Python Tools for Data Analysis

Python and its packages – download and installation

There are various ways in which one can access and install Python and its packages. Here we will discuss a couple of them.

Anaconda

Anaconda is a popular Python distribution consisting of more than 195 popular Python packages. Installing Anaconda automatically installs many of the packages we will use, but they can be accessed only through an IDE called Spyder (more on this later), which itself is installed on Anaconda installation. Anaconda also installs Jupyter notebooks and when you click on the Notebook icon, it opens a browser tab and a Command Prompt.

Anaconda can be downloaded and installed from the following web address: http://continuum.io/downloads it is already installed on the lab machines in the college.

Download the suitable installer and double click on the `.exe` file and it will install Anaconda. Two of the features that you must check after the installation are:

- Jupyter Notebook http://jupyter.org/
- Spyder IDE https://pythonhosted.org/spyder/

Search for them in the "Start" icon's search, if it doesn't appear in the list of programs and files by default. We will be using Spyder and Jupyter Notebook extensively

Jupyter Notebooks can be opened by clicking on the icon. Alternatively, you can use the Command Prompt to open Jupyter Notebook. Just navigate to the directory where you have installed Anaconda and then write `Jupyter notebook.`

Installing a Python package

There are several ways to install a Python package. pip and conda are two of the most common methods.

As you might be aware, `pip` is a package management system that is used to install and manage software packages written in Python. To be able to use it to install other packages, `pip` needs to be installed first. Information on using pip is available at https://packaging.python.org/tutorials/installing-packages/

Conda is installed as part of the Anaconda install. More information on managing packages with conda is available at https://conda.io/docs/user-guide/tasks/manage-pkgs.html

Python and its packages for predictive modelling

In this section, we will discuss some commonly used packages for predictive modelling.

pandas: The most important and versatile package that is used widely in data science domains is `pandas` and it is no wonder that you can see `import pandas` at the beginning of any data science code snippet. Among other things, the `pandas` package facilitates:

- The reading of a dataset in a usable format (data frame in case of Python)
- Calculating basic statistics
- Running basic operations like sub-setting a dataset, merging/concatenating two datasets, handling missing data, and so on

The various methods in `pandas` will be explained as and when we use them.

To get an overview, navigate to the official page of *pandas* here: http://pandas.pydata.org/index.html

NumPy: NumPy, in many ways, is a MATLAB equivalent in the Python environment. It has powerful methods to do mathematical calculations and simulations. The following are some of its features:

- A powerful and widely used a N-d array element
- An ensemble of powerful mathematical functions used in linear algebra, Fourier transforms, and random number generation
- A combination of random number generators and an N-d array elements is used to generate dummy datasets to demonstrate various procedures, a practice we will follow.

To get an overview, navigate to official page of NumPy at http://www.NumPy.org/

matplotlib: matplotlib is a Python library that easily generates high-quality 2-D plots. Again, it is very similar to MATLAB.

- It can be used to plot all kind of common plots, such as histograms, stacked and unstacked bar charts, scatterplots, heat diagrams, box plots, power spectra, error charts, and so on
- It can be used to edit and manipulate all the plot properties such as title, axes properties, colour, scale, and so on

To get an overview, navigate to the official page of *matplotlib* at: http://matplotlib.org

Scikit-learn: `scikit-learn` is the mainstay of any predictive modelling in Python. It is a robust collection of all the data science algorithms and methods to implement them. Some of the features of `scikit-learn` are as follows:

- It is built entirely on Python packages like `pandas`, `NumPy`, and `matplotlib`
- It is very simple and efficient to use

- It has methods to implement most of the predictive modelling techniques, such as linear regression, logistic regression, clustering, and Decision Trees
- It gives a very concise method to predict the outcome based on the model and measure the accuracy of the outcomes

Note

To get an overview, navigate to the official page of `scikit-learn` here: http://scikit-learn.org/stable/index.html

Python packages, other than these, if used, will be situation based and can be installed using the method described earlier in this lab.

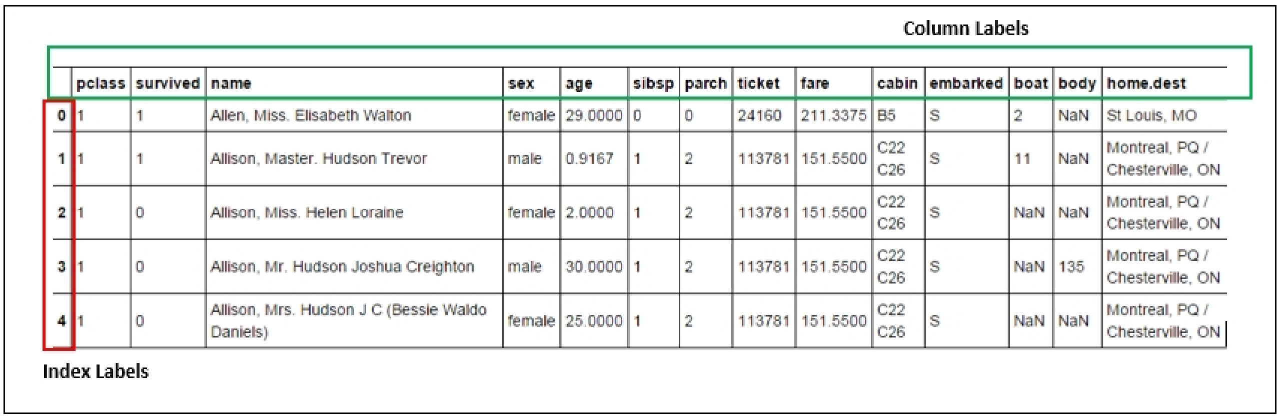
Basic Data Analysis with Python

Reading data

Data frames

A **data frame** is one of the most common data structures available in Python. Data frames are very similar to the tables in a spreadsheet or a SQL table. In Python vocabulary, it can also be thought of as a dictionary of series objects (in terms of structure). A data frame, like a spreadsheet, has index labels (analogous to rows) and column labels (analogous to columns). It is the most commonly used pandas object and is a 2D structure with columns of different or same types. Most of the standard operations, such as aggregation, filtering, pivoting, and so on which can be applied on a spreadsheet or the SQL table can be applied to data frames using methods in `pandas`.

The following screenshot is an illustrative picture of a data frame. We will learn more about working with them as we progress:



	pclass	survived	name	sex	age	sibsp	parch	ticket	fare	cabin	embarked	boat	body	home.dest
0	1	1	Allen, Miss. Elisabeth Walton	female	29.0000	0	0	24160	211.3375	B5	S	2	NaN	St Louis, MO
1	1	1	Allison, Master. Hudson Trevor	male	0.9167	1	2	113781	151.5500	C22 C26	S	11	NaN	Montreal, PQ / Chesterville, ON
2	1	0	Allison, Miss. Helen Loraine	female	2.0000	1	2	113781	151.5500	C22 C26	S	NaN	NaN	Montreal, PQ / Chesterville, ON
3	1	0	Allison, Mr. Hudson Joshua Creighton	male	30.0000	1	2	113781	151.5500	C22 C26	S	NaN	135	Montreal, PQ / Chesterville, ON
4	1	0	Allison, Mrs. Hudson J C (Bessie Waldo Daniels)	female	25.0000	1	2	113781	151.5500	C22 C26	S	NaN	NaN	Montreal, PQ / Chesterville, ON

Delimiters

A **delimiter** is a special character that separates various columns of a dataset from one another. The most common (one can go to the extent of saying that it is a default delimiter) delimiter is a comma (`,`). A `.csv` file is called so because it has comma separated values. However, a dataset can have any special character as its delimiter and one needs to know how to juggle and manage them in order to do an exhaustive and exploratory analysis and build a robust predictive model.

Various methods of importing data in Python

`pandas` is the Python library/package of choice to import, wrangle, and manipulate datasets. The datasets come in various forms; the most frequent being in the `.csv` format. The delimiter (a special character that separates the values in a dataset) in a CSV file is a comma. Now we will look at the various methods in which you can read a dataset in Python.

Case 1 – reading a dataset using the read_csv method

Open an new spyder scrip or Jupyter Notebook.

Download the Titanic dataset from the lab folder (titanic3.csv). This is a very popular dataset that contains information about the passengers travelling on the famous ship Titanic on the fateful sail that saw it sinking.

A common practice is to share a variable description file with the dataset describing the context and significance of each variable. Since this is the first dataset we are encountering, here is the data description of this dataset to get a feel of how data description files actually look like:

```
VARIABLE DESCRIPTIONS:
pclass          Passenger Class
                (1 = 1st; 2 = 2nd; 3 = 3rd)
survival        Survival
                (0 = No; 1 = Yes)
name            Name
sex             Sex
age             Age
sibsp           Number of Siblings/Spouses Aboard
parch           Number of Parents/Children Aboard
ticket          Ticket Number
fare            Passenger Fare
cabin           Cabin
embarked        Port of Embarkation
                (C = Cherbourg; Q = Queenstown; S = Southampton)
boat            Lifeboat
body            Body Identification Number
home.dest       Home/Destination
```

The following code snippet is enough to import the dataset and get you started:

```
import pandas as pd
data = pd.read_csv('C:/YOUR_FILE_PATH/titanic3.csv')
```

The name of the method doesn't unveil its full might. It is a kind of misnomer in the sense that it makes us think that it can be used to read only CSV files, which is not the case. Various kinds of files, including `.txt` files having delimiters of various kinds can be read using this method.

Let's learn a little bit more about the various arguments of this method in order to assess its true potential. Although the `read_csv` method has close to 30 arguments. The general form of a `read_csv` statement is something similar to:

```
pd.read_csv(filepath, sep=', ', dtype=None, header=None, skiprows=None,
index_col=None, skip_blank_lines=TRUE, na_filter=TRUE)
```

More details available at https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html

Case 2 - reading a dataset using the open method of Python

pandas is a very robust and comprehensive library to read, explore, and manipulate a dataset. But, it might not give an optimal performance with very big datasets as it reads the entire dataset, all at once, and blocks the majority of computer memory. Instead, you can try one of the Python's file handling methods—`open`. One can read the dataset line by line or in chunks by running a `for` loop over the rows and delete the chunks from the memory, once they have been processed. Let us look at some of the use case examples of the `open` method.

Reading a dataset line by line

As you might be aware that while reading a file using the `open` method, we can specify to use a particular mode that is read, write, and so on. By default, the method opens a file in the read-mode. This method can be useful while reading a big dataset, as this method reads data line-by-line (not at once, unlike what pandas does). You can read datasets into chunks using this method.

Let us now go ahead and open a file using the `open` method and count the number of rows and columns in the dataset:

```
data=open('C:/FILE_PATH/Customer Churn Model.txt','r')
cols=data.next().strip().split(',')
no_cols=len(data.next().strip().split(','))
```

Case 3 – reading data from a URL

Several times, we need to read the data directly from a web URL. This URL might contain the data written in it or might contain a file which has the data. For example, navigate to this website, http://winterolympicsmedals.com/ which lists the medals won by various countries in different sports during the Winter Olympics. Now type the following address in the URL address bar: http://winterolympicsmedals.com/medals.csv.

A CSV file will be downloaded automatically. If you choose to download it manually, saving it and then specifying the directory path for the `read_csv` method is a time consuming process. Instead, Python allows us to read such files directly from the URL. Apart from the significant saving in time, it is also beneficial to loop over the files when there are many such files to be downloaded and read in.

A simple `read_csv` statement is required to read the data directly from the URL:

```
import pandas as pd
medal_data=pd.read_csv('http://winterolympicsmedals.com/medals.csv')
```

Alternatively, to work with URLs to get data, one can use a couple of Python packages, which we have not used till now, that is `csv` and `urllib`. The readers can go to the documentation of the packages to learn more about these packages. It is sufficient to know that `csv` provides a range of methods to handle the CSV files, while `urllib` is used to navigate and access information from the URL. Here is how it can be done:

```
import csv
import urllib2

url='http://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data'
response=urllib2.urlopen(url)
cr=csv.reader(response)

for rows in cr:
  print rows
```

Case 4 – miscellaneous cases

Apart from the standard cases described previously, there are certain less frequent cases of data file handling that might need to be taken care of. Let's have a look at two of them.

Reading from an .xls or .xlsx file

Go to the lab drive and look for `.xls` and `.xlsx` versions of the Titanic dataset. They will be named `titanic3.xls` and `titanic3.xlsx`. Download both of them and save them on your computer. The ability to read Excel files with all its sheets is a very powerful technique available in pandas. It is done using a `read_excel` method, as shown in the following code:

```
import pandas as pd
data=pd.read_excel('C:/FILE_PATH/titanic3.xls','titanic3')

import pandas as pd
data=pd.read_excel('C:/FILE_PATH/titanic3.xlsx','titanic3')
```

It works with both, `.xls` and `.xlsx` files. The second argument of the `read_excel` method is the sheet name that you want to read in.

Another available method to read a delimited data is `read_table`. The `read_table` is exactly similar to `read_csv` with certain default arguments for its definition. In some sense, `read_table` is a more generic form of `read_csv`.

Writing to a CSV or Excel file

A data frame can be written in a CSV or an Excel file using a `to_csv` or `to_excel` method in pandas. Let's go back to the `df` data frame that we created in *Case 2 – reading a dataset using the open method of Python*. This data frame can be exported to a directory in a CSV file, as shown in the following code:

```
df.to_csv('C:/FILE_PATH/Customer Churn Model.csv'
```

Or to an Excel file, as follows:

```
df.to_excel('C:/FILE_PATH/Customer Churn Model.csv'
```

Basics – summary, dimensions, and structure

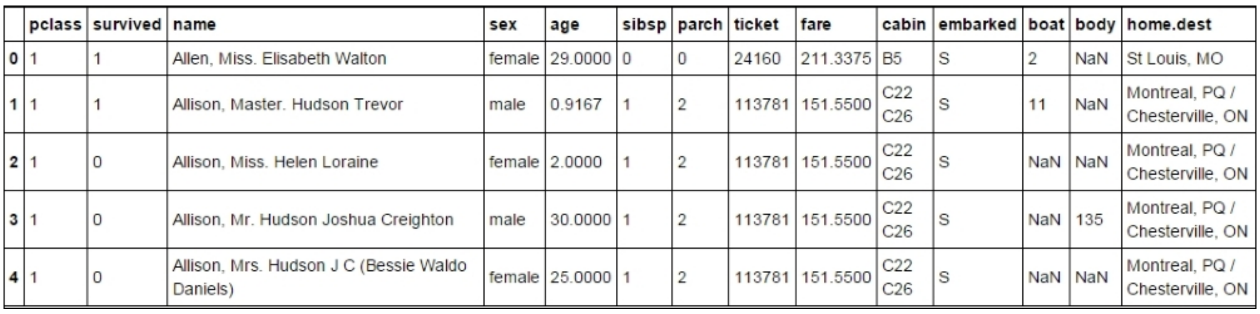
After reading in the data, there are certain tasks that need to be performed to get the touch and feel of the data:

- To check whether the data has read in correctly or not
- To determine how the data looks; its shape and size
- To summarize and visualize the data
- To get the column names and summary statistics of numerical variables

Let us go back to the example of the Titanic dataset and import it again. The `head()` method is used to look at the first first few rows of the data, as shown:

```
import pandas as pd
data=pd.read_csv('c:/FILE_PATH/titanic3.csv')
data.head()
```

The result will look similar to the following screenshot:



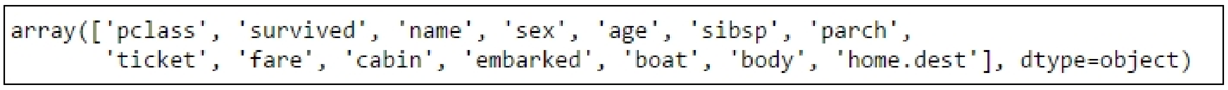
	pclass	survived	name	sex	age	sibsp	parch	ticket	fare	cabin	embarked	boat	body	home.dest
0	1	1	Allen, Miss. Elisabeth Walton	female	29.0000	0	0	24160	211.3375	B5	S	2	NaN	St Louis, MO
1	1	1	Allison, Master. Hudson Trevor	male	0.9167	1	2	113781	151.5500	C22 C26	S	11	NaN	Montreal, PQ / Chesterville, ON
2	1	0	Allison, Miss. Helen Loraine	female	2.0000	1	2	113781	151.5500	C22 C26	S	NaN	NaN	Montreal, PQ / Chesterville, ON
3	1	0	Allison, Mr. Hudson Joshua Creighton	male	30.0000	1	2	113781	151.5500	C22 C26	S	NaN	135	Montreal, PQ / Chesterville, ON
4	1	0	Allison, Mrs. Hudson J C (Bessie Waldo Daniels)	female	25.0000	1	2	113781	151.5500	C22 C26	S	NaN	NaN	Montreal, PQ / Chesterville, ON

In the `head()` method, one can also specify the number of rows they want to see. For example, `head(10)` will show the first 10 rows.

The next attribute of the dataset that concerns us is its dimension, that is the number of rows and columns present in the dataset. This can be obtained by typing `data.shape`.

The result obtained is `(1310,14)`, indicating that the dataset has 1310 rows and 14 columns.

As discussed earlier, the column names of a data frame can be listed using `data.column.values`, which gives the following output as the result:

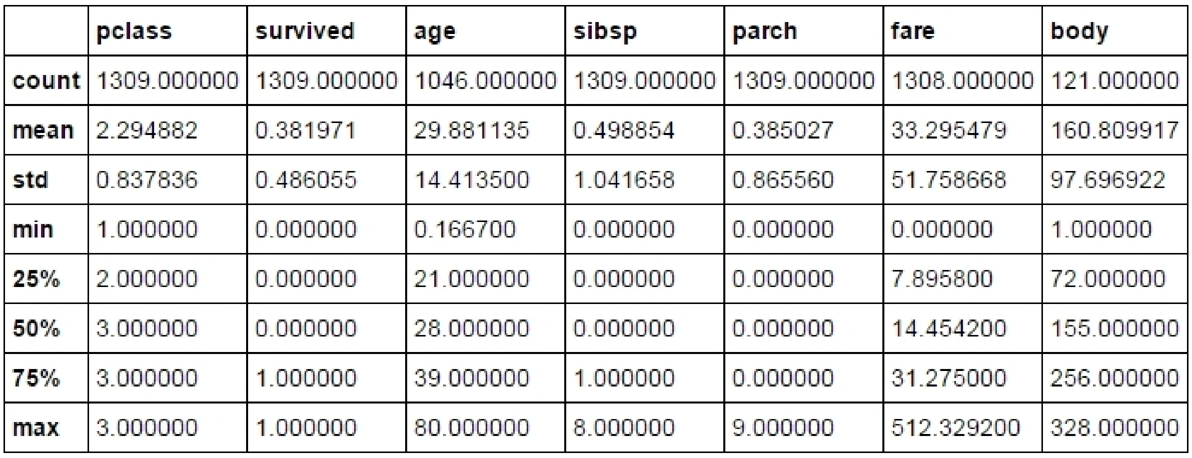


```
array(['pclass', 'survived', 'name', 'sex', 'age', 'sibsp', 'parch',
       'ticket', 'fare', 'cabin', 'embarked', 'boat', 'body', 'home.dest'], dtype=object)
```

Another important thing to do while glancing at the data is to create summary statistics for the numerical variables. This can be done by:

```
data.describe()
```

We get the following result:

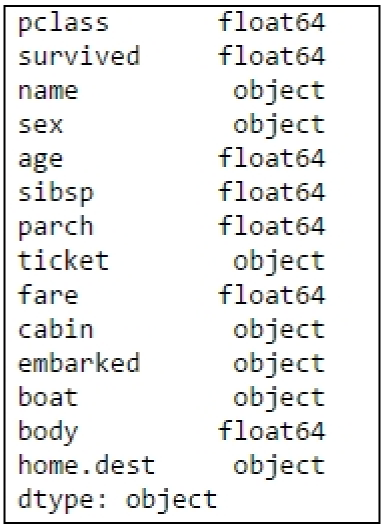


	pclass	survived	age	sibsp	parch	fare	body
count	1309.000000	1309.000000	1046.000000	1309.000000	1309.000000	1308.000000	121.000000
mean	2.294882	0.381971	29.881135	0.498854	0.385027	33.295479	160.809917
std	0.837836	0.486055	14.413500	1.041658	0.865560	51.758668	97.696922
min	1.000000	0.000000	0.166700	0.000000	0.000000	0.000000	1.000000
25%	2.000000	0.000000	21.000000	0.000000	0.000000	7.895800	72.000000
50%	3.000000	0.000000	28.000000	0.000000	0.000000	14.454200	155.000000
75%	3.000000	1.000000	39.000000	1.000000	0.000000	31.275000	256.000000
max	3.000000	1.000000	80.000000	8.000000	9.000000	512.329200	328.000000

Knowing the type each column belongs to is the key to determine their behavior under some numerical or manipulation operation. Hence, it is of critical importance to know the type of each column. This can be done as follows:

```
data.dtypes
```

We get the following result from the preceding code snippet:



```
pclass         float64
survived       float64
name            object
sex             object
age            float64
sibsp          float64
parch          float64
ticket          object
fare           float64
cabin           object
embarked        object
boat            object
body           float64
home.dest       object
dtype: object
```

Handling missing values

Checking for missing values and handling them properly is an important step in the data preparation process, if they are left untreated they can:

- Lead to the behaviour between the variables not being analysed correctly
- Lead to incorrect interpretation and inference from the data

To see how; move up a few pages to see how the `describe` method is explained. Look at the output table; why are the counts for many of the variables different from each other? There are 1310 rows in the dataset, as we saw earlier in the section. Why is it then that the count is 1046 for `age`, 1309 for `pclass`, and 121 for `body`. This is because the dataset doesn't have a value for 264 (1310-1046) entries in the `age` column, 1 (1310-1309) entry in the `pclass` column, and

1189 (1310-121) entries in the `body` column. In other words, these many entries have missing values in their respective columns. If a column has a count value less than the number of rows in the dataset, it is most certainly because the column contains missing values.

Checking for missing values

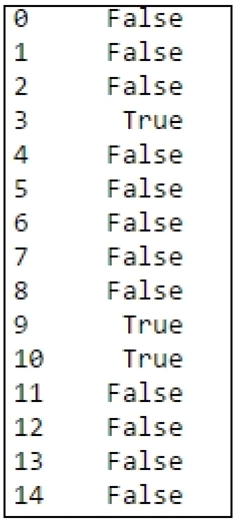
There are a multitude of in-built methods to check for missing values. Let's go through some of them. Suppose you wish to find the entries that have missing values in a column of a data frame. It can be done as follows for the `body` column of the `data` data frame:

```
pd.isnull(data['body'])
```

This will give a series indicating `True` in the cells with missing values and `False` for non-missing values. Just the opposite can be done as follows:

```
pd.notnull(data['body'])
```

The result will look something similar to the following screenshot:



```
0      False
1      False
2      False
3       True
4      False
5      False
6      False
7      False
8      False
9       True
10      True
11     False
12     False
13     False
14     False
```

Fig. 2.10: The notnull method gives False for missing values and True for non-missing values

The number of entries with missing values can be counted for a particular column to verify whether our calculation earlier about the number of missing entries was correct or not. This can be done as follows:

```
pd.isnull(data['body']).values.ravel().sum()
```

The result we get is 1189. This is the same number of missing entries from the body column as we have calculated in the preceding paragraph. In the preceding one-liner, the values (`True/False`; `1/0` in binary) have been stripped off the series and have been converted into a row (using the `ravel` method) to be able to sum them up. The sum of `1/0` values (`1` for missing values and `0` for non-missing) gives the number of total missing values.

The opposite of `isnull` is `notnull.` This should give us `121` as the result:

```
pd.nottnull(data['body']).values.ravel().sum()
```

Before we dig deeper into how to handle missing data, let's see what constitutes the missing data and how missing values are generated and propagated.

What constitutes missing data?

`Nan` is the default keyword for a missing value in Python. `None` is also considered as a missing value by the `isnull` and `notnull` functions.

How missing values are generated and propagated

There are various ways in which a missing values are incorporated in the datatset:

- **Data extraction**: While extracting data from a database, the missing values can be incorporated in the dataset due to various incompatibilities between the database server and the extraction process. In this case, the value is actually not missing but is being shown as missing because of the various incompatibilities. This can be corrected by optimizing the extraction process.
- **Data collection**: It might be the case that at the time of collection, certain data points are not available or not applicable and hence can't be entered into the database. Such entries become missing values and can't be obtained by changing the data extraction process because they are actually missing. For example, in case of a survey in a village, many people might not want to share their annual income; this becomes a missing value. Some datasets might have missing values because of the way they are collected. A time series data will have data starting from the relevant time and before that time it will have missing values.

Any numerical operator on a missing value propagates the missing value to the resultant variable. For example, while summing the entries in two columns, if one of them has a missing value in one of the entries, the resultant sum variable will also have a missing value.

Treating missing values

There are basically two approaches to handle missing values: deletion and imputation. Deletion means deleting the entire row with one or more missing entries. Imputation means replacing the missing entries with some values based on the context of the data.

Deletion

One can either delete a complete row or column. One can specify when to delete an entire row or column (when any of the entries are missing in a row or all of the entries are missing in a row). For our dataset, we can write something, as shown:

```
data.dropna(axis=0,how='all')
```

The statement when executed will drop all the rows (`axis=0` means rows, `axis=1` means columns) in which all the columns have missing values (the `how` parameter is set to `all`). One can drop a row even if a single column has a missing value. One needs to specify the `how` method as `'any'` to do that:

```
data.dropna(axis=0,how='any')
```

Imputation

Imputation is the method of adding/replacing missing values with some other values such as `0`, a string, or mean of non-missing values of that variable. There are several ways to impute a missing value and the choice of the best method depends on the context of the data.

One method is to fill the missing values in the entire dataset with some number or character variable. Thus, it can be done as follows:

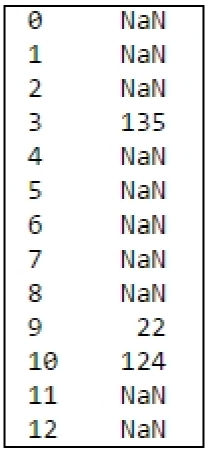
```
data.fillna(0)
```

This will replace the missing values anywhere in the dataset with the value `0`. One can impute a character variable as well:

```
data.fillna('missing')
```

The preceding statement will impute a `missing` string in place of `NaN`, `None`, blanks, and so on. Another way is to replace the missing values in a particular column only is as shown below.

If you select the `body` column of the data by typing `data['body']`, the result will be something similar to the following screenshot:



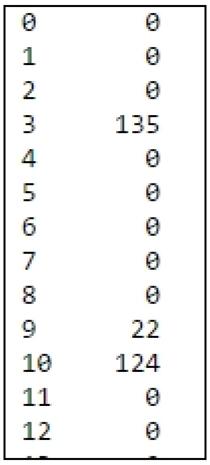
```
0      NaN
1      NaN
2      NaN
3      135
4      NaN
5      NaN
6      NaN
7      NaN
8      NaN
9       22
10     124
11     NaN
12     NaN
```

Fig. 2.11: The values in the body column of the Titanic dataset without imputation for missing values

One can impute zeros to the missing values using the following statement:

```
data['body'].fillna(0)
```

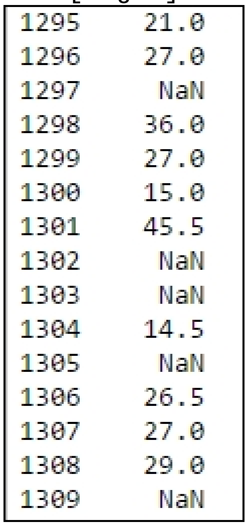
But after imputing `0` to the missing values, we get something similar to the following screenshot:



```
0       0
1       0
2       0
3     135
4       0
5       0
6       0
7       0
8       0
9      22
10    124
11      0
12      0
```

A common imputation is with the mean or median value of that column. This basically means that the missing values are assumed to have the same values as the mean value of that column (excluding missing values, of course), which makes perfect sense. Let us see how we can do that using the `fillna` method. Let us have a look at the `age` column of the dataset:

```
data['age']
```

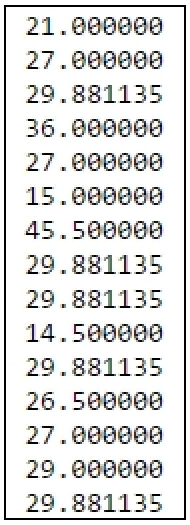


```
1295    21.0
1296    27.0
1297     NaN
1298    36.0
1299    27.0
1300    15.0
1301    45.5
1302     NaN
1303     NaN
1304    14.5
1305     NaN
1306    26.5
1307    27.0
1308    29.0
1309     NaN
```

As shown in the preceding screenshot, some of the entries in the age column have missing values. Let us see how we can impute them with mean values:

```
data['age'].fillna(data['age'].mean())
```

The output looks something similar to the following screenshot:

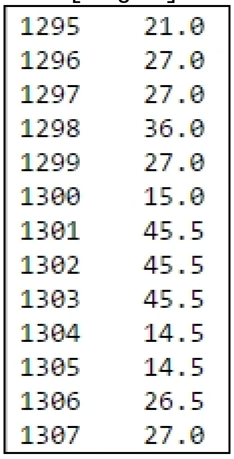


```
21.000000
27.000000
29.881135
36.000000
27.000000
15.000000
45.500000
29.881135
29.881135
14.500000
29.881135
26.500000
27.000000
29.000000
29.881135
```

As you can see, all the `NaN` values have been replaced with `29.881135`, which is the mean of the `age` column.

One can use any function in place of mean, the most commonly used functions are median or some defined calculation using `lambda`. Apart from that, there are two very important methods in `fillna` to impute the missing values: `ffill` and `backfill`. As the name suggests, `ffill` replaces the missing values with the nearest preceding non-missing value while the `backfill` replaces the missing value with the nearest succeeding non-missing value. It will be clearer with the following example:

```
data['age'].fillna(method='ffill')
```

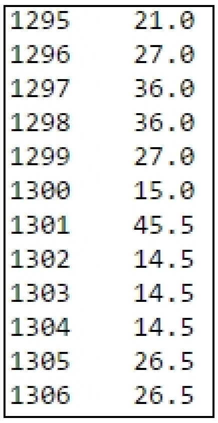


```
1295    21.0
1296    27.0
1297    27.0
1298    36.0
1299    27.0
1300    15.0
1301    45.5
1302    45.5
1303    45.5
1304    14.5
1305    14.5
1306    26.5
1307    27.0
```

As it can be seen, the missing value in row number `1297` is replaced with the value in row number `1296`.

With the `backfill` statement, something similar happens:

```
data['age'].fillna(method='backfill')
```



```
1295    21.0
1296    27.0
1297    36.0
1298    36.0
1299    27.0
1300    15.0
1301    45.5
1302    14.5
1303    14.5
1304    14.5
1305    26.5
1306    26.5
```

As it can be seen, the missing value in row number `1297` is replaced with the value in row number `1298`.

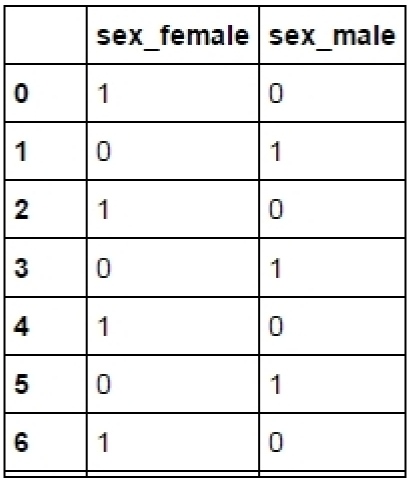
Creating dummy variables

Creating dummy variables is a method to create separate variable for each category of a categorical variable., Although, the categorical variable contains plenty of information and might show a causal relationship with output variable, it can't be used in the predictive models like linear and logistic regression without any processing.

In our dataset, `sex` is a categorical variable with two categories that are male and female. We can create two dummy variables out of this, as follows:

```
dummy_sex=pd.get_dummies(data['sex'],prefix='sex')
```

The result of this statement is, as follows:



	sex_female	sex_male
0	1	0
1	0	1
2	1	0
3	0	1
4	1	0
5	0	1
6	1	0

This process is called dummifying, the variable creates two new variables that take either `1` or `0` value depending on what the sex of the passenger was. If the sex was female, `sex_female` would be `1` and `sex_male` would be `0`. If the sex was male, `sex_male` would be `1` and `sex_female` would be `0`. In general, all but one dummy variable in a row will have a `0` value. The variable derived from the value (for that row) in the original column will have a value of `1`.

These two new variables can be joined to the source data frame, so that they can be used in the models. The method to that is illustrated, as follows:

```
column_name=data.columns.values.tolist()
column_name.remove('sex')
data[column_name].join(dummy_sex)
```

The column names are converted to a list and the sex is removed from the list before joining these two dummy variables to the dataset, as it will not make sense to have a sex variable with these two dummy variables.

Visualizing a dataset by basic plotting

Plots are a great way to visualize a dataset and gauge possible relationships between the columns of a dataset. There are various kinds of plots that can be drawn. For example, a scatter plot, histogram, box-plot, and so on.

Let's import the `Customer Churn Model` dataset and try some basic plots:

```
import pandas as pd
data=pd.read_csv('C:/FILE_PATH/Customer Churn Model.txt')
```

While plotting any kind of plot, it helps to keep these things in mind:

- If you are using Juypter Notebook, write `% matplotlib inline` in the input cell and run it before plotting to see the output plot inline (in the output cell).
- To save a plot in your local directory as a file, you can use the `savefig` method. Let's go back to the example where we plotted four scatter plots in a 2x2 panel. The name of this image is specified in the beginning of the snippet, as a `figure` parameter of the plot. To save this image one can write the following code:

- `figure.savefig('c:/FILE_PATH/Scatter Plots.jpeg')`

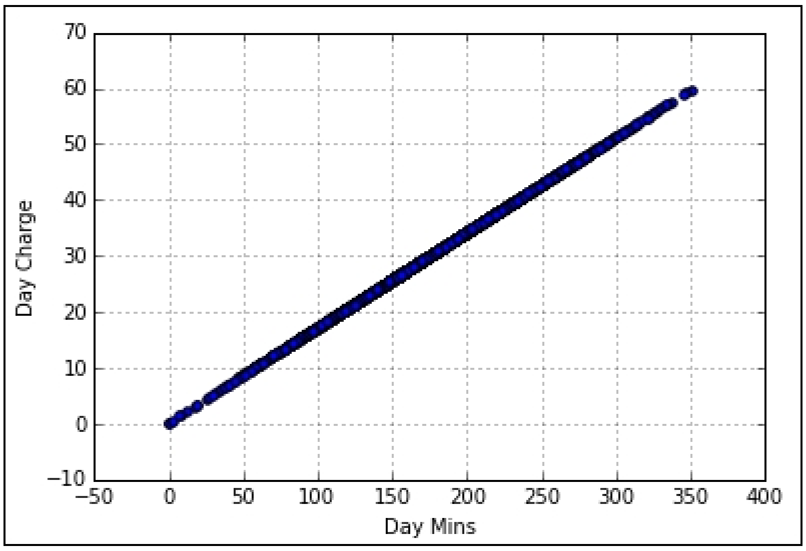
As you can see, while saving the file, one can specify the local directory to save the file and the name of the image and the format in which to save the image (`jpeg` in this case).

Scatter plots

We suspect the **Day Mins** and **Day Charge** to be highly correlated, as the calls are generally charged based on their duration. To confirm or validate our hypothesis, we can draw a scatter plot between Day Mins and Day Charge. To draw this scatter plot, we write something similar to the following code:

```
data.plot(kind='scatter',x='Day Mins',y='Day Charge')
```

The output looks similar to the following figure where the points lie on a straight line confirming our suspicion that they are (linearly) related. As we will see later in the chapter on linear regression, such a situation will give a perfect linear fit for the two variables:

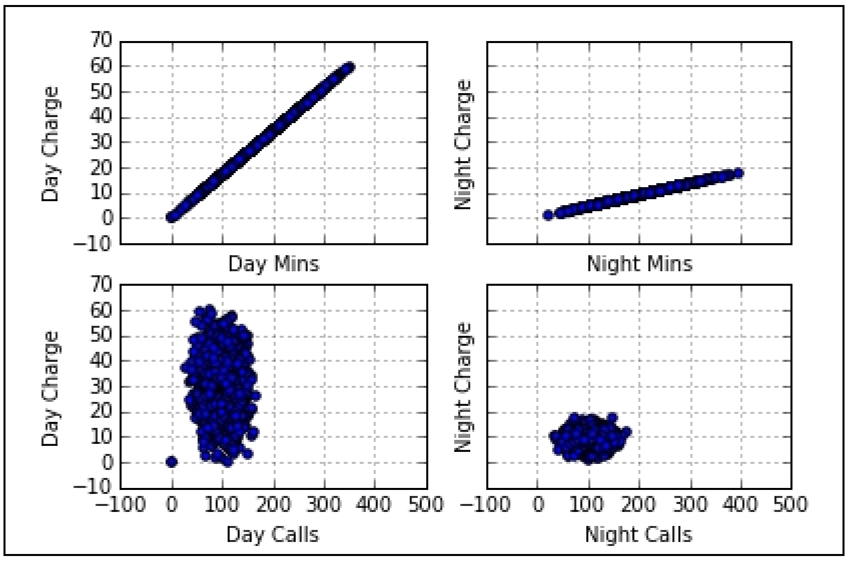


The same is the case when we plot **Night Mins** and **Night Charge** against one another. However, when we plot Night Calls with Night Charge or Day Calls with Day Charge, we don't get to see much of a relationship.

Using the `matplotlib` library, we can get good quality plots and with a lot of flexibility. Let us see how we can plot multiple plots (in different panels) in the same image:

```
import matplotlib.pyplot as plt
figure,axs = plt.subplots(2, 2,sharey=True,sharex=True)
data.plot(kind='scatter',x='Day Mins',y='Day Charge',ax=axs[0][0])
data.plot(kind='scatter',x='Night Mins',y='Night Charge',ax=axs[0][1])
data.plot(kind='scatter',x='Day Calls',y='Day Charge',ax=axs[1][0])
data.plot(kind='scatter',x='Night Calls',y='Night Charge',ax=axs[1][1])
```

Here, we are plotting four graphs in one image in a 2x2 panel using the `subplots` method of the `matplotlib` library. As you can see in the preceding snippet, we have defined the panel to be 2x2 and set `sharex` and `sharey` parameters to be `True`. For each plot, we specify their location by passing appropriate values for the `ax` parameter in the `plot` method. The result looks similar to the following screenshot:



Histograms

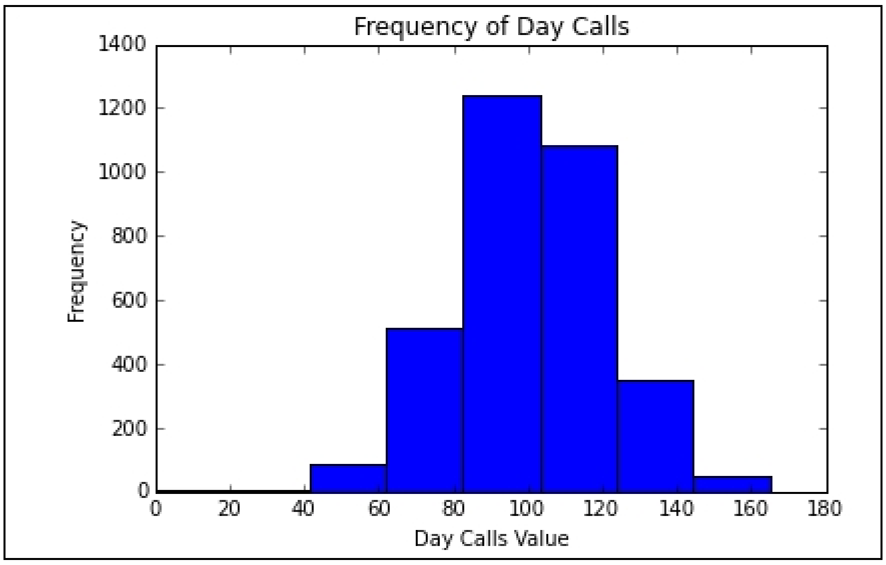
Plotting histograms is a great way to visualize the distribution of a numerical variable. Plotting a histogram is a method to understand the most frequent ranges (or bins as they are called) in which the variable lies. One can also check whether the variable is normally distributed or skewed on one side.

Let's plot a histogram for the `Day Calls` variable. We can do so by writing the following code:

```
import matplotlib.pyplot as plt
plt.hist(data['Day Calls'],bins=8)
plt.xlabel('Day Calls Value')
plt.ylabel('Frequency')
plt.title('Frequency of Day Calls')
```

The first line of the snippet is of prime importance. There we specify the variable for which we have to plot the histogram and the number of bins or ranges we want. The `bins` parameters can be passed as a fixed number or as a list of numbers to be passed as bin-edges. Suppose, a numerical variable has a minimum value of `1` and a maximum value of `1000`. While plotting histogram for this variable, one can either specify `bins=10` or `20`, or one can specify `bins=[0,100,200,300,…1000]` or `[0,50,100,150,200,…..,1000].`

The output of the preceding code snippet appears similar to the following snapshot:



Boxplots

Boxplots are another way to understand the distribution of a numerical variable. It specifies something called quartiles.

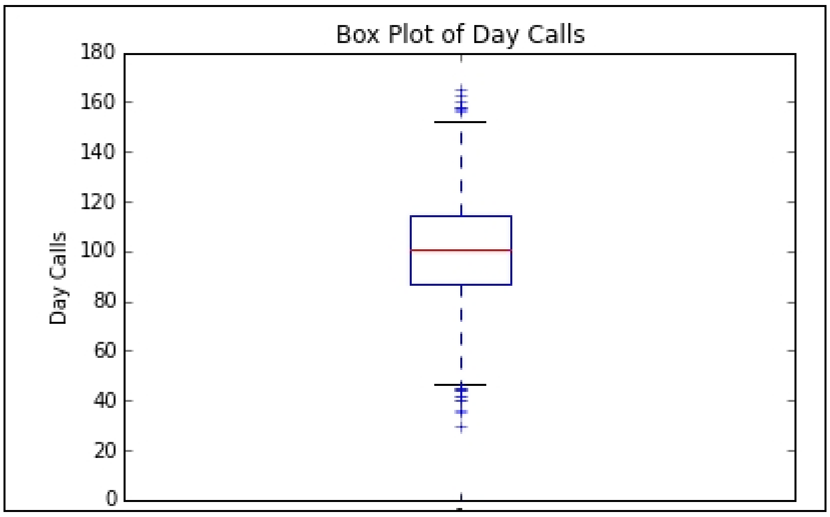
Note

If the numbers in a distribution with 100 numbers are arranged in an increasing order; the 1st quartile will occupy the 25th position, the 3rd quartile will occupy the 75th position, and so on. The median will be the average of the 50th and 51st terms. (I hope you brush up on some of the statistics you have read till now because we are going to use a lot of it, but here is a small refresher). Median is the middle term when the numbers in the distribution are arranged in the increasing order. Mode is the one that occurs with the maximum frequency, while mean is the sum of all the numbers divided by their total count.

Plotting a boxplot in Python is easy. We need to write this to plot a boxplot for `Day Calls`:

```
import matplotlib.pyplot as plt
plt.boxplot(data['Day Calls'])
plt.ylabel('Day Calls')
plt.title('Box Plot of Day Calls')
```

The output looks similar to the following snapshot:



The blue box is of prime importance. The lower-horizontal edge of the box specifies the 1st quartile, while the upper-horizontal edge specifies the 3rd quartile. The horizontal line in the red specifies the median value. The difference in the 1st and 3rd quartile values is called the Inter Quartile Range or IQR. The lower and upper horizontal edges in black specify the minimum and maximum values respectively.

The boxplots are important plots because of the following reasons:

- Boxplots are potent tools to spot outliers in a distribution. Any value that is `1.5*IQR` below the 1st quartile and is `1.5*IQR` above the 1st quartile can be classified as an outlier.
- For a categorical variable, boxplots are a great way to visualize and compare the distribution of each category at one go.

There are a variety of other types of plots that can be drawn depending on the problem at hand. We will learn about them as and when needed. For exploratory analysis, these three types are enough to provide us enough evidence to further or discard our initial hypotheses. These three types can have multiple variations and together with the power of looping and panel-wise plotting, we can make the plotting; hence, the data exploration process is very efficient.