUTLIER Less than 3/2 times of lower quartile

Source: *Flowing Data*

Pandas LAB:

```
mjp= Series([5,4,3,2,1])# a simple series
print mjp
# A series is represented by index on the left and values on the right
print mjp.values
# similar to dictionary. ".values" command returns values in a series
print mjp.index # returns the index values of the series
jeeva = Series([5,4,3,2,1,-7,-29], index =['a','b','c','d','e','f','h']) # The index is specified
print jeeva # try jeeva.index and jeeva.values
print jeeva['a'] # selecting a particular value from a Series, by using index

jeeva['d'] = 9 # change the value of a particular element in series
print jeeva
jeeva[['a','b','c']] # select a group of values
print jeeva[jeeva>0] # returns only the positive values
print jeeva *2 # multiplies 2 to each element of a series

import numpy as np
np.mean(jeeva) # you can apply numpy functions to a Series

player_salary ={'Rooney': 50000, 'Messi': 75000, 'Ronaldo': 85000,
            'Fabregas':40000, 'Van persie': 67000}
new_player = Series(player_salary)# converting a dictionary to a series
print new_player # the series has keys of a dictionary

players =['Klose', 'Messi', 'Ronaldo', 'Van persie', 'Ballack']
player_1 =Series(player_salary, index= players)
print player_1 # I have changed the index of the Series.
# Since, no value was not found for Klose and Ballack,
# it appears as NAN
pd.isnull(player_1)#checks for Null values in player_1,
# pd denotes a pandas dataframe
pd.notnull(player_1)# Checks for null values that are not Null
player_1.name ='Bundesliga players' # name for the Series
player_1.index.name='Player names' #name of the index
player_1

player_1.index =['Neymar', 'Hulk', 'Pirlo', 'Buffon', 'Anderson']
# is used to alter the index of Series
player_1

states ={'State' :['Gujarat', 'Tamil Nadu', ' Andhra', 'Karnataka', 'Kerala'],
             'Population': [36, 44, 67,89,34],
             'Language' :['Gujarati', 'Tamil', 'Telugu', 'Kannada', 'Malayalam']}
india = DataFrame(states) # creating a data frame
india

DataFrame(states, columns=['State', 'Language', 'Population'])
```

```
# change the sequence of column index
new_farme = DataFrame(states, columns=['State', 'Language',
          'Population', 'Per Capita Income'],
          index =['a','b','c','d','e'])
#if you pass a column that isnt in states, it will appear
# with Na values
print new_farme.columns
print new_farme['State'] # retrieveing data like dictionary

new_farme.Population # like Series
new_farme.ix[3] # rows can be retrieved using .ic function
# here I have retrieved 3rd row
new_farme
new_farme['Per Capita Income'] = 99
# the empty per capita income column can be assigned a value
new_farme
new_farme['Per Capita Income'] = np.arange(5)
# assigning a value to the last column
new_farme

series = Series([44,33,22], index =['b','c','d'])
new_farme['Per Capita Income'] = series
#when assigning list or arrays to a column,
# the values lenght should match the length of the DataFrame
new_farme # again the missing values are displayed as NAN

new_farme['Development'] = new_farme.State == 'Gujarat'
# assigning a new column
print new_farme
del new_farme['Development'] # will delete the column 'Development'
new_farme

new_data ={'Modi': {2010: 72, 2012: 78, 2014 : 98},
          'Rahul': {2010: 55, 2012: 34, 2014: 22}}
elections = DataFrame(new_data)
print elections# the outer dict keys are columns
# and inner dict keys are rows
elections.T # transpose of a data frame
DataFrame(new_data, index =[2012, 2014, 2016])
ex= {'Gujarat':elections['Modi'][:-1], 'India': elections['Rahul'][:2]}
px =DataFrame(ex)
px
```