

A COMPLETE TUTORIAL FOR PYTHON WITH DATA SCIENCE

1. Basics of Python for Data Analysis

Why learn Python for data analysis?

Python has gathered a lot of interest recently as a choice of language for data analysis. Here are some reasons which go in favor of learning Python:

- Open Source – free to install
- Awesome online community
- Very easy to learn
- Can become a common language for data science and production of web based analytics products.

Needless to say, it still has few drawbacks too:

- It is an interpreted language rather than compiled language – hence might take up more CPU time. However, given the savings in programmer time (due to ease of learning), it might still be a good choice.

Python 2.7 v/s 3.4

This is one of the most debated topics in Python. You will invariably cross paths with it, especially if you are a beginner. There is no right/wrong choice here. It totally depends on the situation and your need to use. I will try to give you some pointers to help you make an informed choice.

Why Python 2.7?

1. Awesome community support! This is something you'd need in your early days. Python 2 was released in late 2000 and has been in use for more than 15 years.
2. Plethora of third-party libraries! Though many libraries have provided 3.x support but still a large number of modules work only on 2.x versions. If you plan to use Python for specific applications like web-development with high reliance on external modules, you might be better off with 2.7.
3. Some of the features of 3.x versions have backward compatibility and can work with 2.7 version.

Why Python 3.4?

1. Cleaner and faster! Python developers have fixed some inherent glitches and minor drawbacks in order to set a stronger foundation for the future. These might not be very relevant initially, but will matter eventually.
2. It is the future! 2.7 is the last release for the 2.x family and eventually everyone has to shift to 3.x versions. Python 3 has released stable versions for past 5 years and will continue the same.

There is no clear winner but I suppose the bottom line is that you should focus on learning Python as a language. Shifting between versions should just be a matter of time. Stay tuned for a dedicated article on Python 2.x vs 3.x in the near future!

How to install Python?

There are 2 approaches to install Python:

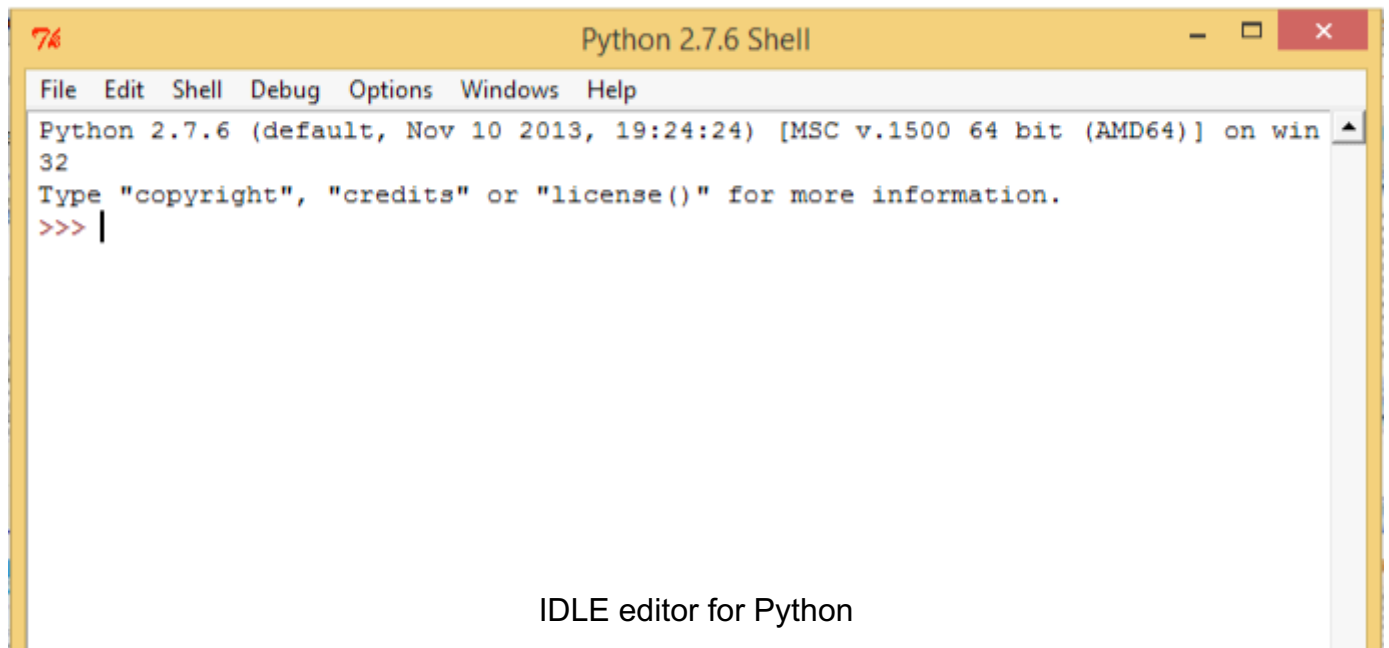
- You can download Python directly from its project site (<https://www.python.org/download/releases/2.7/>) and install individual components and libraries you want
- Alternately, you can download and install a package, which comes with pre-installed libraries. I would recommend downloading Anaconda (<https://www.continuum.io/downloads>). Another option could be **Enthought Canopy Express** (<https://www.enthought.com/downloads/>).

Second method provides a hassle-free installation and hence I'll recommend that to beginners. The imitation of this approach is you have to wait for the entire package to be upgraded, even if you are interested in the latest version of a single library. It should not matter until and unless, until and unless, you are doing cutting edge statistical research.

Choosing a development environment

Once you have installed Python, there are various options for choosing an environment. Here are the 3 most common options:

- Terminal / Shell based
- IDLE (default environment)
- iPython notebook – similar to markdown in R



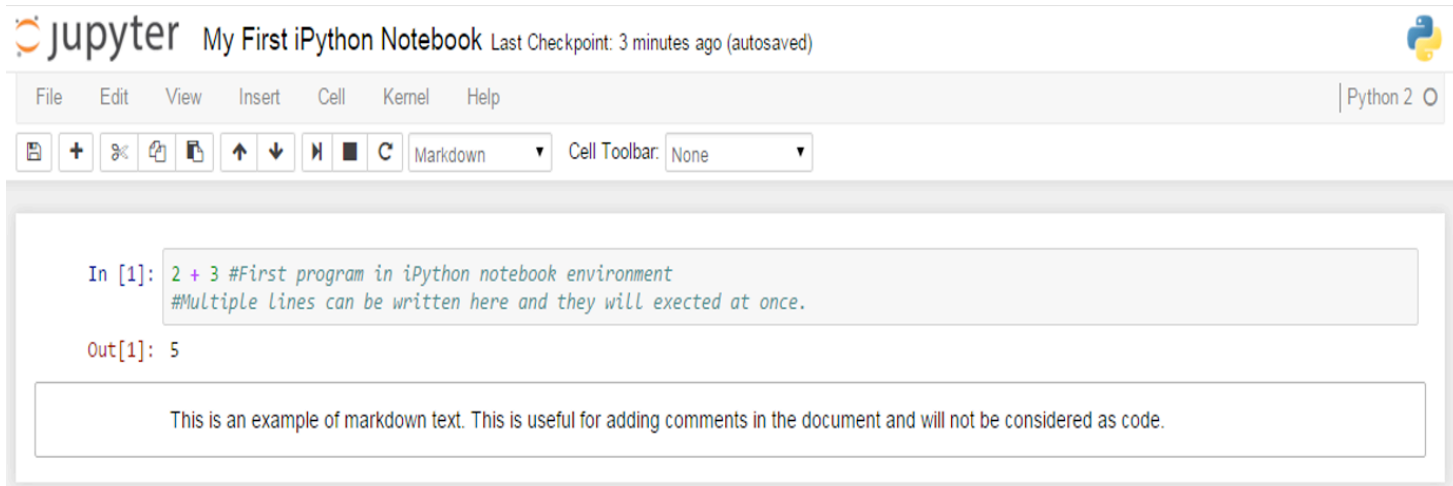
IDLE editor for Python

While the right environment depends on your need, I personally prefer iPython Notebooks a lot. It provides a lot of good features for documenting while writing the code itself and you can choose to run the code in blocks (rather than the line by line execution)

We will use iPython environment for this complete tutorial.

Warming up: Running your 1st Python program

You can use Python as a simple calculator to start with:



Few things to note

- You can start iPython notebook by writing “ipython notebook” on your terminal / cmd, depending on the OS you are working on
- You can name a iPython notebook by simply clicking on the name – Untitled in the above screenshot
- The interface shows In [*] for inputs and Out [*] for output.
- You can execute a code by pressing “Shift + Enter” or “ALT + Enter”, if you want to insert an additional row after.

Before we deep dive into problem solving, let’s take a step back and understand the basics of Python. As we know that data structures and iteration and conditional constructs form the crux of any language. In Python, these include lists, strings, tuples, dictionaries, for-loop, while-loop, if-else, etc. Let’s take a look at some of these.

2. Python libraries and Data Structures

Python Data Structures

Following are some data structures, which are used in Python. You should be familiar with them in order to use them as appropriate.

- Lists – Lists are one of the most versatile data structure in Python. A list can simply be defined by writing a list of comma separated values in square brackets. Lists might contain items of different

types, but usually the items all have the same type. Python lists are mutable and individual elements of a list can be changed.

Here is a quick example to define a list and then access it:

Lists

A list can be simply defined by writing comma separated values in square brackets.

```
In [1]: squares_list = [0,1,4,9,16,25]
```

```
In [2]: squares_list
```

```
Out[2]: [0, 1, 4, 9, 16, 25]
```

Individual elements of a list can be accessed by writing the index number in square bracket. Please note that the first index of a list is 0 and not 1

```
In [3]: squares_list[0] #Indexing returns the item
```

```
Out[3]: 0
```

A range of list can be accessed by having first index and last index

```
In [4]: squares_list[2:4] #Slicing returns a new list
```

```
Out[4]: [4, 9]
```

A Negative index accesses the list from end

```
In [5]: squares_list[-2] #It should return the second last element in the list
```

```
Out[5]: 16
```

A few common methods applicable to lists include: `append()` `extend()` `insert()` `remove()` `pop()` `count()` `sort()` `reverse()`

- **Strings** – Strings can simply be defined by use of single ('), double (") or triple (" " ") inverted commas. Strings enclosed in triple quotes (" " ") can span over multiple lines and are used frequently in doc strings (Python's way of documenting functions). \ is used as an escape character. Please note that Python strings are immutable, so you cannot change part of strings.

Strings

A string can be simply defined by using single ('), double (") or triple (" " ") quotation

```
In [6]: greeting = 'Hello'
print greeting[1]      # Return character on the index 1
print len(greeting)    # Prints length of string
print greeting + 'World' # String Concatenation

e
5
HelloWorld
```

Raw strings can be used to pass on string as is. Python interpreter does not alter the string, if you specify a string to be raw. Raw strings can be defined by adding r to the string

```
In [8]: stmt = r'\n is a newline character by default.'
print stmt

\n is a newline character by default.
```

Python strings are immutable and hence can be changed. Doing so will result in an error

```
In [9]: greeting[1:] = 'i' #Trying to change Hello to Hi. Should result in an error

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-9-2da872e33998> in <module>()
----> 1 greeting[1:] = 'i' #Trying to change Hello to Hi. Should result in an error

TypeError: 'str' object does not support item assignment
```

Common string methods include lower(), upper(), strip(), isdigit(), isspace(), find(), replace(), split() and join(). These are usually very helpful when you need to perform data manipulations or cleaning on text fields.

- Tuples – A tuple is represented by a number of values separated by commas. Tuples are immutable and the output is surrounded by parentheses so that nested tuples are processed correctly. Additionally, even though tuples are immutable, they can hold mutable data if needed.

Since Tuples are immutable and cannot change, they are faster in processing as compared to lists. Hence, if your list is unlikely to change, you should use tuples, instead of lists.

Tuples

A tuple is represented by a number of values separated by commas.

```
In [10]: tuple_example = 0, 1, 4, 9, 16, 25
```

```
In [11]: tuple_example #output would be enclosed in paranthesis
```

```
Out[11]: (0, 1, 4, 9, 16, 25)
```

```
In [12]: tuple_example[2] #Single elements can be accessed in similar fashion
```

```
Out[12]: 4
```

```
In [13]: tuple_example[2] = 6 #Tuples are immutable and hence this should result in error
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-13-6c410e816018> in <module>()
----> 1 tuple_example[2] = 6 #Tuples are immutable and hence this should result in error

TypeError: 'tuple' object does not support item assignment
```

- Dictionary – Dictionary is an unordered set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: {}.

Dictionary

A dictionary is an unordered set of key: value pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: {}.

```
In [20]: extensions = {'Kunal': 9073, 'Tavish': 9128, 'Sunil': 9223, 'Nitin': 9330}
extensions
```

```
Out[20]: {'Kunal': 9073, 'Nitin': 9330, 'Sunil': 9223, 'Tavish': 9128}
```

```
In [22]: extensions['Mukesh'] = 9150
extensions
```

```
Out[22]: {'Kunal': 9073, 'Mukesh': 9150, 'Nitin': 9330, 'Sunil': 9223, 'Tavish': 9128}
```

```
In [23]: extensions.keys()
```

```
Out[23]: ['Sunil', 'Tavish', 'Kunal', 'Mukesh', 'Nitin']
```

Python Iteration and Conditional Constructs

Like most languages, Python also has a FOR-loop which is the most widely used method for iteration. It has a simple syntax:

```
for i in [Python Iterable]:  
    expression(i)
```

Here “Python Iterable” can be a list, tuple or other advanced data structures which we will explore in later sections. Let’s take a look at a simple example, determining the factorial of a number.

```
fact=1  
for i in range(1,N+1):  
    fact *= i
```

Coming to conditional statements, these are used to execute code fragments based on a condition. The most commonly used construct is if-else, with following syntax:

```
if [condition]:  
    __execution if true__  
else:  
    __execution if false__
```

For instance, if we want to print whether the number N is even or odd:

```
if N%2 == 0:  
    print 'Even'  
else:  
    print 'Odd'
```

Now that you are familiar with Python fundamentals, let’s take a step further. What if you have to perform the following tasks:

1. Multiply 2 matrices
2. Find the root of a quadratic equation
3. Plot bar charts and histograms
4. Make statistical models
5. Access web-pages

If you try to write code from scratch, it’s going to be a nightmare and you won’t stay on Python for more than 2 days! But let’s not worry about that. Thankfully, there are many libraries with predefined which we can directly import into our code and make our life easy.

For example, consider the factorial example we just saw. We can do that in a single step as:

```
math.factorial(N)
```

O -course we need to import the math library for that. Let's explore the various libraries next.

Python Libraries

Let's take one step ahead in our journey to learn Python by getting acquainted with some useful libraries. The first step is obviously to learn to import them into our environment. There are several ways of doing so in Python:

```
import math as m
```

```
from math import *
```

In the first manner, we have defined an alias `m` to library `math`. We can now use various functions from `math` library (e.g. `factorial`) by referencing it using the alias `m.factorial()`.

In the second manner, you have imported the entire name space in `math` i.e. you can directly use `factorial ()` without referring to `math`.

Tip: Google recommends that you use first style of importing libraries, as you will know where the functions have come from.

Following are a list of libraries, you will need for any scientific computations and data analysis:

- NumPy stands for Numerical Python. The most powerful feature of NumPy is n-dimensional array. This library also contains basic linear algebra functions, Fourier transforms, advanced random number capabilities and tools for integration with other low-level languages like Fortran, C and C++
- SciPy stands for Scientific Python. SciPy is built on NumPy. It is one of the most useful library for variety of high level science and engineering modules like discrete Fourier transform, Linear Algebra, Optimization and Sparse matrices.
- Matplotlib for plotting vast variety of graphs, starting from histograms to line plots to heat plots. You can use Pylab feature in ipython notebook (ipython notebook –pylab = inline) to use these

plotting features inline. If you ignore the inline option, then pylab converts ipython environment to an environment, very similar to Matlab. You can also use Latex commands to add math to your plot.

- Pandas for structured data operations and manipulations. It is extensively used for data munging and preparation. Pandas were added relatively recently to Python and have been instrumental in boosting Python's usage in data scientist community.
- Scikit Learn for machine learning. Built on NumPy, SciPy and matplotlib, this library contains a lot of efficient tools for machine learning and statistical modeling including classification, regression, clustering and dimensionality reduction.
- Stats models for statistical modeling. Stats models 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 for statistical data visualization. Seaborn is a library for making attractive and informative statistical graphics in Python. It is based on matplotlib. Seaborn aims to make visualization a central part of exploring and understanding data.
- Bokeh for creating interactive plots, dashboards and data applications on modern web-browsers. It empowers the user to generate elegant and concise graphics in the style of D3.js. Moreover, it has the capability of high-performance interactivity over very large or streaming datasets.
- Blaze for extending the capability of Numpy and Pandas to distributed and streaming datasets. It can be used to access data from a multitude of sources including Bcolz, MongoDB, SQLAlchemy, Apache Spark, PyTables, etc. Together with Bokeh, Blaze can act as a very powerful tool for creating effective visualizations and dashboards on huge chunks of data.
- Scrapy for web crawling. It is a very useful framework for getting specific patterns of data. It has the capability to start at a website home url and then dig through web-pages within the website to gather information.
- SymPy for symbolic computation. It has wide-ranging capabilities from basic symbolic arithmetic to calculus, algebra, discrete mathematics and quantum physics. Another useful feature is the capability of formatting the result of the computations as LaTeX code.
- Requests for accessing the web. It works similar to the standard python library urllib2 but is much easier to code. You will find subtle differences with urllib2 but for beginners, Requests might be more convenient.

Additional libraries, you might need:

- os for Operating system and file operations
- networkx and igraph for graph based data manipulations
- regular expressions for finding patterns in text data
- BeautifulSoup for scrapping web. It is inferior to Scrappy as it will extract information from just a single webpage in a run.

Now that we are familiar with Python fundamentals and additional libraries, let's take a deep dive into problem solving through Python. Yes, I mean making a predictive model! In the process, we use some powerful libraries and also come across the next level of data structures. We will take you through the 3 key phases:

- 1.Data Exploration – finding out more about the data we have
- 2.Data Munging – cleaning the data and playing with it to make it better suit statistical modeling
- 3.Predictive Modeling – running the actual algorithms and having fun

3.Exploratory analysis in Python using Pandas

In order to explore our data further, let me introduce you to another animal (as if Python was not enough!) – Pandas



Image Source: Wikipedia

Pandas is one of the most useful data analysis library in Python. They have been instrumental in increasing the use of Python in data science community. We will now use Pandas to read a data set, perform exploratory analysis and build our first basic categorization algorithm for solving this problem.

Before loading the data, lets understand the 2 key data structures in Pandas – Series and Data Frames

Introduction to Series and Data frames

Series can be understood as a 1 dimensional labelled / indexed array. You can access individual elements of this series through these labels.

A data frame is similar to Excel workbook – you have column names referring to columns and you have rows, which can be accessed with use of row numbers. The essential difference being that column names and row numbers are known as column and row index, in case of data frames.

Series and data frames form the core data model for Pandas in Python. The data sets are first read into these data frames and then various operations (e.g. group by, aggregation etc.) can be applied very easily to its columns.

Practice data set – Loan Prediction Problem

The datasets for training and testing are provided with the Assignment.

Here is the description of variables:

VARIABLE DESCRIPTIONS:

Variable	Description
Loan_ID	Unique Loan ID
Gender	Male/ Female
Married	Applicant married (Y/N)
Dependents	Number of dependents
Education	Applicant Education (Graduate/ Under Graduate)
Self_Employed	Self employed (Y/N)
ApplicantIncome	Applicant income
CoapplicantIncome	Coapplicant income
LoanAmount	Loan amount in thousands
Loan_Amount_Term	Term of loan in months
Credit_History	credit history meets guidelines
Property_Area	Urban/ Semi Urban/ Rural
Loan_Status	Loan approved (Y/N)

Let's begin with exploration

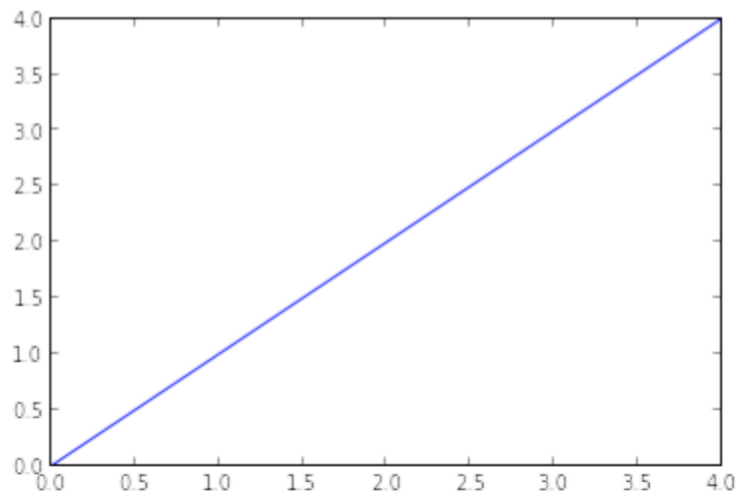
To begin, start iPython interface in Inline Pylab mode by typing following on your terminal / windows command prompt:

```
ipython notebook --pylab=inline
```

This opens up iPython notebook in pylab environment, which has a few useful libraries already imported. Also, you will be able to plot your data inline, which makes this a really good environment for interactive data analysis. You can check whether the environment has loaded

correctly, by typing the following command (and getting the output as seen in the figure below):

```
plot(arange(5))
```



Importing libraries and the data set:

Following are the libraries we will use during this tutorial:

- numpy
- matplotlib
- pandas

Please note that you do not need to import matplotlib and numpy because of Pylab environment. I have still kept them in the code, in case you use the code in a different environment.

After importing the library, you read the dataset using function `read_csv()`. This is how the code looks like till this stage:

```
import pandas as pd
import numpy as np
import matplotlib as plt

df = pd.read_csv("/home/kunal/Downloads/Loan_Prediction/train.csv") #Reading the dataset
in a dataframe using Pandas
```

Quick Data Exploration

Once you have read the dataset, you can have a look at few top rows by using the function `head()`

```
df.head(10)
```

In [3]: `df.head(10)` #Printing first 10 rows of dataset

Out[3]:

	Loan_ID	Gender	Married	Dependents	Education	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Cr
0	LP001002	Male	No	0	Graduate	No	5849	0	NaN	360	1
1	LP001003	Male	Yes	1	Graduate	No	4583	1508	128	360	1
2	LP001005	Male	Yes	0	Graduate	Yes	3000	0	66	360	1
3	LP001006	Male	Yes	0	Not Graduate	No	2583	2358	120	360	1
4	LP001008	Male	No	0	Graduate	No	6000	0	141	360	1
5	LP001011	Male	Yes	2	Graduate	Yes	5417	4196	267	360	1
6	LP001013	Male	Yes	0	Not Graduate	No	2333	1516	95	360	1
7	LP001014	Male	Yes	3+	Graduate	No	3036	2504	158	360	0
8	LP001018	Male	Yes	2	Graduate	No	4006	1526	168	360	1
9	LP001020	Male	Yes	1	Graduate	No	12841	10968	349	360	1

This should print 10 rows. Alternately, you can also look at more rows by printing the dataset.

Next, you can look at summary of numerical fields by using describe () function

```
df.describe()
```

```
In [4]: df.describe() #Get summary of numerical variables
```

	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History
count	614.000000	614.000000	592.000000	600.000000	564.000000
mean	5403.459283	1621.245798	146.412162	342.000000	0.842199
std	6109.041673	2926.248369	85.587325	65.12041	0.364878
min	150.000000	0.000000	9.000000	12.00000	0.000000
25%	2877.500000	0.000000	100.000000	360.00000	1.000000
50%	3812.500000	1188.500000	128.000000	360.00000	1.000000
75%	5795.000000	2297.250000	168.000000	360.00000	1.000000
max	81000.000000	41667.000000	700.000000	480.00000	1.000000

describe () function would provide count, mean, standard deviation (std), min, quartiles and max in its output (Read this article (<https://www.analyticsvidhya.com/blog/2014/07/statistics/>) to refresh basic statistics to understand population distribution)

Here are a few inferences, you can draw by looking at the output of describe () function:

1. LoanAmount has (614 – 592) 22 missing values.
2. Loan_Amount_Term has (614 – 600) 14 missing values.
3. Credit_History has (614 – 564) 50 missing values.
4. We can also look that about 84% applicants have a credit_history. How? The mean of Credit_History field is 0.84 (Remember, Credit_History has value 1 for those who have a credit history and 0 otherwise)
5. The ApplicantIncome distribution seems to be in line with expectation. Same with Co applicant Income

Please note that we can get an idea of a possible skew in the data by comparing the mean to the median, i.e. the 50% figure.

For the non-numerical values (e.g. Property_Area, Credit_History etc.), we can look at frequency distribution to understand whether they make sense or not. The frequency table can be printed by following command:

```
df['Property_Area'].value_counts()
```

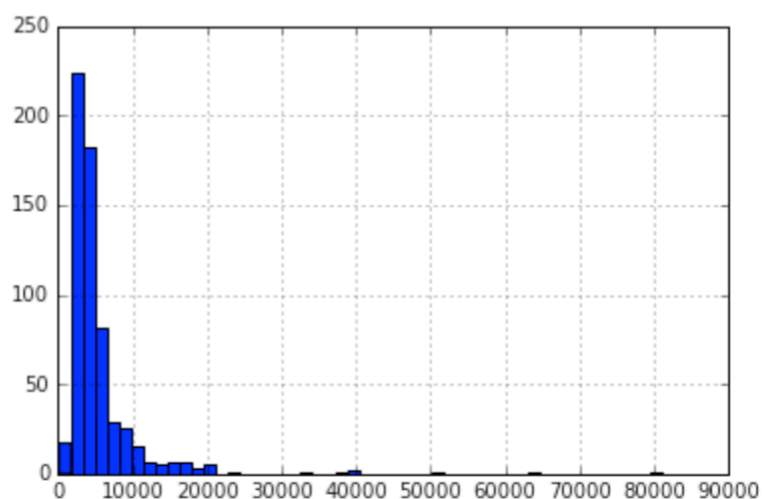
Similarly, we can look at unique values of port of credit history. Note that `dfname['column_name']` is a basic indexing technique to access a particular column of the data frame. It can be a list of columns as well. For more information, refer to the “10 Minutes to Pandas” resource shared above.

Distribution analysis

Now that we are familiar with basic data characteristics, let us study distribution of various variables. Let us start with numeric variables – namely ApplicantIncome and LoanAmount

Let's start by plotting the histogram of ApplicantIncome using the following commands:

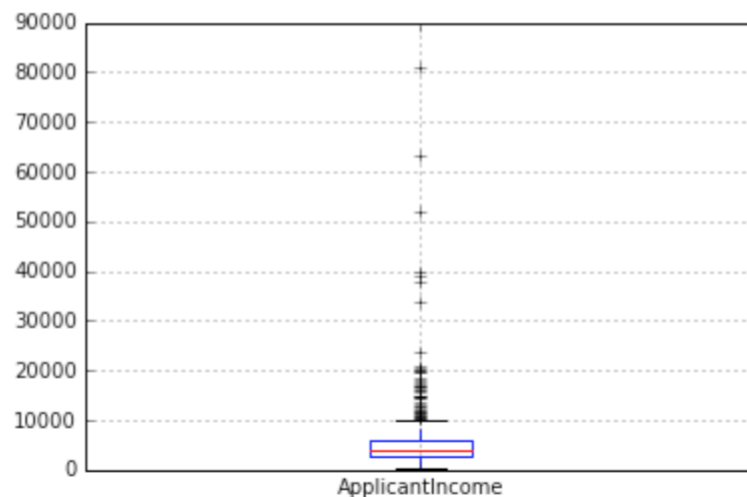
```
df['ApplicantIncome'].hist(bins=50)
```



Here we observe that there are few extreme values. This is also the reason why 50 bins are required to depict the distribution clearly.

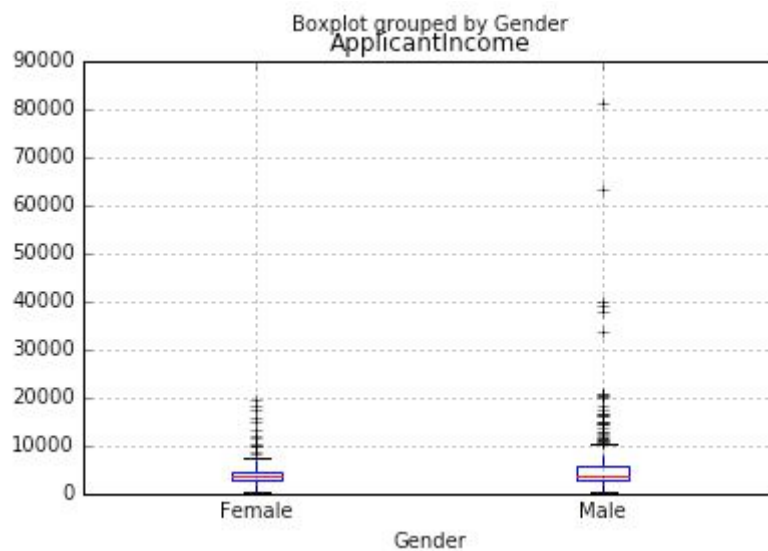
Next, we look at box plots to understand the distributions. Box plot for fare can be plotted by:

```
df.boxplot(column='ApplicantIncome')
```



This confirms the presence of a lot of outliers/extreme values. This can be attributed to the income disparity in the society. Part of this can be driven by the fact that we are looking at people with different education levels. Let us segregate them by Education:

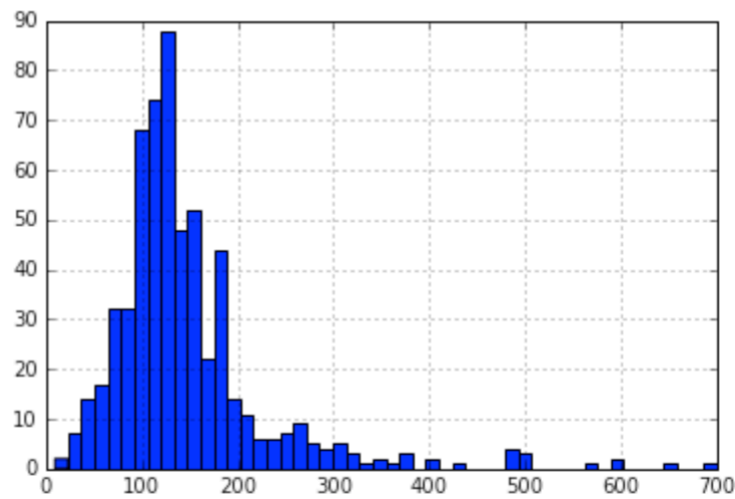
```
df.boxplot(column='ApplicantIncome', by = 'Education')
```



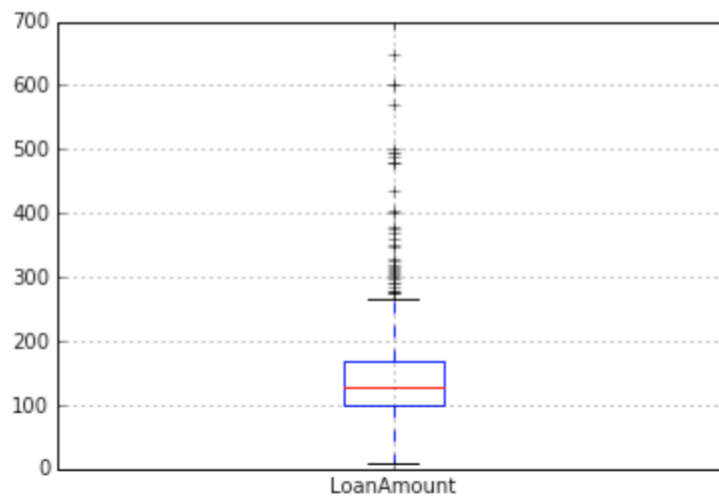
We can see that there is no substantial different between the mean income of graduate and non-graduates. But there are a higher number of graduates with very high incomes, which are appearing to be the outliers.

Now, let's look at the histogram and boxplot of LoanAmount using the following command:

```
df['LoanAmount'].hist(bins=50)
```



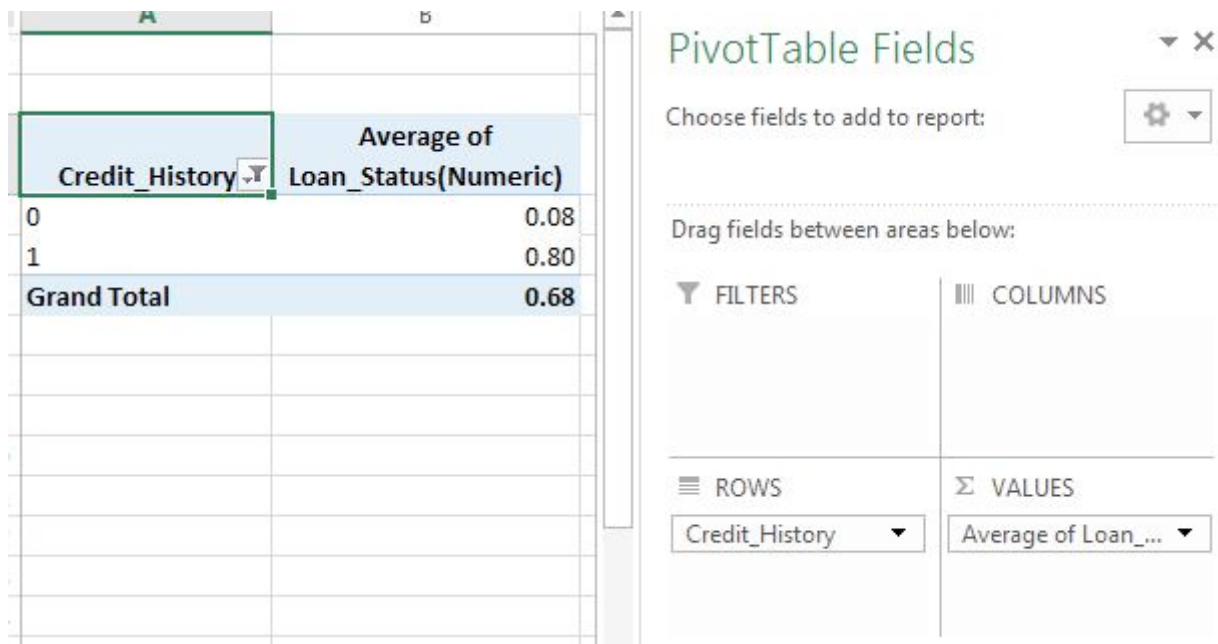
```
df.boxplot(column='LoanAmount')
```



Again, there are some extreme values. Clearly, both ApplicantIncome and LoanAmount require some amount of data munging. LoanAmount has missing and well as extreme values, while ApplicantIncome has a few extreme values, which demand deeper understanding. We will take this up in coming sections.

Categorical variable analysis

Now that we understand distributions for ApplicantIncome and LoanIncome, let us understand categorical variables in more details. We will use Excel style pivot table and cross-tabulation. For instance, let us look at the chances of getting a loan based on credit history. This can be achieved in MS Excel using a pivot table as:



Credit_History	Average of Loan_Status(Numeric)
0	0.08
1	0.80
Grand Total	0.68

Note: here loan status has been coded as 1 for Yes and 0 for No. So the mean represents the probability of getting loan.

Now we will look at the steps required to generate a similar insight using Python. Please refer to this article (<https://www.analyticsvidhya.com/blog/2016/01/12-pandas-techniques-python-data-manipulation/>) for getting a hang of the different data manipulation techniques in Pandas.

```
temp1 = df['Credit_History'].value_counts(ascending=True)
temp2 = df.pivot_table(values='Loan_Status', index=['Credit_History'], aggfunc=lambda x:
x.map({'Y':1, 'N':0}).mean())
print 'Frequency Table for Credit History:'
print temp1

print '\nProbability of getting loan for each Credit History'
class: print temp2
```

Frequency Table for Credit History:

0 89

1 475

Name: Credit_History, dtype: int64

Probability of getting loan for each Credit History class:

Credit_History

0 0.078652

1 0.795789

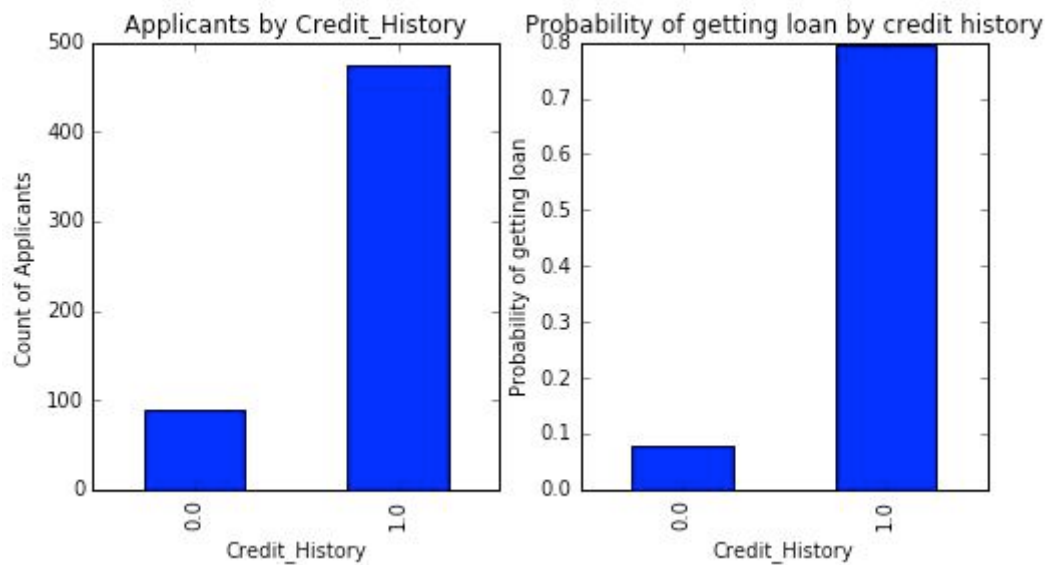
Name: Loan_Status, dtype: float64

(https://www.analyticsvidhya.com/wp-content/uploads/2016/01/11.-pivot_python.png)

Now we can observe that we get a similar pivot_table like the MS Excel one. This can be plotted as a bar chart using the “matplotlib” library with following code:

```
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(8,4))
ax1 = fig.add_subplot(121)
ax1.set_xlabel('Credit_History')
ax1.set_ylabel('Count of Applicants')
ax1.set_title("Applicants by Credit_History")
temp1.plot(kind='bar')

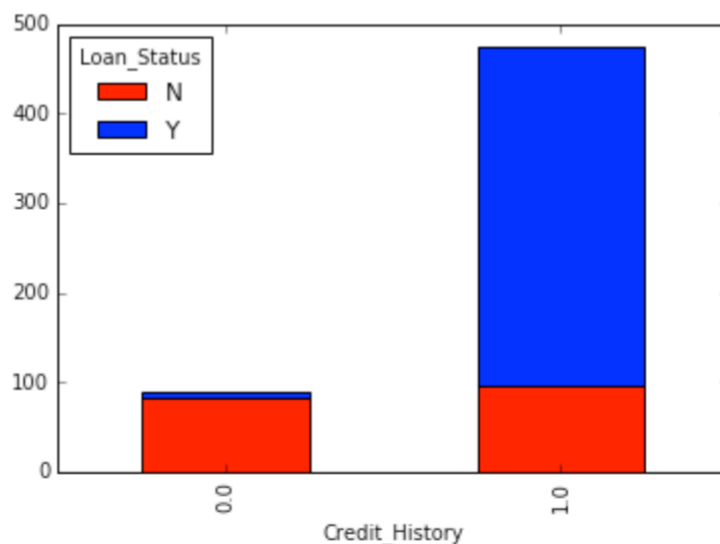
ax2 = fig.add_subplot(122)
temp2.plot(kind = 'bar')
ax2.set_xlabel('Credit_History')
ax2.set_ylabel('Probability of getting loan')
ax2.set_title("Probability of getting loan by credit history")
```



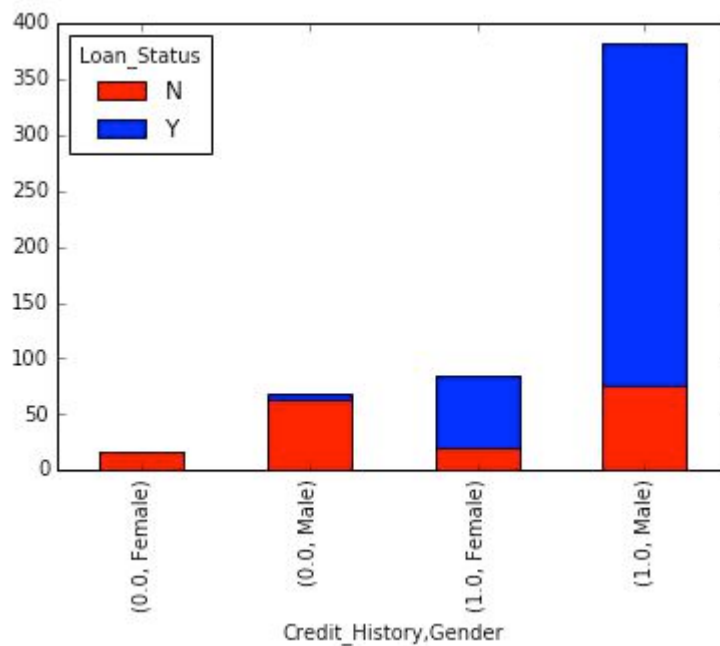
This shows that the chances of getting a loan are eight-fold if the applicant has a valid credit history. You can plot similar graphs by Married, Self-Employed, Property_Area, etc.

Alternately, these two plots can also be visualized by combining them in a stacked chart::

```
temp3 = pd.crosstab(df['Credit_History'], df['Loan_Status'])
temp3.plot(kind='bar', stacked=True, color=['red','blue'], grid=False)
```



You can also add gender into the mix (similar to the pivot table in Excel):



If you have not realized already, we have just created two basic classification algorithms here, one based on credit history, while other on 2 categorical variables (including gender). You can quickly code this to create your first submission on AV Datahacks.

We just saw how we can do exploratory analysis in Python using Pandas. I hope your love for pandas (the animal) would have increased by now – given the amount of help, the library can provide you in analyzing datasets.

Next let's explore ApplicantIncome and LoanStatus variables further, perform data munging (<https://www.analyticsvidhya.com/blog/2014/09/data-munging-python-using-pandas-baby-steps-python/>) and create a dataset for applying various modeling techniques. I would strongly urge that you take another dataset and problem and go through an independent example before reading further.

4. Data Munging in Python: Using Pandas

Data munging – recap of the need

While our exploration of the data, we found a few problems in the data set, which needs to be solved before the data is ready for a good model. This exercise is typically referred as “Data Munging”. Here are the problems, we are already aware of:

1. There are missing values in some variables. We should estimate those values wisely depending on the amount of missing values and the expected importance of variables.
2. While looking at the distributions, we saw that ApplicantIncome and LoanAmount seemed to contain extreme values at either end. Though they might make intuitive sense, but should be treated appropriately.

In addition to these problems with numerical fields, we should also look at the non-numerical fields i.e. Gender, Property_Area, Married, Education and Dependents to see, if they contain any useful information.

If you are new to Pandas, I would recommend reading this article (<https://www.analyticsvidhya.com/blog/2016/01/12-pandas-techniques-python-data-manipulation/>) before moving on. It details some useful techniques of data manipulation.

Check missing values in the dataset

Let us look at missing values in all the variables because most of the models don't work with missing data and even if they do, imputing them helps more often than not. So, let us check the number of nulls / NaNs in the dataset

```
df.apply(lambda x: sum(x.isnull()),axis=0)
```

This command should tell us the number of missing values in each column as isnull() returns 1, if the value is null.


```
In [14]: df.apply(lambda x: sum(x.isnull()),axis=0)

Out[14]: Loan_ID          0
        Gender          13
        Married         3
        Dependents      15
        Education        0
        Self_Employed    32
        ApplicantIncome  0
        CoapplicantIncome 0
        LoanAmount       22
        Loan_Amount_Term 14
        Credit_History    50
        Property_Area     0
        Loan_Status       0
        dtype: int64
```

Though the missing values are not very high in number, but many variables have them and each one of these should be estimated and added in the data. Get a detailed view on different imputation techniques through this article (<https://www.analyticsvidhya.com/blog/2016/01/guide-data-exploration/>).

Note: Remember that missing values may not always be NaNs. For instance, if the Loan_Amount_Term is 0, does it makes sense or would you consider that missing? I suppose your answer is missing and you're right. So we should check for values which are unpractical.

How to fill missing values in LoanAmount?

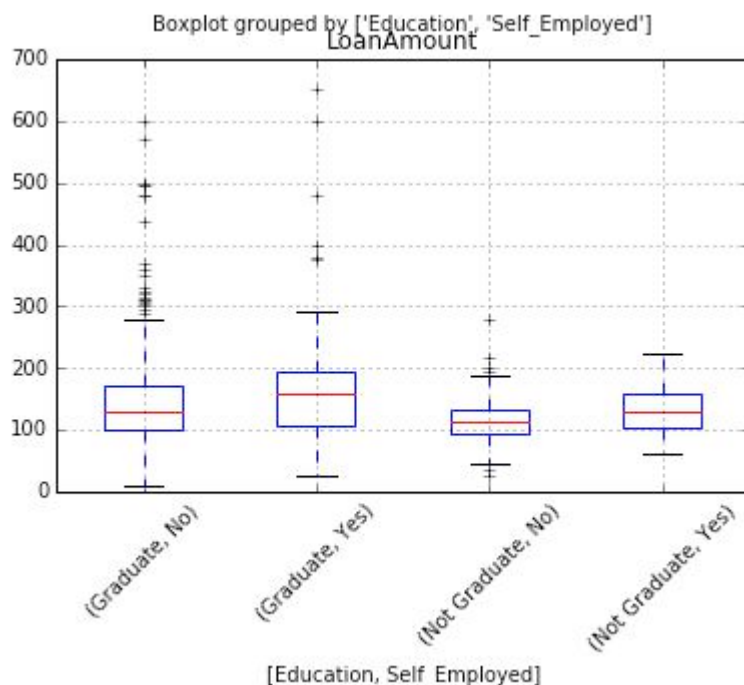
There are numerous ways to fill the missing values of loan amount – the simplest being replacement by mean, which can be done by following code:

```
df['LoanAmount'].fillna(df['LoanAmount'].mean(), inplace=True)
```

The other extreme could be to build a supervised learning model to predict loan amount on the basis of other variables and then use age along with other variables to predict survival.

Since, the purpose now is to bring out the steps in data munging, I'll rather take an approach, which lies somewhere in between these 2 extremes. A key hypothesis is that the whether a person is educated or self-employed can combine to give a good estimate of loan amount.

First, let's look at the boxplot to see if a trend exists:



Thus we see some variations in the median of loan amount for each group and this can be used to impute the values. But first, we have to ensure that each of Self_Employed and Education variables should not have a missing values.

As we say earlier, Self_Employed has some missing values. Let's look at the frequency table:

```
In [40]: df['Self_Employed'].value_counts()
Out[40]: No      500
         Yes      82
         Name: Self_Employed, dtype: int64
```

Since ~86% values are "No", it is safe to impute the missing values as "No" as there is a high probability of success. This can be done using the following code:

```
df['Self_Employed'].fillna('No',inplace=True)
```

Now, we will create a Pivot table, which provides us median values for all the groups of unique values of Self_Employed and Education features. Next, we define a function, which returns the values of these cells and apply it to fill the missing values of loan amount:

```
table = df.pivot_table(values='LoanAmount', index='Self_Employed' ,columns='Education',
aggfunc=np.median)

# Define function to return value of this
pivot_table def fage(x):
    return table.loc[x['Self_Employed'],x['Education']]

# Replace missing values
df['LoanAmount'].fillna(df[df['LoanAmount'].isnull()].apply(fage, axis=1), inplace=True)
```

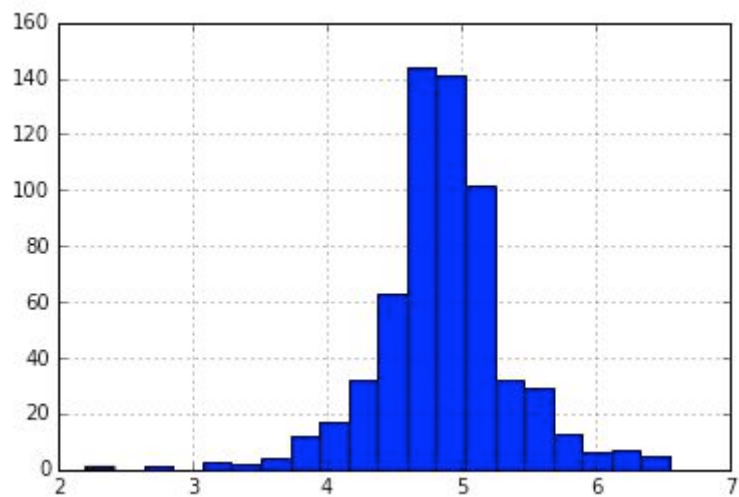
This should provide you a good way to impute missing values of loan amount.

How to treat for extreme values in distribution of LoanAmount and ApplicantIncome?

Let's analyze LoanAmount first. Since the extreme values are practically possible, i.e. some people might apply for high value loans due to specific needs. So instead of treating them as outliers, let's try a log transformation to nullify their effect:

```
df['LoanAmount_log'] = np.log(df['LoanAmount'])
df['LoanAmount_log'].hist(bins=20)
```

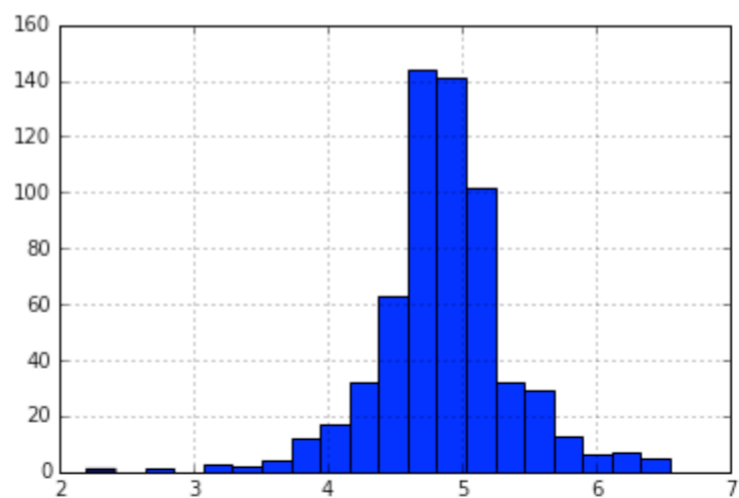
Looking at the histogram again:



Now the distribution looks much closer to normal and effect of extreme values has been significantly subsided.

Coming to ApplicantIncome. One intuition can be that some applicants have lower income but strong support Co-applicants. So it might be a good idea to combine both incomes as total income and take a log transformation of the same.

```
df['TotalIncome'] = df['ApplicantIncome'] +
df['CoapplicantIncome']
df['TotalIncome_log'] =
np.log(df['TotalIncome'])
df['LoanAmount_log'].hist(bins=20)
```



Now we see that the distribution is much better than before. I will leave it up to you to impute the missing values for Gender, Married, Dependents, Loan_Amount_Term, Credit_History. Also, I encourage you to think about possible additional information which can be derived from the data. For example, creating a column for LoanAmount/TotalIncome might make sense as it gives an idea of how well the applicant is suited to pay back his loan.

Next, we will look at making predictive models.

5. Building a Predictive Model in Python

After, we have made the data useful for modeling, let's now look at the python code to create a predictive model on our data set. Skicit-Learn (sklearn) is the most commonly used library in Python for this purpose and we will follow the trail. I encourage you to get a refresher on sklearn through this article (<https://www.analyticsvidhya.com/blog/2015/01/scikit-learn-python-machine-learning-tool/>).

Since, sklearn requires all inputs to be numeric, we should convert all our categorical variables into numeric by encoding the categories. This can be done using the following code:

```
from sklearn.preprocessing import LabelEncoder

var_mod = ['Gender', 'Married', 'Dependents', 'Education', 'Self_Employed', 'Property_Area', 'Loan_
Status']

le = LabelEncoder()
for i in var_mod:
    df[i] = le.fit_transform(df[i])
df.dtypes
```

Next, we will import the required modules. Then we will define a generic classification function, which takes a model as input and determines the Accuracy and Cross-Validation scores. Since this is an introductory article, I will not go into the details of coding. Please refer to this article (<https://www.analyticsvidhya.com/blog/2015/08/common-machine-learning-algorithms/>) for getting details of the algorithms with R and Python codes. Also, it'll be good to get a refresher on cross-validation through this article (<https://www.analyticsvidhya.com/blog/2015/11/improve-model-performance-cross-validation-in-python-r/>), as it is a very important measure of power performance.

```
#Import models from scikit learn module:
from sklearn.linear_model import LogisticRegression
from sklearn.cross_validation import KFold #For K-fold cross validation
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier, export_graphviz
from sklearn import metrics

#Generic function for making a classification model and accessing performance:
def classification_model(model, data, predictors, outcome):
    #Fit the model:
    model.fit(data[predictors],data[outcome])

    #Make predictions on training set:
    predictions = model.predict(data[predictors])

    #Print accuracy
    accuracy = metrics.accuracy_score(predictions,data[outcome])
    print "Accuracy : %s" % "{0:.3%}".format(accuracy)

    #Perform k-fold cross-validation with 5 folds
    kf = KFold(data.shape[0], n_folds=5)
    error = []
    for train, test in kf:
        # Filter training data
        train_predictors = (data[predictors].iloc[train,:])

        # The target we're using to train the algorithm.
        train_target = data[outcome].iloc[train]

        # Training the algorithm using the predictors and
        target. model.fit(train_predictors, train_target)

        #Record error from each cross-validation run
        error.append(model.score(data[predictors].iloc[test:], data[outcome].iloc[test]))
```

```
print "Cross-Validation Score : %s" % "{0:.3%}".format(np.mean(error))

#Fit the model again so that it can be refered outside the function:
model.fit(data[predictors],data[outcome])
```

End Notes

I hope this tutorial will help you maximize your efficiency when starting with data science in Python. I am sure this not only gave you an idea about basic data analysis methods but it also showed you how to implement some of the more sophisticated techniques available today.

Python is really a great tool, and is becoming an increasingly popular language among the data scientists. The reason being, it's easy to learn, integrates well with other databases and tools like Spark and Hadoop. Majorly, it has great computational intensity and has powerful data analytics libraries.

So, learn Python to perform the full life-cycle of any data science project. It includes reading, analyzing, visualizing and finally making predictions.