**STATISTICS USING PYTHON (16CS353)**

**UNIT-1**

**Syllabus:** 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.

Definitions:

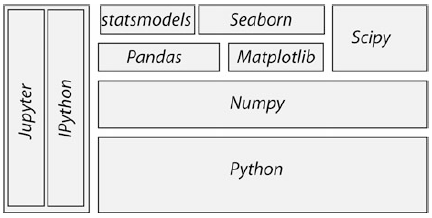
* Statistics are the sets of mathematical equations that we used to analyze the available data. It provides information.
* The field of statistics is the science of learning from data

1. **Why Statistics?:** 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.

Following are few of the examples that explain how statistics are helpful in our daily activities:

1. **Weather forecasting:** Whenever you are planning a trip or a journey, you take in account the weather conditions of the place. Have you ever think how do you get that information? There are some computers models build on statistical concepts. These computer models compare prior weather with the current weather and predict future weather.
2. **Online shopping:** Every time you go to an online shopping site like Flipkart or Amazon, you see the ratings of the products, the reviews of the customers etc. All this data that you see is available with the help of but statistics.
3. **Politics:** Every time you go to cast your vote, you compare the performance of the current government to that of the other previous governments. You also decide whom to vote on the basis of the promises made by the other party leaders which often use phrases like “We will try to improve the education rates to ……….”. All this data is part of statistics. News reporter makes a prediction of winner for elections based on political campaigns.
4. **Insurance:** You know that in order to drive your car you are required by law to have car insurance. If you have a mortgage on your house, you must have it insured as well. The rate that an insurance company charges you is based upon statistics from all drivers or homeowners in your area.
5. **Stock market:** Stock market is a concept we all hear about in our day to day lives and it will not come as a surprise to you that it is completely based on statistics as stock analysts also use statistical computer models to forecast what is happening in the economy.
6. **Sports:** Sport selections are mostly done on the basis of data about the performance and the form of the player. Also in the modern world that we live in statistics of player make the match or game more interesting and provide ground for fans to argue about the dominance of their favorite player in the game.
7. **Medical:** Statistics play a big role in the medical field. Before any drugs prescribed, scientist must show a statistically valid rate of effectiveness. Statistics are behind all the study of medical. Doctors predict disease based on statistics concepts. Suppose a survey shows that 75%-80% people have cancer and not able to find the reason. When the statistics become involved, then you can have a better idea of how the cancer may affect your body or is smoking is the major reason for it.
8. **Emergency Preparedness**: What happens if the forecast indicates that a hurricane is imminent or that tornadoes are likely to occur? Emergency management agencies move into high gear to be ready to rescue people. Emergency teams rely on statistics to tell them when danger may occur.
9. **Genetics:** Many people are afflicted with diseases that come from their genetic make-up and these diseases can potentially be passed on to their children. Statistics are critical in determining the chances of a new baby being affected by the disease.
10. **Consumer Goods:** Wal-Mart, a worldwide leading retailer, keeps track of everything they sell and use statistics to calculate what to ship to each store and when. From analyzing their vast store of information, for example, Wal-Mart decided that people buy strawberry Pop Tarts when a hurricane is predicted in Florida! So they ship this product to Florida stores based upon the weather forecast.
11. **Python Packages for Statistics**
    1. **Python Distributions:** The *Python* core distribution contains only the essential features of a general programming language. For example, it does not even contain a specialized module for working efficiently with vectors and matrices! To serve specific needs of the programmers, specialized modules are being developed by dedicated volunteers. The relationship of the most important *Python* packages for statistical applications is delineated in the following Figure.

****

**Figure1. The structure of the *Python* packages for statistical applications.**

To facilitate the use of *Python*, the so-called *Python distributions* collect matching versions of the most important packages. Popular *Python* distributions are:

**• *WinPython***recommended for Windows users. At the time of writing, the latest version was 3.5.1.3 (newer versions also ok). *WinPython*, which is free and customizable. <https://winpython.github.io/>

**• *Anaconda***by Continuum. For Windows, Mac, and Linux. Can be used to install Python 2.x and 3.x, even simultaneously! The latest *Anaconda* version at time of writing was 4.0.0 (newer versions also ok). *Anaconda* has become very popular recently, and is free for educational purposes. <https://store.continuum.io/cshop/anaconda/>

Neither of these two distributions requires administrator rights.

Python 2.7.10 and 3.5.1, under Windows and Linux, using the following package versions:

* *ipython 4.1.2* : : : For interactive work.
* *numpy 1.11.0* : : : For working with vectors and arrays.
* *scipy 0.17.1* : : : All the essential scientific algorithms, including those for basic

statistics.

* *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.
* *statsmodels 0.8.0* : : : 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:

* *xlrd 0.9.4* : : : For reading and writing MS Excel files.
* *PyMC 2.3.6* : : : For Bayesian statistics, including Markov chain Monte Carlo

simulations.

* *scikit-learn 0.17.1* : : : For machine learning.
* *scikits.bootstrap 0.3.2* : : : Provides bootstrap confidence interval algorithms for

scipy.

* *lifelines 0.9.1.0* : : : Survival analysis in *Python*.
* *rpy2 2.7.4* : : : Provides a wrapper for *R*-functions in *Python*.

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

**2.2. PyPI (The Python Package Index):** It is a repository of software for the *Python* programming language currently with more than 80,000 packages.

Packages from *PyPI* can be installed easily, from the Windows command shell (cmd) or the Linux terminal, with:

$pip install [\_package\_]

To update a package, use:

$pip install [\_package\_] -U

To get a list of all the *Python* packages installed on your computer, type

$pip list

*Note: Anaconda* uses conda, a more powerful installation manager. But *pip* also works with *Anaconda*.

**2.3. *Installation of Python***

**a. Under Windows:**

**Install Python:**

* Download WinPython from https://winpython.github.io/.
* Run the downloaded .exe-file, and install *WinPython* into the [\_WinPythonDir\_] of your choice.
* After the installation, make a change to your *Windows Environment*, by typing Win -> env -> Edit environment variables for your account:
  + Add[\_WinPythonDir\_]\python-3.5.1;[\_WinPythonDir\_] \python-3.5.1\Scripts\; to your PATH.

**Install Anaconda:**

* Download *Anaconda* from https://store.continuum.io/cshop/anaconda/.
* Follow the installation instructions from the webpage.
* During the installation, allow *Anaconda* to make the suggested modifications to your environment PATH.
* After the installation: in the *Anaconda Launcher*, click *update* (besides the Apps), in order to ensure that you are running the latest version.

**Installing Additional Packages:**

**Using pip:** Packages from *PyPI* can be installed easily at terminal, with:

$pip install [\_package\_]

**Using wheel file:** pre-compiled packages from Christoph Gohlke, available under ttp://www.lfd.uci.edu/~gohlke/pythonlibs/: download the [\_xxx\_x].whl file for your current version of *Python*, and then

install it simply with pip install [\_xxx\_].whl.

**b. Under Linux**

* Download *Anaconda* for *Python 3.5*
* Openterminal, and navigate to the location where you downloaded the file to.
* Install *Anaconda* with bash Anaconda3-4.0.0-Linux-x86.sh
* Update your Linux installation with sudo apt-get update

**2.4.First Python Programs**

**2.4.1.**

**a. Python Shell:** *Python* is an interpreted language. The simplest way to start *Python* is to type python on the command line.

Ex:

>>> print('Hello World')

Hello World

However, better start out with the *IPython/Jupyter qtconsole.* 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 see information about the possible input arguments for the command print.

**b. Python Modules:** Often we 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 the following on the command line:

$ python helloWorld.py

**Ex:** write a *Python* module which prints out the square of the numbers from zero to five.

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

def squared(x):

return x\*\*2

for i in range(6):

print(i, squared(i))

print('Done')

**2.4.2 Python Data Structures:**

*Python* offers a number of powerful data structures.

* *Tuples* to group objects of different types.
* *Lists* to group objects of the same types.
* *Arrays* to work with numerical data.
* *Dictionaries* for named, structured data sets.
* *DataFrames* for statistical data analysis.

1. **Tuple:** A collection of different things. Tuples are “immutable”, i.e., they cannot be modified after creation.

In [1]: import numpy as np

In [2]: myTuple = ('abc', 0, 3, 0.2, 2.5)

In [3]: myTuple[2]

Out[3]: 0.2

1. **List:** Lists are “mutable”, i.e., their elements can be modified. Therefore lists are typically used to collect items of the same type (numbers, strings, : : :).

Note that “+” concatenates lists.

In [4]: myList = ['abc', 'def', 'ghij']

In [5]: myList.append('klm')

In [6]: myList

Out[6]: ['abc', 'def', 'ghij', 'klm']

In [7]: myList2 = [1,2,3]

In [8]: myList3 = [4,5,6]

In [9]: myList2 + myList3

Out[9]: [1, 2, 3, 4, 5, 6]

1. **Dictionary:** Dictionaries are unordered *(key/value)* collections of content, where the content is addressed as dict['key']. Dictionaries can be created with the command dict, or by using curly brackets {...}:

In [14]: myDict = dict(one=1, two=2, info='some information')

In [15]: myDict2 = {'ten':1, 'twenty':20,

'info':'more information'}

In [16]: myDict['info']

Out[16]: 'some information'

In [17]: myDict.keys()

Out[17]: dict\_keys(['one', 'info', 'two'])

1. **Data Frames:** these are the data structures optimized for working with named, statistical data. These are defined in pandas.
2. ***Indexing and Slicing:*** The rules for addressing individual elements in *Python* lists or tuples or in *numpy* arrays are pretty simple really.

a[start:end] # items start through end-1

a[start:] # items start through the rest of the array

a[:end] # items from the beginning through end-1

a[:] # a copy of the whole array

There is also the step value, which can be used with any of the above:

a[start:end:step] # start through not past end, by step

The key points to remember are that indexing starts at 0, *not* at 1; The other feature is that start or end may be a negative number, which means it counts from the end of the array instead of the beginning. So:

a[-1] # last item in the array

a[-2:] # last two items in the array

a[:-2] # everything except the last two items

As a result, a[:5] gives you the first five elements



***2.4. Vectors and Arrays***

*numpy* is the *Python* module that makes working with numbers efficient. It is commonly imported with:

import numpy as np

By default, it produces vectors. The commands most frequently used to generate numbers are:

**np.zeros():** generates zeros. Note that it takes only one(!) input. If you want to generate a matrix of zeroes, this input has to be a tuple, containing the number of rows/columns!

In [1]: import numpy as np

In [2]: np.zeros(3)

Out[2]: array([ 0., 0., 0.])

In [3]: np.zeros( (2,3) )

Out[3]: array([[ 0., 0., 0.],

[ 0., 0., 0.]])

**np.ones():** generates ones. Note that it takes only one(!) input. If you want to generate a matrix of ones, this input has to be a tuple, containing the number of rows/columns!

In [1]: import numpy as np

In [2]: np.ones(3)

Out[2]: array([ 1., 1., 1.])

In [3]: np.zeros( (2,3) )

Out[3]: array([[1., 1., 1.],

[1., 1., 1.]])

**np.random.randn():** generates normally distributed numbers, with a mean of 0 and a standard deviation of 1. Takes two arguments to indicate number of rows and columns.

In [1]: import numpy as np

In [2]: np.random.randn(5)

Out[2]: [-0.32681637 0.47614574 -0.46351652 2.61584517 -2.26964951]

In [3]: np.random.randn(2,5)

Out[3]: [[-0.48765098 0.04198691 0.3842734 -1.57954682 -1.22633607]

[-0.43406053 -1.23842362 1.36874546 -2.20495994 0.87471632]]

**np.arange():** generates a range of numbers. Parameters can be start, end, steppingInterval. Note that the end-value is excluded! While this can sometimes be a bit awkward, it has the advantage that consecutive sequences can be easily generated, without any overlap, and without missing any data points:

In [4]: np.arange(3)

Out[4]: array([0, 1, 2])

In [5]: np.arange(1,3,0.5)

Out[5]: array([ 1. , 1.5, 2. , 2.5])

In [7]: np.arange(3,5,0.5)

Out[9]: array([ 3., 3.5, 4., 4.5])

**np.linspace():** generates linearly spaced numbers.

In [10]: np.linspace(0,10,6)

Out[10]: array([ 0., 2., 4., 6., 8., 10.])

**np.array():** generates a numpy array from given numerical data.

In [11]: np.array([[1,2], [3,4]])

Out[11]: array([ [1, 2],

[3, 4] ])

Note: Matrices are simply “lists of lists”. Therefore the first element of a matrix gives

you the first row:

In [12]: Amat = np.array([ [1, 2],

[3, 4] ])

In [13]: Amat[0]

Out[13]: array([1, 2])

A vector is not the same as a one-dimensional matrix! This is one of the few really un-intuitive features of *Python*, and can lead to mistakes that are hard to find. For example, vectors cannot be transposed, but matrices can.

In [14]: x = np.arange(3)

In [15]: Amat = np.array([ [1,2], [3,4] ])

In [16]: x.T == x

Out[16]: array([ True, True, True], dtype=bool)

In [17]: Amat.T == Amat

Out[17]: array([[ True, False],

[False, True]], dtype=bool)

***2.4. Functions, Modules, and Packages***

*Python* has three different levels of modularization:

**Function:** A *function* is defined by the keyword def, and can be defined anywhere in *Python*. It returns the object in the return statement, typically at the end of the function.

**Modules:** A *module* is a file with the extension “.py”. Modules can contain function and variable definitions, as well as valid *Python* statements.

**Packages:** A *package* is a folder containing multiple *Python* modules, and must have a file named \_\_init\_\_.py. For example, *numpy* is a *Python* package. Since *packages* are mainly important for grouping a larger number of modules, they won’t be discussed in this book.

**Pandas Data Structures**

Pandas deal with the following three data structures

* 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. |

1. **Series:** Series is a one-dimensional array like structure with homogeneous data. For example, the following series is a collection of integers 10, 23, 56, …

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 10 | 23 | 56 | 17 | 52 | 61 | 73 | 90 | 26 | 72 |

**Key Points:**

* Homogeneous data
* Size Immutable
* Values of Data Mutable

1. **DataFrame:** DataFrame is a two-dimensional array with heterogeneous data. For example,

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Age** | **Gender** | **Rating** |
| Steve | 32 | Male | 3.45 |
| Lia | 28 | Female | 4.6 |
| Vin | 45 | Male | 3.9 |
| Katie | 38 | Female | 2.78 |

The table represents the data of a sales team of an organization with their overall performance rating. The data is represented in rows and columns. Each column represents an attribute and each row represents a person.

### Key Points:

* Heterogeneous data
* Size Mutable
* Data Mutable

1. **Panel:** Panel is a three-dimensional data structure with heterogeneous data. It is hard to represent the panel in graphical representation. But a panel can be illustrated as a container of Data Frame.

**Key Points:**

* Heterogeneous data
* Size Mutable
* Data Mutable

## pandas.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 and dicts |
| 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

**a. 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)

**b. 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.

**Example 2**

#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,index=[100,101,102,103])

print s

Its **output** is as follows −

100 a

101 b

102 c

103 d

dtype: object

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

**c. 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 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)

print s

Its **output** is as follows −

a 0.0

b 1.0

c 2.0

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).

1. **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

1. **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'

**Pandas.DataFrames**

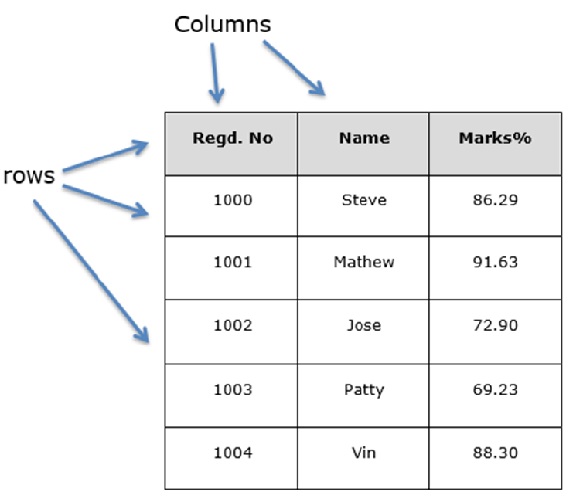
A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.

Features of DataFrame

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

Structure

Let us assume that we are creating a data frame with student’s data.



You can think of it as an SQL table or a spreadsheet data representation.

pandas.DataFrame

A pandas DataFrame can be created using the following constructor −

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

The parameters of the constructor are as follows −

|  |  |
| --- | --- |
| **S.No** | **Parameter & Description** |
| 1 | **data**  data takes various forms like ndarray, series, map, lists, dict, constants and also another DataFrame. |
| 2 | **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. |
| 3 | **columns**  For column labels, the optional default syntax is - np.arrange(n). This is only true if no index is passed. |
| 4 | **dtype**  Data type of each column. |
| 4 | **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

In the subsequent sections of this chapter, we will see how to create a DataFrame using these inputs.

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

Its **output** is as follows −

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 −

Age Name

0 28 Tom

1 34 Jack

2 29 Steve

3 42 Ricky

**Note** − Observe the values 0,1,2,3. They are the default index assigned to each 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 −

Age Name

rank1 28 Tom

rank2 34 Jack

rank3 29 Steve

rank4 42 Ricky

**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.** A **panel** is a 3D container of data. The term **Panel data** is derived from econometrics and is partially responsible for the name pandas − **pan(el)-da(ta)**-s.

The names for the 3 axes are intended to give some semantic meaning to describing operations involving panel data. They are −

* **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.

pandas.Panel()

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 |
| major\_axis | axis=1 |
| minor\_axis | axis=2 |
| 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

**Note** − Observe the changes in the dimensions.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

1. **Data Input**

This chapter shows how to read data into *Python*. Thus it forms the link between the chapter on *Python*, and the first chapter on statistical data analysis. It may be surprising, 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 Input from Text Files**

***3.1.1 Visual Inspection***

When the data are 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 a 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 equal to the number of columns that you expect. It can happen that everything gets read in—but into one large single column!

**Simple Text-Files**

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

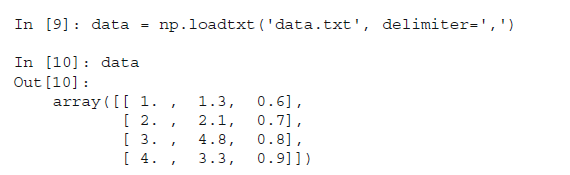
1, 1.3, 0.6

2, 2.1, 0.7

3, 4.8, 0.8

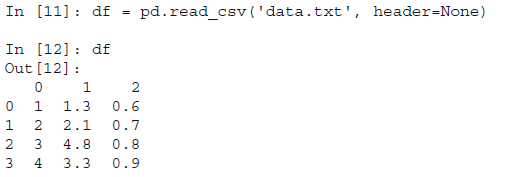
4, 3.3, 0.9

can be read in and displayed with

****

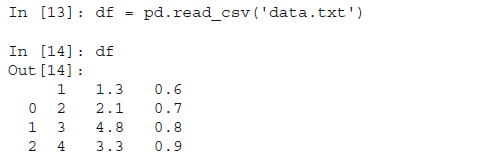
where data is a *numpy array*. Without the flag delimiter=',', the function

*np.loadtxt* crashes. An alternative way to read in these data is with

****

where df is a *pandas DataFrame*.Without the flag header=None, the entries of the

first row are falsely interpreted as the column labels!

****

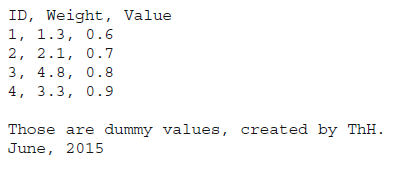
The *pandas* routine has the advantage that the first column is recognized as

integer, whereas the second and third columns are float.

**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 skipfooter, so we can read in the

data easily with

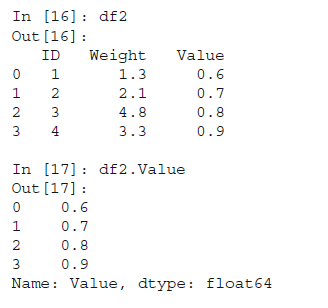
****

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.:

****

**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).

• 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 read a single sheet:

# using the ExcelFile class

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

data = xls.parse('Sheet1', index\_col=None,

na\_values=['NA'])

# using the read\_excel function

data = pd.read\_excel('path\_to\_file.xls', 'Sheet1',

index\_col=None, na\_values=['NA'])

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**4.1 Datatypes**

The choice of appropriate statistical procedure depends on the data type. Data can be *categorical* or *numerical*. If the variables are numerical, we are led to a certain statistical strategy. In contrast, if the variables represent qualitative categorizations, then we follow a different path.

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. *Bivariate data* have two parameters, for example, the *x*=*y* position in a plane, or the income as a function of age. *Multivariate data* have three or more variables, e.g., the position of a particle in space, etc.

***4.1.1 Categorical***

**a) Boolean**

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

For example,

• female/male

• smoker/nonsmoker

• True/False

**b) Nominal**

Many classifications require more than two categories. Such data are called *nominal data*. An example is *married/single/divorced*.

**c) Ordinal**

In contrast to nominal data, *ordinal data* are ordered and have a logical sequence, e.g., *very few/few/some/many/very many*.

***4.1.2 Numerical***

**a) Numerical Continuous**

Whenever possible, it is best to record the data in their original continuous format, and only with a meaningful number of decimal places. For example, it does not make sense to record the body size with more than 1mm accuracy, as there are larger changes in body height between the size in the morning and the size in the evening, due to compression of the intervertebral disks.

**b) Numerical Discrete**

Some numerical data can only take on integer values. These data are called *numerical discrete*. For example *Number of children: 0 1 2 3 4 5* ….