**Introduction to Pandas**

Pandas is a software library focused on fast and easy data manipulation and analysis in Python. In particular, it offers high-level data structures (like DataFrame and Series) and data methods for manipulating and visualizing numerical tables and time series data. It is built on top of NumPy and is highly optimized for performance (about 15x faster), with critical code paths written in Cython or C. The ndarray data structure and NumPy’s broadcasting abilities are heavily used.  
Pandas creator Wes McKinney starting developing the library in 2008 during his tenure at AQR, a quantitative investment management firm. He was motivated by a distinct set of data analysis requirements that were not well-addressed by any single tool at his disposal at the time.

**Pandas Features**

1. Data structures with labeled axes supporting automatic or explicit data alignment capable of handling both time-series and non-time-series data
2. Ability to add and remove columns on the fly
3. Flexible handling of missing data
4. SQL-like merge and other relational operations
5. Tools for reading and writing data between in-memory data structures and different file formats  (csv, xls, HDF5, SQL databases)
6. Reshaping and pivoting of data sets
7. Label-based slicing, fancy indexing, and subsetting of large data sets
8. Group by engine allowing split-apply-combine operations on data sets
9. Time series-functionality: date range generation and frequency conversion, moving window statistics, moving window linear regressions, date shifting and lagging
10. Hierarchical axis indexing to work with high-dimensional data in a lower-dimensional data structure

**Pandas Installation**

* **Make sure you have Python and pip**

All modern operating systems (Windows, Mac OS and Linux) come pre-installed with Python. To check whether your OS has Python, start up the command line tool (Command Prompt on Windows, Terminal on MacOS or Linux) and type python. If you see some welcome messages followed by the >>> sign, it means that you’ve successfully launched the Python interpreter.  
Next, ensure that you have pip installed by typing pip at the command line. If you do not see a help file, follow this guide: <https://packaging.python.org/en/latest/installing/> and get pip.

* **Install Pandas**

Simply go to your command line tool and type

pip install pandas

Ensure that the installation was successful by launching Python and write

import pandas as pd

print pd.\_\_version\_\_

* **Alternative – install Anaconda**

Anaconda is a free Python distribution that comes loaded with 330+ of the most popular Python packages (including pandas) for data science.  Download it here: <https://store.continuum.io/cshop/anaconda/>

**Reading Data into Pandas**

* **Reading a CSV file**

Reading a comma separated file is as simple as calling the read\_csv function. By default, the read\_csv function expects the column separator to be a comma, but you can change that using the sep parameter.

Syntax:

pd.read\_csv(filepath, sep=, header=, names=, skiprows=, na\_values= ... )

Help File: for a detailed explanation of all parameters, run

pd.read\_csv?

* **Reading an Excel File**

Pandas allows you to read from and write to Excel files, so you can easily read from Excel, write your code in Python, and then write back out to Excel –  no need for VBA. Reading Excel files requires the xlrd library. You can install it via pip using:

pip install xlrd

Syntax:

pd.read\_excel('my-excel.xlsx', 'sheet1')

* **Support for SQL Databases**

Pandas enables you to connect to external databases like Teradata or MS SQL database to read/write data. First, we create a connection to the database (supplying username, password and DB name if required) Then we pass a SQL query as a Python string through that connection. The query is run on the database and the table returned is imported into a pandas object.

Sample query:

import sqlite3

from pandas.io import sql

conn = sqlite3.connect('C:/Users/userX/Downloads/towed.db')

query = "SELECT \* FROM towed WHERE make = 'FORD';"

results = pd.read\_sql(query, con=conn)

print results.head()

* **Reading from the clipboard**

If you can copy a table (using CTRL + C) on the web or from an Excel sheet, you can quickly import it into your code using the pandas function *read\_clipboard()*.

**Pandas Data Structure**

Pandas has two **core** data structures – Series and DataFrame. Core operations using these structures include

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Create | Select | Insert | Update | View |
| Filter | Append | Sort | Join | Merge |
| Group | Summarize | Reshape | Map | Apply |

**Pandas Series**

A Series is a one-dimensional array-like object containing an array of data (of any NumPy data type) and an associated array of data labels, called its index.

* **Creating a Series**

It can be created in the same way as a NumPy array is created. Syntax:

Series(numpy-array, index = )

Example:

Series([21, 42, -31, 85], index=['d', 'b', 'a', 'c'])

If you do not specify an index for the data, a default one consisting of the integers 0 through N - 1 (where N is the length of the data) is created. Unlike the NumPy array, though, this index can be something other than integers.

We can also convert a Python dictionary to a Series, where the keys will become the index

d = {'Chicago': 1000, 'New York': 1300, 'Portland': 900, 'San Francisco': 1100, 'Austin': 450, 'Boston': None}

cities = pd.Series(d)

cities

* **Attributes of a Series**

These include *.values* and *.index*, using which we can get the array representation and index object of the Series respectively. We can assign a name to the Series using the *.index.name*

Example:

my\_series = Series(np.random.randn(5))

my\_series.values

my\_series.index

my\_series.index.name = 'row.names'

* **Subsetting a Series**

We can use the labels in the index, a list of labels, a Boolean vector or positional slicing for extracting elements from a Series. This is mostly similar to numPy array slicing except the returned values have the index associated.

Example:

my\_series = Series(np.arange(50, 71, 5), index = list('abcde'))

my\_series

my\_series['a'] # slice using index label

my\_series[['a', 'c', 'e']] # slicing using a list of labels

my\_series[0:3] # positional slicing

my\_series[my\_series > .60] # slicing using a boolean

* **Array Operations on a Series**

Array or Vectorized operations on a Series preserve the index-value links.

my\_series \* 2

np.sqrt(my\_series)

* **Check if an item exists in a Series**

This can be done using the in keyword. 'b' in my\_series

Missing values appear as NaN. Funtions isnull and notnull are used to detect missings.

index2 = ['a', 'd', 'e', 'f', 'g']

my\_series2 = my\_series[index2]; my\_series2

my\_series2.isnull() # or pd.isnull(my\_series2)

my\_series2.notnull() # or pd.notnull(my\_series2)

[](http://www.dezyre.com/data-science-in-python-online-training/36?from=tpanda1sgb)

**Pandas Dataframe**

It is 2-dimensional table-like data structure that

* Has both a row and column index for
  + Fast lookups
  + Data alignment and joins
* Is quite similar to the R data frame.
* Can contain columns of different data types
* Can be thought of a dict of Series objects.
* Has a number of associated methods that make commonplace tasks very simple

* **Creating a DataFrame**

Syntax:

DataFrame(data=, index=, columns=)

where ‘data’ can be a 2-d numpy array.

Example:

my\_df = DataFrame(np.arange(20, 32).reshape(3, 4),

columns = ['c1', 'c2', 'c3', 'c4'],

index = list('abc'))

One of the most common ways of creating a pandas DataFrame is from a Python dict of arrays or lists. The dict keys will be used as column names, and a string list can be supplied to act as the index.

Example:  
# creating DataFrame using a dict of equal length lists

my\_dict = {'ints': np.arange(5),

'floats': np.arange(0.1, 0.6, 0.1),

'strings': list('abcde')}

my\_df2 = DataFrame(my\_dict, index = list('vwxyz'))

* **Adding or Deleting Columns**

New columns can be created or derived from other existing columns. The syntax is similar to R DataFrames.

my\_df2['const'] = np.pi

my\_df2

del my\_df2['const']

We can also use the drop DataFrame method to remove one or more columns or rows

my\_df2.drop(['x', 'y']) # delete rows

my\_df2.drop(['const', 'ints'], axis=1) # delete columns

* **Attributes of the DataFrame**

Some of the most commonly used ones are –*index, columns, dtypes, info*

Code:

# Create a DataFrame using a dict of lists

data = {'country': ['Belgium', 'France', 'Germany', 'Netherlands', 'UK'],

'population': [11.3, 64.3, 81.3, 16.9, 64.9],

'area': [30510, 671308, 357050, 41526, 244820],

'capital': ['Brussels', 'Paris', 'Berlin', 'Amsterdam', 'London']}

countries = pd.DataFrame(data)

countries

Some DataFrame attributes

countries.index # Check row names

countries.columns # Check column names

countries.dtypes # Check data types

countries.info # Gives overview of the dataset

* **Some popular DataFrame methods**

***set\_index()***  
for setting an arbitrary *Index*.  
If we don't like what the index looks like, we can reset it and set one of our columns like this

countries = countries.set\_index('country')

***sort() or sort\_index()***  
for arranging the data. It has a simple syntax:

countries.sort('population', ascending=False)

***describe()***  
This is used to  compute summary statistics for each column numerical (default) column. Syntax:

countries.describe()

***plot()***  
This is used to quickly visualize the data in different ways. The available plotting types are: *‘line’ (default), ‘bar’, ‘barh’, ‘hist’, ‘box’ , ‘kde’, ‘area’, ‘pie’, ‘scatter’, ‘hexbin’*.

countries['population'].plot(kind='bar') # barcharts

countries.plot(kind='scatter', x='population', y='area') # scatterplots

* **Subsetting a DataFrame**

**Selecting 1 Column**

For a DataFrame, basic indexing selects the columns. An individual column can be retrieved as a Series using  
*df['col'] or df.col*  
This is especially helpful for creating boolean indexes.

Examples:

my\_df2['floats']

countries.area

**Selecting 2+ Columns**

Multiple columns are retrieved as a DataFrame using a list of column names  
*df[['col1', 'col2']]*

Examples:

my\_df2[['ints', 'strings']]

countries[['area', 'population']]

**Advanced Indexing using loc(), iloc()and ix( )**

For advanced indexing, the DataFrame attributes  
***.loc****(label indexing), and****.iloc****(integer indexing)*  
 are used.

**Using loc( )**  
Syntax: *df.loc[[indices], [colnames]]*  
Where [indices] could be specified as a list, splice (start : end) or a Boolean.

Examples  
# Using a row index (before comma) and a column name (after comma)

countries.loc['Germany', 'area']

# Using a row index splice and a column index splice

countries.loc['France':'Germany', :]

# Using a boolean for rows and a list of column names

countries.loc[countries['population']>5, ['capital', 'area']]

**Using iloc( )**  
Selecting by position with iloc works similar as indexing numpy arrays

Syntax:  
*df.iloc[row-positions, col-positions]*

Example:  
# Using splices for both rows and columns  
*countries.iloc[0:2, 1:3]*

**Using ix( )**  
This method is more general than the loc and iloc methods and can work both with labels and positions.  
Syntax: *df.ix[, ]*

Here,  
specify-cols could be done as a singular/list/splice of column name(s)  
Additionally we could even specify integer ranges (splices).  
specify-rows can be done using indices (if you want to subset rows by name) or by using integer splices (if you want to subset by position)

Examples:

**Subsetting Columns**

print my\_df2.ix[:, 'strings'] # select a column by name

print my\_df2.ix[:, ['strings', 'floats']] # select multiple columns by name

print my\_df2.ix[:, 0:2] # select columns by position

**Subsetting Rows**

print my\_df2.ix[0] # first row

print my\_df2.ix[2] # second row

print my\_df2.ix[0:2] # by position: returns the first 2 rows

print my\_df2.ix['x':'z'] # by index: returns the last three rows

**IMPORTANT NOTE**  
The columns returned when indexing a DataFrame is a **view** on the underlying data, **not a copy**. Thus, any in-place modifications to the Series will be reflected in the original DataFrame. The column can be explicitly copied using the Series copy method.

**Sorting Data**

* **Series**

To sort a series on its index, use:   *my\_series.sort\_index()*  
To sort a series on its values, use: *my\_series.order(ascending=)*

Examples:  
# Create a Series with explicit index

s9 = Series(np.random.randn(5), index=list('dcbae')); print s9

s9.sort\_index() # Sorting on the index

s9.order(ascending=False) # Sorting on the values

* **DataFrame**

For Reordering rows or columns use: *sort\_index()*  
For Sorting on column values use: *sort() or sort\_index(by=)*

Example:

d9 = DataFrame(np.random.randn(9).reshape(3,3),

index=list('cba'),

columns=list('prq'))

print d9

# without arguments, sort\_index() will sort the index (rows) of the DataFrame

d9.sort\_index()

# To sort column names

d9.sort\_index(axis=1)

# Sort the data by the values of a column

d9.sort\_index(by='p')

# Sort the data by the values of 2 columns

d9.sort\_index(by=['p', 'r'], ascending=False)

**Handling Missing Data**

Pandas treats the numpy NaN and the Python None as missing values.

* These can be detected in a Series or DataFrame using notnull() which returns a boolean.
* To filter out missing data from a Series, or to remove rows (default action) or columns with missing data in a DataFrame, we use dropna()
* Missing Value imputation is done using the fillna() method

Examples:

# Create a string Series and set some values to missing

s12 = Series(['abc', 'pqr', np.nan, 'xyz', np.nan, 'ijk', None])

# Detect missing values

s12.notnull()

# Replace missing values with a string

s12.fillna('--missing--')

# Create a numeric Series

s13 = Series(np.random.randn(8), index=list('abcdefgh'))

# set a few values to missing

s13[::2] = np.nan

# Fill with median

s13.fillna(s13.median())

# Note: We could use 0, or .mean() or some arbitrary method

**Descriptive Statistics**

* **Numeric Data**

Pandas objects have a set of common math/stat methods that extract

* a single summary value from a Series
* a Series of summary values by row/column from a DataFrame (along a specified axis)

These Methods include:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| count | sum | mean | median |  |
| min/max | skew | kurt | cumsum |  |

Note: When these methods are called on a DataFrame, they are applied over each row/column as specified and results collated into a Series. Missing values are ignored by default by these methods.  
Pass *skipna=False* to disable this.

Example:

d11 = DataFrame(np.random.randn(25).reshape(5,5),

index=list('abcde'),

columns=list('vwxyz')); print df8

# Getting colsums is as simple as calling the .sum() method of a DataFrame

d11.sum()

# For rowsums, pass axis=1 to the .sum() method

d11.sum(axis=1)

# Find the min/max for each column/row

d11.min(axis=0)            # by column

d11.min(axis=1)      # by row

# Find the location of the min value across rows

d11.idxmin()

# Calculate quantiles for each column

d11.quantile([0.2, 0.4, 0.6, 0.8])

**The describe() function – Numeric Data**

This function deserves a special mention because of its versatility. It works on numeric Series and produces the summary statistics including – min, max, count, mean, standard deviation, median and percentiles (25th, 75th). You can call describe on a Series (a column in a DataFrame) or an entire DataFrame (in which case it will produce results for each **numeric** column.).

# Summary Stats for each column!

d11.describe()

* **Categorical Data**

Pandas has some interesting methods for working on Categorical data. These include functions for getting unique values *(unique), frequency tables (value\_counts), membership (isin).*

Examples

# Getting distinct values in a Series

s12 = Series(list('the quick brown fox jumped over the lazy dog'))

s12.unique()

# Can also use: set(s12)

# Getting a Frequency Table

s12.value\_counts()

# isin returns a boolean indicating the position where a match occurred

colours = Series(['red', 'blue', 'white', 'green', 'black', 'white', None])

colours.isin(['white'])

**The describe() function – Categorical Data**

Calling the describe() function on categorical data returns summary information about the Series that includes the

- count of non-null values,  
- the number of unique values,  
- the mode of the data  
- the frequency of the mode

Example

colours.describe()