**STATISTICS USING PYTHON (16CS353)**

**UNIT-1**

**Syllabus:**

**Python and Statistics:** Why Statistics? Python Packages for Statistics, *First Python Programs,* Pandas: Data Structures for Statistics, Data Input:Input from Text Files:*Visual Inspection, Reading ASCII-Data into Python,* Input from MS Excel,Datatypes:Categorical, Numerical.

**Why Statistics?**

**Statistics** is a branch of [mathematics](https://en.wikipedia.org/wiki/Mathematics) dealing with [data](https://en.wikipedia.org/wiki/Data) collection, organization, analysis, interpretation and presentation.

In general, statistics will help to

* Clarify the question.
* Identify the variable and the measure of that variable that will answer that question.
* Determine the required sample size.
* Describe variation.
* Make quantitative statements about estimated parameters.
* Make predictions based on your data.

**Importance of Statistics**

* Statistics makes the work simple & provides a clear picture on the work we do on daily basis.
* The statistical methods helps us to research on different streams such as medicine, economics, business, social science and so on.
* Statistics provides us different types of organized data with the help of graphs, diagrams and charts.
* Statistics comes handy while we do critical analysis.

Statistics can be categorized into 2 types:  
  
1. **Descriptive Statistics:**

* It is used for summarizing observations etc.D[escriptive statistics](https://www.statisticshowto.datasciencecentral.com/probability-and-statistics/descriptive-statistics/) describes data (for example, a chart or graph).
* Descriptive statistics include measures of central tendency (mean, median, mode), measures of variation (standard deviation, variance), and relative position (quartiles, percentiles).

**2. Inferential Statistics:**

* **inferential statistics** allows you to make predictions (“inferences”) from that data.
* Inferential statistics are concerned with making inferences based on relations found in the sample, to relations in the population.
* We will start by considering the basic principles of significance testing: the sampling and test statistic distribution, p-value, significance level, power and type I and type II errors. Then we will consider a large number of statistical tests and techniques that help us make inferences for different types of data and different types of research designs. For each individual statistical test we will consider how it works, for what data and design it is appropriate and how results should be interpreted. You will also learn how to perform these tests using freely available software.

In the web, you will find very extensive information on statistics inEnglish at

• http://www.statsref.com/

• http://www.vassarstats.net/

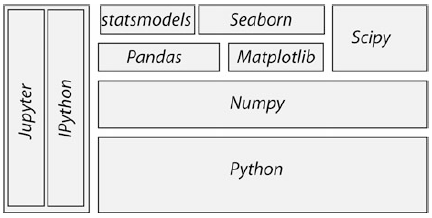
• http://www.biostathandbook.com/

• http://onlinestatbook.com/2/index.html

• <http://www.itl.nist.gov/div898/handbook/index.htm>

**Python Packages for Statistics**

The *Python* core distribution contains only the essential features of a general programming language. For example, it does not even contain a specialized modulefor working efficiently with vectors and matrices! These specialized modules are being developed by dedicated volunteers. The relationship of the most important*Python* packages for statistical applications is delineated in Fig. 2.1.

****

**Fig. 2.1** The structure of the most important *Python* packages for statistical applications

[**Pandas**](https://pandas.pydata.org/)

Pandas are a library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and [time series](https://en.wikipedia.org/wiki/Time_series). Pandas is [free software](https://en.wikipedia.org/wiki/Free_software) released under the three-clause BSD license

[**Statsmodels**](http://statsmodels.sourceforge.net/)

Statsmodels is a Python module that allows users to explore data, estimate statistical models, and perform statistical tests. An extensive list of descriptive statistics, statistical tests, plotting functions, and result statistics are available for different types of data and each estimator.

# **seaborn**

Seaborn is a Python data visualization library based on [matplotlib](https://matplotlib.org). It provides a high-level interface for drawing attractive and informative statistical graphics.

[**NumPy**](http://www.numpy.org/)

NumPy is an open source extension module for Python. The module NumPy provides fast precompiled functions for numerical routines.

It adds support to Python for large, multi-dimensional arrays and matrices. Besides that it supplies a large library of high-level mathematical functions to operate on these arrays

[**SciPy**](https://www.scipy.org/)

SciPy is widely used in scientific and technical computing. SciPy contains modules for optimization, linear algebra, integration, interpolation, special functions, FFT (Fast Fourier transform), signal and image processing, ODE (Ordinary Differential Equation) solvers and other tasks common in science and engineering.

[**matplotlib**](https://matplotlib.org/)

Matplotlib is a plotting library for NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like wxPython, Qt, or GTK+.

**IPython:**

IPython (Interactive Python) is an interactive shell for the Python programming language that offers enhanced Dynamic object introspection, additional shell syntax, tab completion and rich history.

**Jupyter Notebook**

The Jupyter Notebook is an incredibly powerful tool for interactively developing and presenting data science projects. A notebook integrates code and its output into a single document that combines visualizations, narrative text, mathematical equations, and other rich media.

[**scikit-learn**](http://scikit-learn.org/stable/)

scikit-learn is an open source library for the Python. It features various classification, regression and clustering algorithms including support vector machines, logistic regression, naive Bayes, random forests, gradient boosting, *k*-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.

[**Mlpy**](http://mlpy.sourceforge.net/)

Mlpy is a Python machine learning library built on top of NumPy/SciPy, the GNU Scientific Library. mlpy provides a wide range of  machine learning methods for supervised and unsupervised problem.mlpy is multi platform, it works with Python 2 and 3.

Most of these packages come either with the *WinPython* or *Anaconda* distributions, or can be installed easily using pip or conda.

* *WinPython* recommended for Windows users.

<https://winpython.github.io/>

* *Anaconda* by Continuum. For Windows, Mac, and Linux.

<https://store.continuum.io/cshop/anaconda/>

https://www.anconda.com/

The programs included in this book have been tested with Python 2.7.10 and 3.5.1, under Windows and Linux, using the following package versions:

* *ipython4.1.2* : : : For interactive work.
* *numpy1.11.0* : : : For working with vectors and arrays.
* *scipy0.17.1* : : : All the essential scientific algorithms, including those for basicstatistics.
* *matplotlib 1.5.1* : : : The de-facto standard module for plotting and visualization.
* *pandas 0.18.0* : : : Adds *DataFrames*(imagine powerful spreadsheets) to *Python*.
* *patsy 0.4.1* : : : For working with statistical formulas.
* *statsmodels0.8.0* : : : For statistical modeling and advanced analysis.
* *seaborn 0.7.0* : : : For visualization of statistical data.

In addition to these fairly general packages, some specialized packages have also been used in the examples accompanying this book:

* *xlrd0.9.4* : : : For reading and writing MS Excel files.
* *PyMC2.3.6* : : : For Bayesian statistics, including Markov chain Monte Carlosimulations.
* *scikit-learn 0.17.1* : : : For machine learning.
* *scikits.bootstrap 0.3.2* : : : Provides bootstrap confidence interval algorithms forscipy.
* *lifelines 0.9.1.0* : : : Survival analysis in *Python*.
* *rpy2 2.7.4* : : : Provides a wrapper for *R*-functions in *Python*.

To get *PyMC* to run, you may need to install a C-compiler. On my *Windows* platform, I installed *Visual Studio 15*,and set the environment variable SET VS90COMNTOOLS=%VS14COMNTOOLS%.

To use *R*-function from within *Python*, you also have to install *R*. Like *Python*, *R* is available for free, and can be downloaded from the *Comprehensive R ArchiveNetwork* (<http://cran.r-project.org/>).

***First Python Programs***

**a) HelloWorld**

Python Shell

*Python* is an interpreted language. The simplest way to start *Python* is to typepython on the command line. (When I say *command line* I refer in *Windows* tothe command shell started with cmd, and in *Linux* or *Mac OS X* to the terminal.)Then you can already start to execute *Python* commands, e.g., the command to print“HelloWorld” to the screen: print('Hello World'). On myWindows computer, this results in

Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:54:25) [

MSC v.1900 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for moreinformation.

>>>print('Hello World')

Hello World

>>>

However, I never use the basic *Python* shell any more, but always start out withthe *IPython/Jupyter qtconsole* described in more detail in Sect. 2.3. The *Qt console*is an interactive programming environment which offers a number of advantages.For example, when you type print( in the *Qt console*, you immediately seeinformation about the possible input arguments for the command print.

Python Modules

Often we want to store our commands in a file for later reuse. *Python* files have the extension .py, and are referred to as *Python modules*. Let us create a new file with the name helloWorld.py, containing the line print('Hello World')

This file can now be executed by typing python helloWorld.py on thecommand line.In *Windows* you can actually run the file by double-clicking it, or by simplytyping helloWorld.py if the extension .py is associated with the *Python* programinstalled on your computer. In *Linux* and *Mac OS X*the procedure is slightly moreinvolved. There, the file needs to contain an additional first line specifying the pathto the *Python* installation.

#! \usr\bin\python

print('Hello World')

On these two systems, you also have to make the file executable, by typingchmod +x helloWorld.py, before you can run it with helloWorld.py.

**b) SquareMe**

To increase the level of complexity, let us write a *Python* module which prints outthe square of the numbers from zero to five. We call the file squareMe.py, and itcontains the following lines

**Listing 2.1** squareMe.py

# This file shows the square of the numbers from 0 to 5.

def squared(x):

return x\*\*2

for ii in range(6):

print(ii, squared(ii))

print('Done')

Let me explain what happens in this file, line-by-line:

**1** The first line starts with “#”, indicating a comment-line.

**3–4** These two lines define the function *squared*, which takes the variable *x* asinput, and returns the square (x\*\*2) of this variable.

**Note:** The range of the function is defined by the indentation! This is afeature loved by many *Python* programmers, but often found confusing bynewcomers. Here the last indented line is *line 4*, which ends the functiondefinition.

**6–7** Here the program loops over the first 6 numbers. Also the range of the forloopis defined by the indentation of the code.

In *line 7*, each number and its corresponding square are printed to the output.

**9** This command is not indented, and therefore is executed after the for-loophas ended.

**Notes**

* Since *Python* starts at 0, the loop in *line 6* includes the numbers from 0 to 5.
* In contrast to some other languages *Python* distinguishes the syntax for function calls from the syntax for addressing elements of an array etc: function calls, as in *line 7*, are indicated with round brackets ( ... ); and individual elements of arrays or vectors are addressed by square brackets [ ... ].

**Pandas: Data Structures for Statistics**

*pandas* is a widely used *Python* package which has been contributed by WesMcKinney. It provides data structures suitable for statistical analysis, and adds functions that facilitate data input, data organization, and data manipulation. It is common to import pandas as pd, which reduces the typing a bit (<http://pandas>.pydata.org/).

A good introduction to pandas has been written by Olson (2012).

Pandas in Python deal with three data structures namely

* Series
* Data Frame
* Panel

These data structures are built on top of Numpy array, which means they are fast.

|  |  |  |
| --- | --- | --- |
| **Data Structure** | **Dimensions** | **Description** |
| Series | 1 | 1D labeled homogeneous array, size immutable. |
| Data Frames | 2 | General 2D labeled, size-mutable tabular structure with potentially heterogeneously typed columns. |
| Panel | 3 | General 3D labeled, size-mutable array. |

# Series

Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.). The axis labels are collectively called index.

A pandas Series can be created using the following constructor –

**pandas. Series( data, index, dtype, copy)**

The parameters of the constructor are as follows −

|  |  |
| --- | --- |
| **S.No** | **Parameter & Description** |
| 1 | **data**  data takes various forms like ndarray, list, constants |
| 2 | **index**  Index values must be unique and hashable, same length as data. Default **np.arrange(n)** if no index is passed. |
| 3 | **Dtype**  dtype is for data type. If None, data type will be inferred |
| 4 | **Copy**  Copy data. Default False |

A series can be created using various inputs like −

* Array
* Dict
* Scalar value or constant

## Create an Empty Series

A basic series, which can be created is an Empty Series.

### Example

#import the pandas library and aliasing as pd

import pandas as pd

s = pd.Series()

print s

Its **output** is as follows −

Series([], dtype: float64)

## Create a Series from ndarray

If data is an ndarray, then index passed must be of the same length. If no index is passed, then by default index will be **range(n)** where **n** is array length, i.e., [0,1,2,3…. **range(len(array))-1].**

### Example 1

#import the pandas library and aliasing as pd

import pandas as pd

import numpy as np

data = np.array(['a','b','c','d'])

s = pd.Series(data)

print s

Its **output** is as follows −

0 a

1 b

2 c

3 d

dtype: object

We did not pass any index, so by default, it assigned the indexes ranging from 0 to **len(data)-1**, i.e., 0 to 3.

You can get the array representation and index object of the Series via its values and index attributes, respectively:

s.values

array(['a', 'b', 'c', 'd'], dtype=object)

s.index

RangeIndex(start=0, stop=4, step=1)

### Example 2

#import the pandas library and aliasing as pd

import pandas as pd

import numpy as np

data = np.array([1,2,3,4,5])

s = pd.Series(data,index=['a','b','c','d','e'])

print(s)

Its **output** is as follows −

a 1

b 2

c 3

d 4

e 5

dtype: int32

We passed the index values here. Now we can see the customized indexed values in the output.

Compared with a regular NumPy array, you can use values in the index when selecting single values or a set of values:

s['a']

1

s[['a','b','d']]

a 1

b 2

d 4

dtype: int32

NumPy array operations, such as filtering with a boolean array, scalar multiplication, or applying math functions, will preserve the index-value link:

s[s>2]

c 3

d 4

e 5

dtype: int32

s\*2

a 2

b 4

c 6

d 8

e 10

dtype: int32

np.exp(s)

a 2.718282

b 7.389056

c 20.085537

d 54.598150

e 148.413159

dtype: float64

Another way to think about a Series is as a fixed-length, ordered dict, as it is a mapping of index values to data values. It can be substituted into many functions that expect a dict:

'b' in s

True

'k' in s

False

## Create a Series from dict

A **dict** can be passed as input and if no index is specified, then the dictionary keys are taken in a sorted order to construct index. If **index** is passed, the values in data corresponding to the labels in the index will be pulled out.

### Example 1

import pandas as pd

data = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

s = pd.Series(data)

print(s)

Its **output** is as follows −

Ohio 35000

Texas 71000

Oregon 16000

Utah 5000

dtype: int64

When only passing a dict, the index in the resulting Series will have the dict’s keys in sorted order.

states = ['California', 'Ohio', 'Oregon', 'Texas']

s1= pd.Series(data, index=states)

print(s1)

alifornia NaN

Ohio 35000.0

Oregon 16000.0

Texas 71000.0

dtype: float64

In this case, 3 values found in sdata were placed in the appropriate locations, but since no value for 'California' was found, it appears as NaN (not a number) which is con- sidered in pandas to mark missing or NA values. I will use the terms “missing” or “NA” to refer to missing data. The isnull and notnull functions in pandas should be used to detect missing data:

pd.isnull(s)

California True

Ohio False

Oregon False

Texas False

dtype: bool

pd.notnull(s)

California False

Ohio True

Oregon True

Texas True

dtype: bool

**A critical Series feature for many applications is that it automatically aligns differently- indexed data in arithmetic operations:**

**s+s1**

California NaN

Ohio 70000.0

Oregon 32000.0

Texas 142000.0

dtype: float64

**Observe** − Dictionary keys are used to construct index.

### Example 2

#import the pandas library and aliasing as pd

import pandas as pd

import numpy as np

data = {'a' : 0., 'b' : 1., 'c' : 2.}

s = pd.Series(data,index=['b','c','d','a'])

print s

Its **output** is as follows −

b 1.0

c 2.0

d NaN

a 0.0

dtype: float64

**Observe** − Index order is persisted and the missing element is filled with NaN (Not a Number).

## Create a Series from Scalar

If data is a scalar value, an index must be provided. The value will be repeated to match the length of **index**

#import the pandas library and aliasing as pd

import pandas as pd

import numpy as np

s = pd.Series(5, index=[0, 1, 2, 3])

print s

Its **output** is as follows −

0 5

1 5

2 5

3 5

dtype: int64

## Accessing Data from Series with Position

Data in the series can be accessed similar to that in an **ndarray.**

### Example 1

Retrieve the first element. As we already know, the counting starts from zero for the array, which means the first element is stored at zeroth position and so on.

import pandas as pd

s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve the first element

print s[0]

Its **output** is as follows −

1

### Example 2

Retrieve the first three elements in the Series. If a : is inserted in front of it, all items from that index onwards will be extracted. If two parameters (with : between them) is used, items between the two indexes (not including the stop index)

import pandas as pd

s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve the first three element

print s[:3]

Its **output** is as follows −

a 1

b 2

c 3

dtype: int64

### Example 3

Retrieve the last three elements.

import pandas as pd

s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve the last three element

print s[-3:]

Its **output** is as follows −

c 3

d 4

e 5

dtype: int64

## Retrieve Data Using Label (Index)

A Series is like a fixed-size **dict** in that you can get and set values by index label.

### Example 1

Retrieve a single element using index label value.

import pandas as pd

s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve a single element

print s['a']

Its **output** is as follows −

1

### Example 2

Retrieve multiple elements using a list of index label values.

import pandas as pd

s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve multiple elements

print s[['a','c','d']]

Its **output** is as follows −

a 1

c 3

d 4

dtype: int64

### Example 3

If a label is not contained, an exception is raised.

import pandas as pd

s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve multiple elements

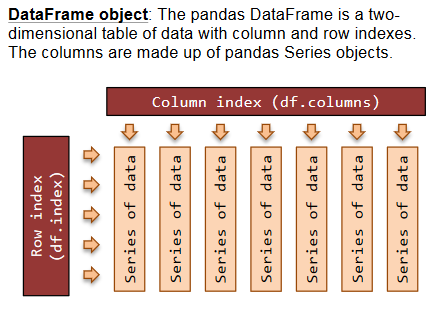
print s['f']

Its **output** is as follows −

…

KeyError: 'f'

# **DataFrame**



A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.

### Features of DataFrame

* Potentially columns are of different types
* Size – Mutable
* Labeled axes (rows and columns)
* Can Perform Arithmetic operations on rows and columns

A pandas DataFrame can be created using the following constructor −

**pandas.DataFrame( data, index, columns, dtype, copy)**

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| **data** | data takes various forms like ndarray, series, map, lists, dict, constants and also another DataFrame. |
| **index** | For the row labels, the Index to be used for the resulting frame is Optional Default np.arrange(n) if no index is passed. |
| **columns** | For column labels, the optional default syntax is - np.arange(n). This is only true if no index is passed. |
| **dtype** | Data type of each column. |
| **copy** | This command (or whatever it is) is used for copying of data, if the default is False. |

**Create DataFrame**

A pandas DataFrame can be created using various inputs like −

* Lists
* dict
* Series
* Numpy ndarrays
* Another DataFrame

## Create an Empty DataFrame

A basic DataFrame, which can be created is an Empty Dataframe.

### Example

#import the pandas library and aliasing as pd

import pandas as pd

df = pd.DataFrame()

print df

output:

Empty DataFrame

Columns: []

Index: []

## Create a DataFrame from Lists

The DataFrame can be created using a single list or a list of lists.

### Example 1

import pandas as pd

data = [1,2,3,4,5]

df = pd.DataFrame(data)

print(df)

Its **output** is as follows −

0

0 1

1 2

2 3

3 4

4 5

### Example 2

import pandas as pd

data = [['Alex',10],['Bob',12],['Clarke',13]]

df = pd.DataFrame(data,columns=['Name','Age'])

print(df)

Its **output** is as follows −

Name Age

0 Alex 10

1 Bob 12

2 Clarke 13

### Example 3

import pandas as pd

data = [['Alex',10],['Bob',12],['Clarke',13]]

df = pd.DataFrame(data, columns=['Name','Age'], dtype=float)

print(df)

Its **output** is as follows −

Name Age

0 Alex 10.0

1 Bob 12.0

2 Clarke 13.0

**Note** − Observe, the **dtype** parameter changes the type of Age column to floating point.

## Create a DataFrame from Dict of ndarrays / Lists

All the **ndarrays** must be of same length. If index is passed, then the length of the index should equal to the length of the arrays.

If no index is passed, then by default, index will be range(n), where **n** is the array length.

### Example 1

import pandas as pd

data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}

df = pd.DataFrame(data)

print(df)

Its **output** is as follows −

Name Age

0 Tom 28

1 Jack 34

2 Steve 29

3 Ricky 42

**Note** − Observe the values 0,1,2,3. They are the default index assigned to each row using the function range(n).

### Example 2

Let us now create an indexed DataFrame using arrays.

import pandas as pd

data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}

df = pd.DataFrame(data, index=['rank1','rank2','rank3','rank4'])

print(df)

Its **output** is as follows −

Name Age

rank1 Tom 28

rank2 Jack 34

rank3 Steve 29

rank4 Ricky 42

**Note** − Observe, the **index** parameter assigns an index to each row.

## Create a DataFrame from List of Dicts

List of Dictionaries can be passed as input data to create a DataFrame. The dictionary keys are by default taken as column names.

### Example 1

The following example shows how to create a DataFrame by passing a list of dictionaries.

import pandas as pd

data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]

df = pd.DataFrame(data)

print(df)

Its **output** is as follows −

a b c

0 1 2 NaN

1 5 10 20.0

**Note** − Observe, NaN (Not a Number) is appended in missing areas.

### Example 2

The following example shows how to create a DataFrame by passing a list of dictionaries and the row indices.

import pandas as pd

data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]

df = pd.DataFrame(data, index=['first', 'second'])

print df

Its **output** is as follows −

a b c

first 1 2 NaN

second 5 10 20.0

### Example 3

The following example shows how to create a DataFrame with a list of dictionaries, row indices, and column indices.

import pandas as pd

data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]

#With two column indices, values same as dictionary keys

df1 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b'])

#With two column indices with one index with other name

df2 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b1'])

print df1

print df2

Its **output** is as follows −

#df1 output

a b

first 1 2

second 5 10

#df2 output

a b1

first 1 NaN

second 5 NaN

**Note** − Observe, df2 DataFrame is created with a column index other than the dictionary key; thus, appended the NaN’s in place. Whereas, df1 is created with column indices same as dictionary keys, so NaN’s appended.

## Create a DataFrame from Dict of Series

Dictionary of Series can be passed to form a DataFrame. The resultant index is the union of all the series indexes passed.

### Example

import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),

'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)

print(df)

Its **output** is as follows −

one two

a 1.0 1

b 2.0 2

c 3.0 3

d NaN 4

**Note** − Observe, for the series one, there is no label **‘d’** passed, but in the result, for the **d** label, NaN is appended with NaN.

Let us now understand **column selection, addition**, and **deletion** through examples.

## Column Selection

We will understand this by selecting a column from the DataFrame.

### Example

import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),

'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)

print df ['one']

Its **output** is as follows −

a 1.0

b 2.0

c 3.0

d NaN

Name: one, dtype: float64

## Column Addition

We will understand this by adding a new column to an existing data frame.

### Example

import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),

'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)

# Adding a new column to an existing DataFrame object with column label by passing new series

print ("Adding a new column by passing as Series:")

df['three']=pd.Series([10,20,30],index=['a','b','c'])

print(df)

print ("Adding a new column using the existing columns in DataFrame:")

df['four']=df['one']+df['three']

print df

Its **output** is as follows −

Adding a new column by passing as Series:

one two three

a 1.0 1 10.0

b 2.0 2 20.0

c 3.0 3 30.0

d NaN 4 NaN

Adding a new column using the existing columns in DataFrame:

one two three four

a 1.0 1 10.0 11.0

b 2.0 2 20.0 22.0

c 3.0 3 30.0 33.0

d NaN 4 NaN NaN

## Column Deletion

Columns can be deleted or popped; let us take an example to understand how.

### Example

# Using the previous DataFrame, we will delete a column

# using del function

import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),

'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd']),

'three' : pd.Series([10,20,30], index=['a','b','c'])}

df = pd.DataFrame(d)

print ("Our dataframe is:")

print(df)

# using del function

print ("Deleting the first column using DEL function:")

**del df['one']**

print df

# using pop function

print ("Deleting another column using POP function:")

df.pop('two')

print df

Its **output** is as follows −

Our dataframe is:

one three two

a 1.0 10.0 1

b 2.0 20.0 2

c 3.0 30.0 3

d NaN NaN 4

Deleting the first column using DEL function:

three two

a 10.0 1

b 20.0 2

c 30.0 3

d NaN 4

Deleting another column using POP function:

three

a 10.0

b 20.0

c 30.0

d NaN

## Row Selection, Addition, and Deletion

We will now understand row selection, addition and deletion through examples. Let us begin with the concept of selection.

### Selection by Label

Rows can be selected by passing row label to a **loc** function.

import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),

'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)

print(df.loc['b'])

Its **output** is as follows −

one 2.0

two 2.0

Name: b, dtype: float64

The result is a series with labels as column names of the DataFrame. And, the Name of the series is the label with which it is retrieved.

### Selection by integer location

Rows can be selected by passing integer location to an **iloc** function.

import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),

'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)

print(df.iloc[2])

Its **output** is as follows −

one 3.0

two 3.0

Name: c, dtype: float64

### Slice Rows

Multiple rows can be selected using ‘ : ’ operator.

import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),

'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)

print(df[2:4])

Its **output** is as follows −

one two

c 3.0 3

d NaN 4

### Addition of Rows

Add new rows to a DataFrame using the **append** function. This function will append the rows at the end.

import pandas as pd

df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])

df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])

df = df.append(df2)

print df

Its **output** is as follows −

a b

0 1 2

1 3 4

0 5 6

1 7 8

### Deletion of Rows

Use index label to delete or drop rows from a DataFrame. If label is duplicated, then multiple rows will be dropped.

If you observe, in the above example, the labels are duplicate. Let us drop a label and will see how many rows will get dropped.

import pandas as pd

df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])

df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])

df = df.append(df2)

# Drop rows with label 0

df = df.drop(0)

print df

Its **output** is as follows −

a b

1 3 4

1 7 8

In the above example, two rows were dropped because those two contain the same label 0.

# **Pandas - Panel**

A **panel** is a 3D container of data.

A Panel can be created using the following constructor −

pandas.Panel(data, items, major\_axis, minor\_axis, dtype, copy)

The parameters of the constructor are as follows −

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| data | Data takes various forms like ndarray, series, map, lists, dict, constants and also another DataFrame |
| items | axis=0, each item corresponds to a DataFrame contained inside. |
| major\_axis | axis=1, it is the index (rows) of each of the DataFrames. |
| minor\_axis | axis=2, it is the columns of each of the DataFrames. |
| dtype | Data type of each column |
| copy | Copy data. Default, **false** |

## Create Panel

A Panel can be created using multiple ways like −

* From ndarrays
* From dict of DataFrames

### From 3D ndarray

# creating an empty panel

import pandas as pd

import numpy as np

data = np.random.rand(2,4,5)

p = pd.Panel(data)

print p

Its **output** is as follows −

<class 'pandas.core.panel.Panel'>

Dimensions: 2 (items) x 4 (major\_axis) x 5 (minor\_axis)

Items axis: 0 to 1

Major\_axis axis: 0 to 3

Minor\_axis axis: 0 to 4

**Note** − Observe the dimensions of the empty panel and the above panel, all the objects are different.

### From dict of DataFrame Objects

#creating an empty panel

import pandas as pd

import numpy as np

data = {'Item1' : pd.DataFrame(np.random.randn(4, 3)),

'Item2' : pd.DataFrame(np.random.randn(4, 2))}

p = pd.Panel(data)

print p

Its **output** is as follows −

<class 'pandas.core.panel.Panel'>

Dimensions: 2 (items) x 4 (major\_axis) x 5 (minor\_axis)

Items axis: 0 to 1

Major\_axis axis: 0 to 3

Minor\_axis axis: 0 to 4

### Create an Empty Panel

An empty panel can be created using the Panel constructor as follows −

#creating an empty panel

import pandas as pd

p = pd.Panel()

print p

Its **output** is as follows −

<class 'pandas.core.panel.Panel'>

Dimensions: 0 (items) x 0 (major\_axis) x 0 (minor\_axis)

Items axis: None

Major\_axis axis: None

Minor\_axis axis: None

## Selecting the Data from Panel

Select the data from the panel using −

* Items
* Major\_axis
* Minor\_axis

### Using Items

# creating an empty panel

import pandas as pd

import numpy as np

data = {'Item1' : pd.DataFrame(np.random.randn(4, 3)),

'Item2' : pd.DataFrame(np.random.randn(4, 2))}

p = pd.Panel(data)

print p['Item1']

Its **output** is as follows −

0 1 2

0 0.488224 -0.128637 0.930817

1 0.417497 0.896681 0.576657

2 -2.775266 0.571668 0.290082

3 -0.400538 -0.144234 1.110535

We have two items, and we retrieved item1. The result is a DataFrame with 4 rows and 3 columns, which are the **Major\_axis** and **Minor\_axis** dimensions.

### Using major\_axis

Data can be accessed using the method **panel.major\_axis(index)**.

# creating an empty panel

import pandas as pd

import numpy as np

data = {'Item1' : pd.DataFrame(np.random.randn(4, 3)),

'Item2' : pd.DataFrame(np.random.randn(4, 2))}

p = pd.Panel(data)

print p.major\_xs(1)

Its **output** is as follows −

Item1 Item2

0 0.417497 0.748412

1 0.896681 -0.557322

2 0.576657 NaN

### Using minor\_axis

Data can be accessed using the method **panel.minor\_axis(index).**

# creating an empty panel

import pandas as pd

import numpy as np

data = {'Item1' : pd.DataFrame(np.random.randn(4, 3)),

'Item2' : pd.DataFrame(np.random.randn(4, 2))}

p = pd.Panel(data)

print p.minor\_xs(1)

Its **output** is as follows −

Item1 Item2

0 -0.128637 -1.047032

1 0.896681 -0.557322

2 0.571668 0.431953

3 -0.144234 1.302466

***Data Handling***

1. **Common Procedures**

In statistical data analysis, labeled data structures have turned out to be immenselyuseful. To handle labeled data in *Python*, *pandas* introduces the so-called*DataFrame*objects. A DataFrame is a two-dimensional labeled data structurewith columns of potentially different types. You can think of it like a spreadsheet orSQL table. DataFrames are the most commonly used pandas objects

Let me start with a specific example, by creating a DataFrame with threecolumns, called “Time,” “x,” and “y”:

import numpy as np

import pandas as pd

t = np.arange(0,10,0.1)

x = np.sin(t)

y = np.cos(t)

df = pd.DataFrame({'Time':t, 'x':x, 'y':y})

In *pandas*, rows are addressed through indices and columns through their name.

To address the first column only, you have two options:

df.Time

df['Time']

If you want to extract two columns at the same time, ask for several variables ina list:

data = df[['Time', 'y']]

To display the first or last rows, use

data.head()

data.tail()

To extract the six rows from 5 to 10, use

data[4:10]as 10 \_ 4 D 6. (I know, the array indexing takes some time to get used to. Justkeep in mind that *Python* addresses the *locations between* entries, not the entries,and that it starts at 0!)

The handling of DataFrames is somewhat different from the handling of *numpy*arrays. For example, (numbered) rows and (labeled) columns can be addressedsimultaneously as follows:

df[['Time', 'y']][4:10]

You can also apply the standard row/column notation, by using the method iloc:

df.iloc[4:10, [0,2]]

Finally, sometimes you want to have direct access to the data, not to theDataFrame. You can do this withdata.valueswhich returns a *numpy*array.

1. **Notes on Data Selection**

While *pandas*’ DataFrames are similar tonumpy arrays, their philosophy isdifferent, and I have wasted a lot of nerves addressing data correctly. Therefore Iwant to explicitly point out the differences here:

**numpy**handles “rows” first. E.g., data[0] is the first row of an array**pandas** starts with the columns. E.g., df['values'][0] is the first element ofthe column 'values'.

If a DataFrame has labeled rows, you can extract for example the row “rowlabel”with df.loc['rowlabel']. If you want to address a row by its number, e.g., rownumber “15,” use df.iloc[15]. You can also use iloc to address “rows/columns,”e.g., df.iloc[2:4,3].

Slicing of rows also works, e.g., df[0:5] for the first 5 (!) rows. A sometimesconfusing convention is that if you want to slice out a single row, e.g., row “5,” youhave to use df[5:6]. If you use df[5] alone, you get an error!

***2.5.2 Grouping***

*pandas* offers powerful functions to handle missing data which are often replaced by *nan’s (“Not-A-Number”)*. It also allows more complex types of data manipulation like pivoting. For example, you can use data-frames to efficiently group objects, and do a statistical evaluation of each group. The following data are simulated (but realistic) data of a survey on how many hours a day people watch the TV, grouped into “m”ale and “f”emale responses:

import pandas as pd

import matplotlib.pyplot as plt

data = pd.DataFrame({

'Gender': ['f', 'f', 'm', 'f', 'm',

'm', 'f', 'm', 'f', 'm', 'm'],

'TV': [3.4, 3.5, 2.6, 4.7, 4.1, 4.1,

5.1, 3.9, 3.7, 2.1, 4.3]

})

#--------------------------------------------

# Group the data

grouped = data.groupby('Gender')

# Do some overview statistics

print(grouped.describe())

# Plot the data:

grouped.boxplot()

plt.show()

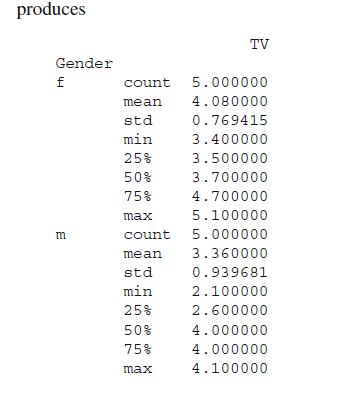
#--------------------------------------------

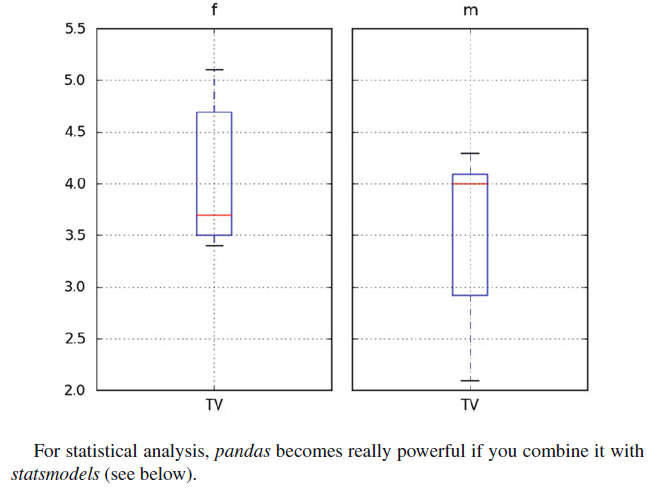
# Get the groups as DataFrames

df\_female = grouped.get\_group('f')

# Get the corresponding numpy-array

values\_female = grouped.get\_group('f').values



****

**Data Input**

Data input tells that how to read data into *Python*. It may besurprising, but reading data into the system in the correct format and checking for erroneous or missing entries is often one of the most time consuming parts of the data analysis.

Data input can be complicated by a number of problems, like different separators between data entries (such as spaces and/or tabulators), or empty lines at the end of the file. In addition, data may have been saved in different formats, such as MS Excel, *Matlab*, HDF5 (which also includes the *Matlab*-format), or in databases. Understandably, we cannot cover all possible input options. But I will try to give an overview of where and how to start with data input.

**3.1 Inputs from Text Files**

***3.1.1 Visual Inspection***

When the data is available as ASCII-files, you should always start out with a visual inspection of the data! In particular, you should check

* + Do the data have a header and/or footer?
  + Are there empty lines at the end of the file?
  + Are there white-spaces before the first number, or at the end of each line? (The latter is a lot harder to see.)
  + Are the data separated by tabulators, and/or by spaces? (Tip: you should use a text-editor which can visualize tabs, spaces, and end-of-line (EOL) characters.)

***3.1.2 Reading ASCII-Data into Python***

In *Python*, I strongly suggest that you start out reading in and inspecting your data in the *Jupyter QtConsole*or in an *Jupyter Notebook*. It allows you to move around much more easily, try things out, and quickly get feedback on how successful your commands have been. When you have your command syntax worked out, you can obtain the command history with %history, copy it into your favorite IDE, and turn it into a program.

While the a numpy command np.loadtxt allows to read in simply formatted text data, most of the time I go straight to *pandas*, as it provides significantly more powerful tools for data-entry. A typical workflow can contain the following steps:

* + Changing to the folder where the data are stored.
  + Listing the items in that folder.
  + Selecting one of these items, and reading in the corresponding data.
  + Checking if the data have been read in completely, and in the correct format.

These steps can be implemented in *IPython* with the following commands:

In [1]: import pandas as pd

In [2]: cd 'C:\Data\storage'

In [3]: pwd # Check if you were successful

In [4]: ls # List the files in that directory

In [5]: inFile = 'data.txt'

In [6]: df = pd.read\_csv(inFile)

In [7]: df.head() # Check if first line is ok

In [8]: df.tail() # Check the last line

After “In [7]” I often have to adjust the options of pd.read\_csv, to read in all the data correctly. Make sure you check that the number of column headers is equalto the number of columns that you expect. It can happen that everything gets read in—but into one large single column!

Using jupyter notebook

import pandas as pd

df=pd.read\_csv('f:/python1/data.txt')

print(df)

**Simple Text-Files**

For example, a file data.txt with the following content

1, 1.3, 0.6

2, 2.1, 0.7

3, 4.8, 0.8

4, 3.3, 0.9

The above data can be read into python in different ways.

# numpy. Loadtxt()

* NumpPy’s loadtxt function allows us to read numerical data file in text format in to Python. To load a CSV (Comma Separated Values) file, we specify delimitter to “,”.
* Load data from a text file.
* Each row in the text file must have the same number of values.

**Syntax:**

numpy.loadtxt(fname, dtype=<class 'float'>, comments='#', delimiter=None, converters=None, skiprows=0, usecols=None, unpack=False, ndmin=0, encoding='bytes')

**fname** : File, filename,

**dtype** : data-type, optional

**comments** : The characters or list of characters used to indicate the start of a comment. None implies no comments.

**delimiter** : str, optional

The string used to separate values.

**converters** : dict, optional

A dictionary mapping column number to a function that will parse the column string into the desired value. E.g., if column 0 is a date string: converters = {0: datestr2num}. Default: None.

**skiprows** : Skip the first skiprows lines; default: 0.

**usecols** : int or sequence, optional

Which columns to read, with 0 being the first. For example, usecols = (1,4,5) will extract the 2nd, 5th and 6th columns. The default, None, results in all columns being read.

**unpack** : bool, optional

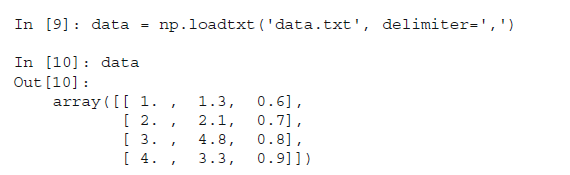
If True, the returned array is transposed, so that arguments may be unpacked using x, y, z = loadtxt(...). When used with a structured data-type, arrays are returned for each field. Default is False.

**ndmin** : int, optional

The returned array will have at least ndmin dimensions. Otherwise mono-dimensional axes will be squeezed. Legal values: 0 (default), 1 or 2.

**encoding** : str, optional

Encoding used to decode the input file. Does not apply to input streams. The special value ‘bytes’ enables backward compatibility workarounds that ensures you receive byte arrays as results if possible and passes ‘latin1’ encoded strings to converters. Override this value to receive unicode arrays and pass strings as input to converters. If set to None the system default is used. The default value is ‘bytes’.

****

where data is a *numpy array*. Without the flag delimiter=',', the function *np.loadtxt* crashes.

An alternative way to read in these data is with pandas read\_csv() function.

**read\_csv( ):**

* **read\_csv** is an important pandas function to read csv files and do operations on it.
* **read.csv** reads data from the csv files and creates a DataFrame object.

pandas.read\_csv(filepath\_or\_buffer, sep=',', delimiter=None, header='infer', names=None,

index\_col=None, usecols=None, squeeze=False,…… skiprows=None … … … … … …)

**parameters:**

**filepath\_or\_buffer** - URL or Dir location of file

**sep** - Stands for seperator, default is ‘, ‘ as in csv(comma seperated values)

**delimiter**- Alternative argument name for sep.

**header**- Headers refer to the column names of your dataset.

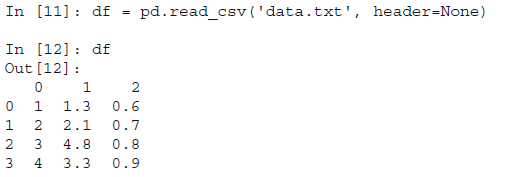
**names**- List of column names to use. If file contains no header row, then you should explicitly pass **header**=None. Duplicates in this list will cause a UserWarning to be issued.

**index\_col-** Makes passed column as index instead of 0, 1, 2, 3…r

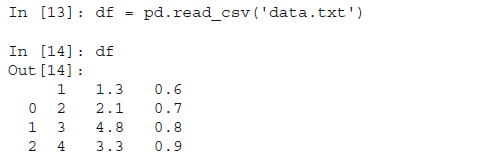
**usecols** - Only uses the passed col[string list] to make data frame

**squeeze** - If true and only one column is passed, returns pandas series

**skiprows** - Skips passed rows in new data frame(allows you to specify the number of lines to skip at the start of the file)

****

where df is a *pandas DataFrame*. Without the flag header=None, the entries of the first row are falsely interpreted as the column labels!

****

The *pandas* routine has the advantage that the first column is recognized as integer, whereas the second and third columns are float.

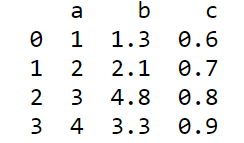
### header\_names

Specify the names of the header using the names argument.

import pandas as pd

df=pd.read\_csv('f:/python1/one.txt', names=['a', 'b', 'c'])

print(df)



### custom index

This specifies a column in the csv file to customize the index using **index\_col.**

import pandas as pd

df=pd.read\_csv('f:/python1/one.txt', index\_col=['name'])

print(df)

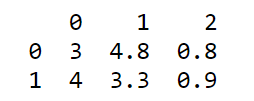
### skiprows

skiprows skips the number of rows specified.

import pandas as pd

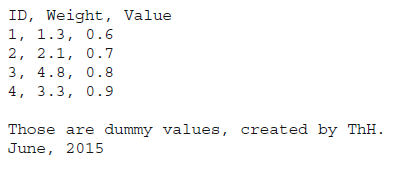
df=pd.read\_csv('f:/python1/one.txt',header=None,skiprows=2)

print(df)



**More Complex Text-Files**

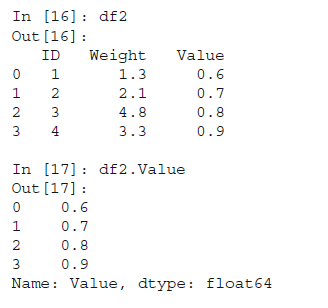
The advantage of using *pandas* for data input becomes clear with more complex files. Take for example the input file “data2.txt,” containing the following lines:

****

One of the input flags of pd.read\_csv is skip footer, so we can read in the data easily with

****

The last option, delimiter='[ ,]\*', is a *regular expression* (see below) specifying that “one or more spaces and/or commas may be used to separate entry values.” Also, when the input file includes a header row with the column names, the data can be accessed immediately with their corresponding column name, e.g.:

****

1. **Regular Expressions**

Working with text data often requires the use of simple *regular expressions*. Regular expressions are a very powerful way of finding and/or manipulating text strings. Many books have been written about them, and good, concise information on regular expressions can be found on the web, for example at:

• https://www.debuggex.com/cheatsheet/regex/python provides a convenient cheat sheet for regular expressions in *Python*.

• http://www.regular-expressions.info gives a comprehensive description of regularexpressions.

Let me give two examples how *pandas* can make use of regular expressions:

1. Reading in data from a file, separated by a combination of commas, semicolons,

or white-spaces:

df = pd.read\_csv(inFile, sep='[ ;,]+')

The square brackets indicate a *combination* (“[: : :]”) of : : :

The plus indicates *one or more* (“+”)

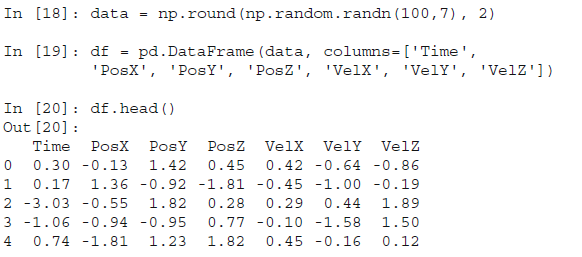
2. Extracting columns with certain name-patterns from a *pandas* DataFrame. In the

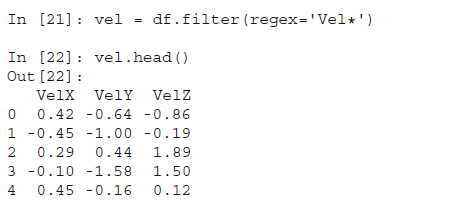
following example, I will extract all the columns starting with Vel:

data=np.round(np.random.randn(100,7),2)

df=pd.DataFrame(data, columns=['Time','Posx','posy','Posz','Velx','Vely','Velz'])

df.head()

****

****

**3.2 Input from MS Excel**

There are two approaches to reading a *Microsoft Excel* file in *pandas*: the function read\_excel, and the class ExcelFile.

* 1. read\_excel is for reading one file with file-specific arguments (i.e., identical data formats across sheets).
  2. ExcelFile is for reading one file with sheet-specific arguments (i.e., different data formats across sheets).

Choosing the approach is largely a question of code readability and execution speed.

The following commands show equivalent class and function approaches to reada single sheet:

# using the ExcelFile class

xls = pd.ExcelFile('path\_to\_file.xls')

data = xls.parse('Sheet1', index\_col=None, na\_values=['NA'])

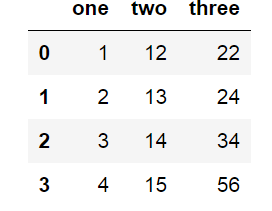
xls = pd.ExcelFile('f:/python1/excel.xlsx')

data = xls.parse('Sheet1', index\_col=None, na\_values=['NA'])

data

# using the read\_excel function

data = pd.read\_excel('path\_to\_file.xls', 'Sheet1', index\_col=None, na\_values=['NA'])

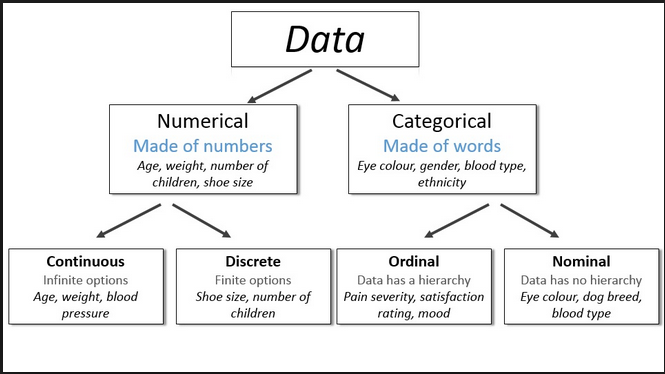


**Datatypes**

**Data** is a group of information collected. The information could be anything, and is often used to prove or disprove a hypothesis or scientific guess, during an experiment.

The choice of appropriate statistical procedure depends on the data type. Data is usually grouped into two different types of information: **categorical** and **numerical**.

In addition, we distinguish between *univariate*, *bivariate*, and *multivariate* data.*Univariate data* are data of only one variable, e.g., the size of a person. *Bivariatedata* have two parameters, for example, the position in a plane, or the income as a function of age. *Multivariate data* have three or more variables, e.g., the positionof a particle in space, etc.



***Numerical***

* Numerical variables can have values that describe a measurable quantity as number like ‘how many’ or ‘how much’.
* Numerical variables are quantitative variables.
* The values of a **numerical** variable are numbers. They can be further classified into **discrete** and **continuous** variables.

1. **Discrete variable**

A variable whose values are based on counts from a set of distinct **whole numbers i**s called discrete.

It cannot take the values of a fraction between one value and next closest value.

**Example:**

* The number of days with rain in a year is discrete.
* *Number of children in the family ( 0 1 2 3* ….)

1. **Continuous variable:**

A variable that may contain **any value within some range (set of real numbers)** is called continuous.

**Example:**

The total annual rainfall is continuous, height, time and temperature

***Categorical***

Categorical variables have the values that describe a quality or characteristic of data like what type of what category.

They fall into mutually exclusive (in one category or another) and exhaustive (include all possible options) categories.

These are qualitative variables (non numeric values)

**a) Boolean**

Boolean data are data which can only have two possible values.

**Examples:**

* female/male
* smoker/nonsmoker
* True/False
* Good/bad

**b) Nominal**

Observations can take values that is not able to be organized in a logical sequence (not in order)

These values are named categories.

**Examples:**

* Red, blue, green
* Apple, banana, orange
* set, business type, eye color, religion, brand e.t.c.

**c) Ordinal**

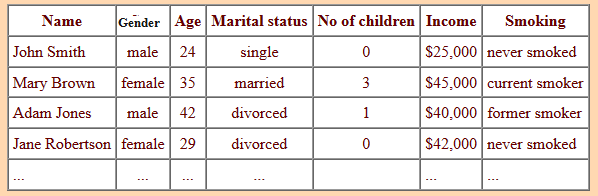
In contrast to nominal data, ordinal data can have values that can be logically ordered or ranked.

The categories associated with ordinal variables can be ranked higher or lower than another, but do not necessarily establish a numeric difference between each category.

**Examples:**

* Academic grades (A,B,C),
* Clothing size (small, medium, large, extra large)
* Attitude (strongly agree, agree, disagree, strongly disagree)
* very few/few/some/many/very many.

Consider the following data set that describes characteristics of the employees of a company.



* *Name* is a label variable.
* *Gender*, *Marital Status* and *Smoking* are nominal categorical variables. (However if we regard 'former smoker' as being **between** 'never smoked' and 'current smoker' then it could be treated as ordinal.)
* *Age* and *Income* are continuous numerical variables. (Although the recorded ages have been truncated to whole numbers, the concept of age is continuous.)
* *Number of children* is a discrete numerical variable (a count).