Python for Data Science

Why Python

- Python is an open source, high-level, object-oriented, interpreted, and general-purpose dynamic programming language.
- It has an easy and clean syntax with simple semantics. Similar to C, Algol but it is also an OOP language; OOP is not mandatory (unlike CPP). No extra symbol to start or end a statement (no @, $ symbol like in perl);
- Python finds its applications across various areas such as website development, mobile apps development, scientific and numeric
- computing, desktop GUI, and complex software development, machine learning and data science
- Development environments such as Ipython Notebook and Spyder that allow for a quick introspection of the data and enable developing of machine learning models interactively.
- No special character at the end (like ; in C).
- The main reasons being:

✓ no declaration required. It is a high-level language

✓ many built-in data structures are already available: dictionary, lists...

✓ no need for memory handling: there is a memory garbage collector

- Python can be run, debugged and tested interactively in the python interpreter.
- Ideal for web scripting and data handling.
- There is a rich set of external libraries especially for science (matplotlib, pandas, numpy,Scikit,SciPy etc.)

Drawbacks

- Indentation probably intimidates beginners but after a couple of hours of coding it becomes natural.
- Slower than compiled languages.

Python-Setup Installing Python

The most up-to-date and current source code, binaries, documentation, news, etc., is available on the official website of Python:
http://www.python.org

Windows Installation

- Here are the steps to install Python on Windows machine.
- Open a Web browser and go to **http://www.python.org/download/**
- Follow the link for the Windows installer *python-XYZ.msi* file where XYZ is the version you need to install.
- To use this installer *python-XYZ.msi*, the Windows system must support Microsoft Installer 2.0. Save the installer file to your local machine and then run it to find out if your machine supports MSI.
- Run the downloaded file. This brings up the Python install wizard, which is really easy to use. Just accept the default settings, wait until the install is finished, and you are done.

- **Unix and Linux Installation**
- Here are the simple steps to install Python on Unix/Linux machine.
- Open a Web browser and go to **http://www.python.org/download/**.
- Follow the link to download zipped source code available for Unix/Linux.
- Download and extract files.
- Editing the *Modules/Setup* file if you want to customize some options.
- **run** ./configure script
- make
- make install

- This installs Python at standard location */usr/local/bin* and its libraries at */usr/local/lib/pythonXX* where XX is the version of Python.

Setting Path

To add the Python directory to the path for a particular session in Windows:

At the command prompt: type path %path%;C:\Python and press Enter.

Note: C:\Python is the path of the Python directory

To add the Python directory to the path for a particular session in Unix:

- **In the csh shell:** type setenv PATH "$PATH:/usr/local/bin/python" and press Enter.
- **In the bash shell (Linux):** type export ATH="$PATH:/usr/local/bin/python" and press Enter.
- **In the sh or ksh shell:** type PATH="$PATH:/usr/local/bin/python" and press Enter.
- **Note:** /usr/local/bin/python is the path of the Python directory

Running Python

- Three different ways to start Python:

1. Interactive Interpreter:

Start Python from Unix, DOS, or any other system that provides you a command-line interpreter or shell window.

Enter **python** the command line.

Start coding right away in the interactive interpreter.

2. Script from the Command-line:

A Python script can be executed at command line by invoking the interpreter on your application, as in the following:

```
$python  script.py              # Unix/Linux          or
python% script.py               # Unix/Linux          or
C:>python script.py             # Windows/DOS
```

Note: Be sure the file permission mode allows execution.

3. Integrated Development Environment:

Run Python from a Graphical User Interface (GUI) environment as well, if you have a GUI application on your system that supports Python.

- **Unix:** IDLE is the very first Unix IDE for Python.
- **Windows:** PythonWin is the first Windows interface for Python and is an IDE with a GUI.
- **Macintosh:** The Macintosh version of Python along with the IDLE IDE is available from the main website, downloadable as either MacBinary or BinHex'd files.

Using Python for Machine Learning

- Python is one of the most popular programming languages for data science and therefore enjoys a large number of useful add-on libraries developed by its great community.
- Being an interpreted high-level programming language, it may seem that Python was designed specifically for the process of trying out different things.
- Although the performance of interpreted languages, such as Python, for computation-intensive tasks is inferior to lower-level programming languages, extension libraries such as *NumPy* and *SciPy* have been developed that build upon lower layer Fortran and C implementations for fast and vectorized operations on multidimensional arrays.
- *scikit-learn* library, is one of the most popular and accessible open source machine learning libraries available in Python.

Installing Python

Python is available for all three major operating systems—Microsoft Windows, Mac OS X, and Linux—and the installer, as well as the documentation, can be downloaded from the official Python website: https://www.python.org.

Python Variants

Python is available in 2.x and 3.x variants

A good summary about the differences between Python 3.4 and 2.7 can be found at https://wiki.python.org/moin/Python2orPython3.

- The Anaconda installer can be downloaded at http://continuum.io/downloads#py34, or https://www.anaconda.com/download/
- Anaconda quick start-guide is available at https://store.continuum.io/static/img/Anaconda-Quickstart.pdf.

Python's frequently used libraries

1.NumPy---------- Used to store and maniupluate data

2.Pandas---------- Built on top of NumPy and provides higher level data manipulation tools that make working with tabular data more convenient.

3.Matplotlib---------- To augment the learning experience and visualize quantitative data

4.SciPy----------Lots of numerical analysis tools

5.networkx---------- Playing with graph

6.Igraph ---------- Playing with graph

Other major libraries include:

Scikit learm,SciPy

Note:

Make sure that the version numbers of your installed packages are equal to, or greater than, those version numbers to ensure the code examples run correctly

- NumPy 1.9.1
- SciPy 0.14.0
- scikit-learn 0.15.2
- matplotlib 1.4.0
- pandas 0.15.2

Basics of Python

Identifiers

- Names used to identify a variable, function, class, module or other object.
- Identifiers start with a letter A to Z or an underscore(_) followed by zero or more letters, underscores and digits(0 to 9)
- Punctuation characters like @,$ and % cannot be used with identifiers.
- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Identifier with single leading underscore indicates that it is a private identifier.
- Identifier with two leading underscores indicates a strongly private identifier.

Keywords

- And exec Not
- Assert finally or
- Break for pass
- Class from print
- Continue global raise
- def if return
- del import try
- elif in while
- else is with
- except lambda yield

- Blocks of code are denoted by line indentation and Python provides no braces to indicate the same.

Ex:

```
if True:
        print "True"
else:
        print "False"
```

Note: Lines indented with same number of spaces would form a block

- Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example:

```
total = item_one + \
        item_two + \
        item_three
```

Statements contained within the [], {}, or () brackets do not need to use the line continuation character.

```
days = ['Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday']
```

- Python accepts single ('), double (") and triple ('" or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.
- The triple quotes are used to span the string across multiple lines.

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

- A hash sign(#) indicates a comment .
- ‘\n’ creates a new line
- ‘;’ allows multiple statements on a single line.

Suites in Python

- A group of individual statements, which make a single code block are called **suites** in Python. Compound or complex statements, such as if, while, def, and class require a header line and a suite.
- Header lines begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines which make up the suite.

```
if expression :
                    suite
elif expression :
                    suite
else :
                    suite
```

Variable Types

- Python uses ‘=‘ sign to assign values to variables
- Variable declaration happens automatically when a value is assigned to it.

```
Ex: a = 10                    # integer assignment
    b = 5.3                   # floating point
    c = “Hello Python”        # A string
    a = b= c = 100            # Multiple Assignment
```

Working with Python

- Python provides a shell, however, we strongly recommend to install ipython instead, which provides nice features such as tab completion.
- In interactive mode, the prompt is represented by the >>> signs
- Python codes are preceded by the >>> signs and outputs are shown without.

- Important points to remember while working with Python:

1. Python is a zero indexing language
2. No ';' character at the end of a line, just a return carriage
3. Indentation is important and used to separate code blocks
4. Everything is an object

Data Structures in Python

The built in data structures in Python include :

- Lists,
- Tuples,
- Dictionaries,
- Strings,
- Sets and
- Frozensets.

✓ Lists, strings and tuples are ordered sequences of objects.

✓ Unlike strings that contain only characters, list and tuples can contain any type of objects.

✓ Lists and tuples are like arrays.

✓ Tuples like strings are immutables.

✓ Lists are mutables so they can be extended or reduced at will.

✓ Sets are mutable unordered sequence of unique elements whereas frozensets are immutable sets.

- Most versatile data type available in Python which can be written as a list of comma separated values (items) between square brackets.
- Items in a list need not be of the same type:

Ex:

>>>l1 =[1,2,3]

To access list elements, use list name or list name followed by index

>>>l1

>>>l1[0]

>>>l2=[1,2,"a"]

>>>l2

>>>l2[0]

To append elements to a list

>>>l1.append(4)

>>>l1

[1,2,3,4]

Update existing value of list

l1[2]=5

Difference between append() and extend()

>>>list1 = ['a','b']

>>>list1.append('c')

>>>list1

['a','b','c']

To extend 'list1' to contain more elements, 'extend' method can be used:

>>>list1.extend(['d', 'e', 'f'])

>>>list1

['a', 'b', 'c', 'd', 'e', 'f']

Search for an element in the list using index() function

>>>my_list = ['a', 'b', 'c', 'd', 'e']

>>>my_list.index('b')

1

>>>my_list1 =['a', 'b' , 'c', 'd', 'b', 'c']

>>>my_list1.index('b')

1

>>>my_list1.index('b',2)

4

- To insert element wherever into a list, use insert()

>>>my_list.insert(2, 'a')

>>>my_list

insert() method inserts an object before the index provided

- To remove first occurrence of an element, use remove()

Ex:

If my_list has elements as a,b,c,d,e,a,b then

>>>my_list.remove('a') will result in

>>>my_list

['b', 'c', 'd', 'e', 'a', 'b']

Ex:

- To remove the last element of the list use pop()

>>>my_list.pop()

Removes the last element

>>>my_list = ['a', 'b', 'c', 'd']

>>>my_list.pop()

- To count the number of elements of a kind use count()

Ex:

>>>my_list = ['a', 'b', 'c', 'd', 'a', 'b']

>>>my_list. count('b')

2

- To perform in-place sorting use sort()

>>>my_list = ['c', 'a', 'd', 'b', 'e']

>>>my_list.sort()

>>>my_list

['a', 'b', 'c', 'd', 'e']

To sort in reverse order use 'reverse=true'

>>>my_list.sort(reverse=True)

>>>my_list

- To reverse the list

>>>my_list.reverse = ['a', 'b', 'c']

>>>my_list.reverse()

>>>my_list

- **Slicing the list**

Slicing uses the symbol *:* to access to part of a list:

```
>>>list[first index: last index: step]
>>>list[:]
>>>a =[0,1,2,3,4,5]
>>>a
[0,1,2,3,4,5]
>>>a[2:]
[2,3,4,5]
>>>a[:2]
[0,1]
>>>a[2:-1]
[2,3,4]
>>>a[:]
[0,1,2,3,4,5]
```

- **List Comprehension**

```
>>>evens = []
>>> for i in range(10):
        if i % 2 == 0:
            evens.append(i)
>>>evens
[0,2,4,6,8]
```

Rather than the above code

```
>>>[i for I in range(10) if i % 2 == 0]
[0,2,4,6,8]
```

- In Python, **tuples** are part of the standard language.
- Very similar to the **list** data structure.
- Tuples are faster and consume less memory
- The main difference being that tuple manipulation are faster than list because tuples are immutable.
- Tuples are enclosed in parentheses

Ex:

```
>>>t = (1,2,"a")
>>>l=(1,2,3)
>>>l[0]
```

- Tuples can also be created without parentheses, by using commas:

```
>>>l = 1,2
>>>l
(1,2)
```

- To create a tuple with single element:

```
>>>singleton =(1, )
```

Tuple can be repeated by multiplying by a number

```
>>>(1,) * 5
(1,1,1,1,1)
```

- **Concatenate tuples**

```
>>>s1 = (1,0)
>>> s1 += (1,)
>>>s1
(1, 0, 1)
```

- Tuples can be indexed and elements in tuple can be counted

```
>>> l = (1,2,3,1)
>>>l.count(1)
2
>>>l.index(2)
1
```

- Signing multiple values

```
>>> (x,y,z) = ('a', 'b', 'c')
>>>x
'a'
```

- Tuples can be unpacked

```
>>>data =(1,2,3)
>>>x, y, z = data
>>>x
1
```

- Tuples can be swapped

```
>>>(x,y)=(1,2)
>>>(x,y) = (y,x)
```

- To find length of the tuple:

```
>>>t=(1,2)
>>>len(t)
2
```

- Slicing a tuple

```
>>>t=(1,2,3,4,5)
>>>t[2:]
(3,4,5)
```

- Copying a tuple

```
>>>t=(1,2,3,4,5)
>>>newt = t
>>>newt
(1,2,3,4,5)
```

- Mathematical operations

```
>>>t =(1,2,3)
>>>max(t)
3
```

- A dictionary is a sequence of items. Each item is a pair made of a key and a value.
- Dictionaries are not sorted. You can access to the list of keys or values independently.
- Items in a dictionary are not sorted
- Dictionaries are built with curly brackets:

Ex:

```
>>>d = {“a”:1, ”b”:2}
>>>d.keys()
[‘a’, ‘b’]
>>>d.values()
[1,2]
```

- To query information in dictionary, use item() method

Ex:

```
>>>d = {“a”:1, ”b”:2}
>>>d.items()
[(‘a’,1),(‘b’,2)]
```

- To check the existence of a specific key use ‘has_key’

```
>>>d.has_key(‘a’)
True
```

- To get value corresponding to a key

```
>>>d.get(‘a’)                    #pop can also be used
1
```

Pop removes the corresponding item from dictionary

- To remove an item(i.e. key-value pair) use popitem() method

```
>>>d1={“a”:1,”b”:2,”c”:3}
>>>d1.popitem()
```

- To clear elements in a dictionary use clear() method

```
>>>d1.clear()
{}
```

- To delete only one item from the

- To create a dictionary from an existing or empty dictionary:

>>>d2.fromkeys([‘b’, ’c’])

Or

>>>{}.fromkeys([‘a’, ‘b’])

Note:

Default values for the keys would be ‘None’

- Combining dictionaries

>>> d1 ={‘a’:1}

>>>d2={‘a’:1,’b’:2}

>>>d1.update(d2)

>>>d1

>>>d1[‘a’]

>>>d2[‘a’]

- Strings

Strings are immutable sequence of characters.

To define strings we can use”

1.Single quotes

2.Double quotes

3.Triple quotes

Latest string convention is triple quotes

Ex:

>>>text =“””Welcome to Python”””

>>>text

‘Welcome to Python’

Triple quotes allow to specify multi-line strings while double and single quotes allow to insert single quote character.

- To access any character in a string use slicing

>>>text[0]

Will return the first character from the string at index 0

>>>text[-1]

Will return the first character from the end of the string or simply returns the last character in the string.

>>>text[0:]

Will return the complete string starting at index 0

- Characters in given string cannot be changed

Ex:

>>>text[0]='a'

Will return an error

- To produce formatted output use '%' symbol

>>>print("%s" % "Welcome")

Welcome

- Operators on Strings

Mathematical operators '+' and '*' can be used to create new strings :

>>>t1 = """Welcome"

>>>t2="""to"""

>>>t3="""Pyhton"""

>>>t4=t1+t2+t3

To get spaces between strings

>>>t4 = t1 + " " + t2 + " " +t3

>>>t4

- Methods on Strings

Few methods are available to check the type of alpha numeric characters present in a string:

>>>"123".isdigit() ----- returns True

>>>"abc".isalpha()----- returns True

>>>"123".isalpha() ----- returns False

>>>"123abc".isalnum()---- returns True

>>>"aa".islower() ---- returns True

>>>"AA".isupper() ---- returns True

>>>"Aa".istitle()---- returns True

because string starts with

- Count the occurrence of a character in a string

>>>mystr =" This is a string"

Now to count the occurrences of the character 'i'

>>>mystr.count('i')

3

- To find length of a string

>>>len(mystr)

16

>>>mystr.title()

>>>mystr.capitalize()

>>>mystr.upper()

>>>mystr.lower()

>>>

>>>mystr.center(40)

>>>mystr.ljust(30)

>>>mystr.rjust(30,'-')

>>>mystr = " This is a string "

>>>mystr.find('is')

>>>mystr.index('is')

>>>mystr.split()

Sets

- Sets are constructed from a sequence (or some other iterable object).
- Sets cannot have duplicated, there are usually used to build sequence of unique items (e.g., set of identifiers).

```
>>>a= set([1,2,3,4,5])
>>>b = set([3,4,5,6])
>>>a | b    #Union
>>>a & b #Intersection
>>>a < b #Subset
>>>a – b #Difference
>>> a ^ b #Symmetric  Difference
>>>c = a.intersection(b)
>>>c
>>>c.issubset(a)
>>>c.issuperset(a)
```

Frozen sets

- *Sets* are mutable, and may therefore not be used, for example, as keys in dictionaries.
- Another problem is that sets themselves may only contain immutable (hashable) values, and thus may not contain other sets.
- Because sets of sets often occur in practice, there is the **frozenset** type, which represents immutable (and, therefore, hashable) sets.

```
>>>a=frozenset([1,2,3,4])
>>>b=frozenset([2,3,4,5])
>>>a.union(b)
frozenset([1,2,3,4,5])
>>>a=set([1,2,3])
>>>b=set([2,3,4])
>>>a.add(b)
Will return error
>>>a.add(frozenset(b))
```

- Using frozenset as key in a dictionary

```
>>>fa = {frozenset([1,2]):1}
>>fa[ frozenset([1,2]) ]
1
```

- To clear screen

```
>>>import os
>>>os.system('cls')
```

- To exit

```
>>>exit()
```

Or

Use Ctrl +z and press enter

```
>>>^z
```

Programming Fundamentals

Operators

Arithmetic:

+, - ,*,/, %, **(exponent), //(Floor Division)

Comparison:

==, !=, <>, >, <, >=, <=

Assignment:

= , +=, -=, *=, /=, %=, **=, //=

Logical :

and, or , not

Membership:

in, not in

Identity :

is, is not

Conditional Constructs: Decision Making

- Anticipating conditions occurring during program execution and specifying actions to be taken based on conditions.
- Actions to be executed are determined by evaluating expressions to TRUE or FALSE.
- Python programming language assumes any **non-zero** and **non-null** values as TRUE, and if it is either **zero** or **null**, then it is assumed as FALSE value.

If Statement

Contains a logical expression using which data is compared and a decision is made based on the result of the comparison.

Syntax:

```
if expression:
        statement(s)
```

If the expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed.

If expression evaluates to FALSE, then the first set of code after the end of the if statement(s) is executed.

Ex: >>>a = input("Enter a number") #reads input from console

>>>a = int(a) # input function returns user's response as a string which is to be converted to int.

```
>>>if (a > 5):
        print("a is greater than 5")
        print(a)
```

If… else statement

- Extension of if statement.
- An **else** statement contains the block of code that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.
- The *else* statement is an optional statement and there could be at most only one **else** statement following **if**.

Syntax:

```
        if expression:
                                statement(s)
        else:
                                 statement(s)
```

Ex: >>> n = input("Enter a number")

```
            Enter a number  10
>>>if n % 2 == 0:
             print (" even number")
        else:
             print("odd number")
```

elif statment

- Used to check multiple expressions for TRUE and executes a block of code.
- elif statement is optional.
- Any arbitrary number of elif statements can follow a if statement.

Syntax:

```
if expression1:
                statement(s)
    elif expression2:                #This structure simulates a switch case
                statement(s)
    elif expression3:
                statement(s)
    else:
                statement(s)
```

Iteration

- The for statement in Python has the ability to iterate over the items of any sequence, such as a list or a string.

Syntax:

```
for iterating_var in sequence:
    statements(s)
```

- If a sequence contains an expression list, it is evaluated first.
- Then, the first item in the sequence is assigned to the iterating variable *iterating_var*.
- Next, the statements block is executed.
- Each item in the list is assigned to *iterating_var*, and the statement(s) block is executed until the entire sequence is exhausted.
- range() function can be used to iterate over a sequence of numbers.
- range() generates an iterator to progress integers starting with 0

Ex:1

```
>>>for i in list(range(5)):
        print(i)
```

Output

0

1

2

3

4

Ex2:

```
>>> n=int(input("Enter Range")
Enter Range 5

>>> for i in list(range(n)):
        if i % 2 == 0:
            print(i)
    else:
        i = i + 1
```

Loop Control Statements

- The Loop control statements change the execution from its normal sequence. When the execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Control Statement	Description
break statement	Terminates the loop statement and transfers execution to the statement immediately following the loop.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
pass statement	The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

Examples

Break statement

```
>>> n = 10
>>> for i in range(n):
        if i<=5:
         print(i)
         break
```

Output

0

Continue statement

```
>>> n = 10
>>> for i in range(n)
        if i<=5:
         print(i)
        i = i + 1
block")
        continue
```

Output

0
1
2
3
4
5

pass statement

```
 >>> n = 10
>>> for i in range(n)
          if i == 3:
          pass
          print("pass
```

While Loop

- Executes a target statement as long as a given condition is true.

Syntax:

```
while expression:
    statement(s)
```

Ex: >>>count = 0

```
>>>while (count <10):
    print(count)
    count = count + 1
```

Ex:>>> count = 0

```
>>> while count<5:
    print(count, “is less than 5”)
    count = count + 1
else:
    print(count, “is not less than 5”)
```

Data Analysis using Python

Python Libraries

- Following libraries of Python are useful for scientific computations

Library	Purpose
NumPy	Numerical Python. Contains basic linear algebra functions, Fourier transforms, advanced random number capabilities and tools for integration with other languages.
SciPy	Scientific Python. It is built on NumPy .Used to perform discrete Fourier transform, Linear Algebra optimization and Sparse matrices.
Matplotlib	Used for plotting variety of graphs like histograms, line plots etc.
Pandas	Used for structured data operations and manipulations and used for data munging and preparation.
Scikit Learn	Used for machine learning. Built on NumPy, SciPy and Matplotlib. Contains tools for Machine learning and statistical modeling including classification, regression, clustering and dimensionality reduction
Statsmodels	Used for statistical modeling to explore data, estimate statistical models and perform statistical tests.
Seaborn	Used for statistical data visualization
Bokeh	Used for creating interactive plots, dashboards on modern web-browsers and large streaming datasets.
Blaze	Used to access data from multitude of sources likeMongoDB, SQLAlchemy, from huge chunks of data

Library	Purpose
Scrapy	Used for web crawling for getting specific patterns of data
SymPy	Used for symbolic computation including basic arithmetic to calculus, algebra, discrete mathematics etc.
os	Used for operating system and file operations
networkx and igraph	Used for graph based data manipulations
regular expressions	Used for finding patterns in text data

Key activities

- Data Exploration:

  Finding out more about data.

- Data Munging:

  Cleaning the data to make it better suitable for statistical modeling

- Predictive Modeling:

  Running the algorithms

Introduction to Jupyter Notebook

- Jupyter is and IDE used for working with Python.
- Note book environment such as Jupyter are very useful for a variety of purpose.
- Interactive and Scripting support
- Browser based.
- Supports all imports and exports
- Supports variety of data types within the same window such as text, code, graphs, videos, pictures
- Great for Visualizations
- Enables parallel computing

Importing dataset into Jupyter

- Before we import a dataset into Jupyter notebook:

1. Start Jupyter notebook using Windows Start button.
2. Click Anaconda.
3. Click Jupyter Notebook. Now Jupyter notebook opens.
4. Click New button and Click Pyhton2/Python3. A new notebook appears.
5. To see the current working directory use the command 'pwd'*(print working directory)* and use Shift + Enter keys. Now the current working directory is displayed.
6. To change the working directory, use jupyter magic command as below:

   Method1: %cd "C:\Users\pc\Desktop\Data Sets"

   Method2 : import os

   os.getcwd()

   os.chdir("C:/Users/pc/Desktop/Data Sets")

Machine Learning Python Packages

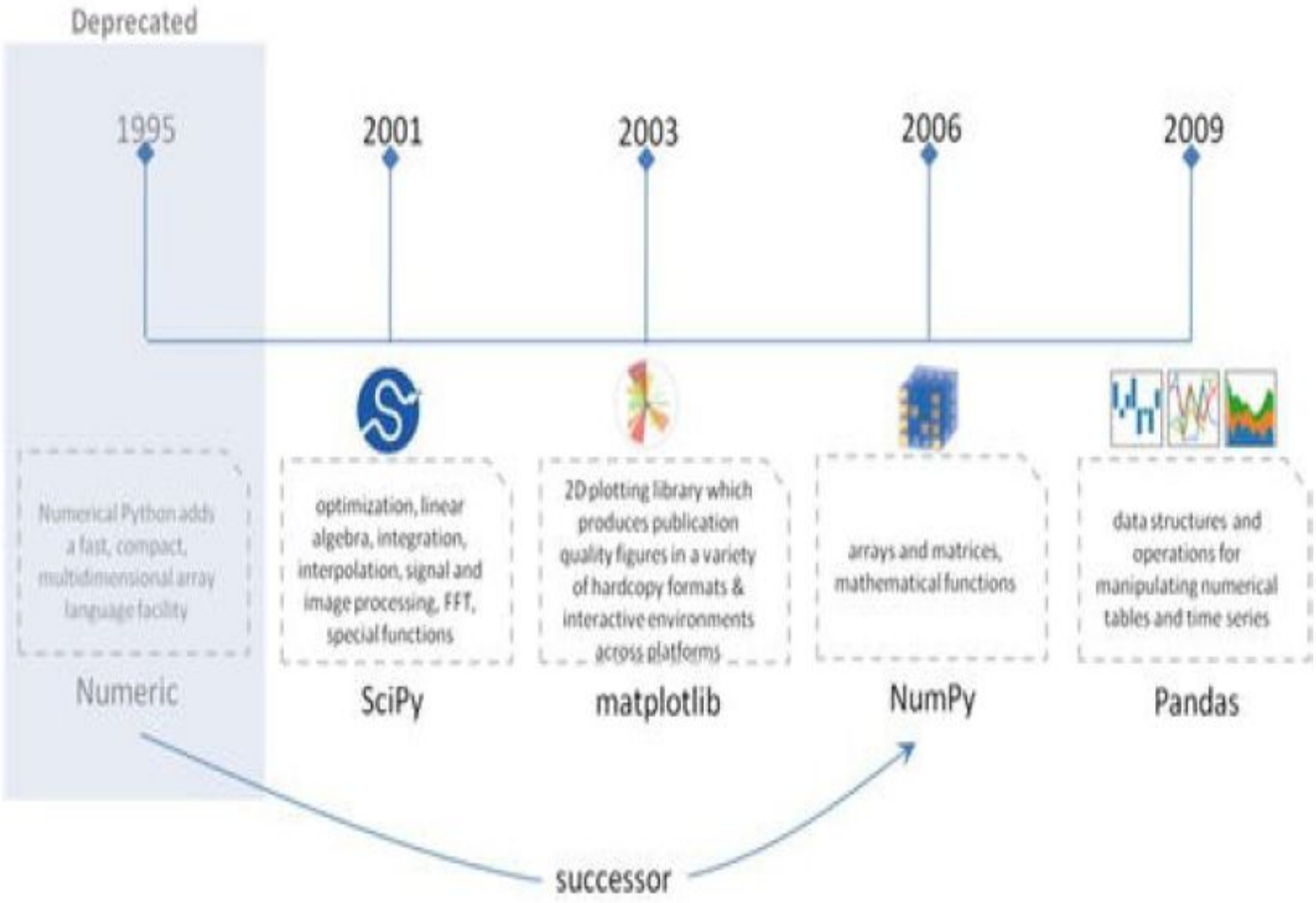
- There is a rich number of open source libraries available to facilitate practical machine learning. These are mainly known as scientific Python
- At a high level we can divide these libraries into data analysis and core machine learning libraries based on their usage/purpose.

1. *Data analysis packages*: These are the sets of packages that provide us the mathematic and scientific functionalities that are essential to perform data preprocessing and transformation.
2. *Core Machine learning packages*: These are the set of packages that provide us with all the necessary machine learning algorithms and functionalities that can be applied on a given dataset to extract the patterns.

- There are four key packages that are most widely used for data analysis .

1. NumPy
2. SciPy
3. Matplotlib
4. Pandas

- Pandas, NumPy, and Matplotlib play a major role and have the scope of usage in almost all data analysis tasks.


Deprecated
1995
2001
2003
2006
2009
Numerical Python adds a fast, compact, multidimensional array language facility
optimization, linear algebra, integration, interpolation, signal and image processing, FFT, special functions
2D plotting library which produces publication quality figures in a variety of hardcopy formats & interactive environments across platforms
arrays and matrices, mathematical functions
data structures and operations for manipulating numerical tables and time series
Numeric
SciPy
matplotlib
NumPy
Pandas
successor

NumPy

- Stands for Numerical Python. It is the core library for scientific computing in Python.
- Numpy is most suitable for performing basic numerical computations such as mean, median, range, etc. Alongside, it also supports the creation of multi-dimensional arrays.
- Provides techniques for loading, storing and manipulating in-memory data in Python.
- Usually real world data is heterogeneous comprising numerical measurements, categorical data, collection of images and may even include audio clips. All varied data can inherently be treated as arrays of numbers.

Ex:

An image can be an array of numbers representing pixel brightness across the area.

An audio clip can be an array of intensity versus time.

- Efficient storage and manipulation of numerical arrays is fundamental to the process of data science.
- NumPy arrays are like Python's built-in list type and provide much more efficient storage and data

operations as the arrays grow larger in size.

To work with numpy, import it :

```
import numpy as np
```
;where np is an alias of numpy

To display NumPy's built-in documentation, use:

```
np?
```

To create arrays from Python lists:

Test the time limits of creating normal list array in python as well as using numpy

%%timeit

l1=list(range(5000000))

l2=list(range(5000000))

%%timeit

np_array_1=np.arange(5000000)

np_array_2=np.arange(5000000)

To explicitly set the data type of the resulting array, use the 'dtype' keyword:

```
np.array([1, 2, 3, 4], dtype='float32')
```

NumPy arrays can explicitly be multi-dimensional which can be initialized using a list of lists.

```
np.array([range(i, i + 3) for i in [2, 4, 6]])
```

To index the first element of the array To index the last element To index the first 10 elements To index odd positions

Finding mean, variance and covaraince

Finding mean

```
from numpy import array
from numpy import mean
v = array([1,2,3,4,5,6])
print(v)
result = mean(v)
print(result)
```

Finding variance

```
from numpy import array
from numpy import var
v = array([1,2,3,4,5,6])
print(v)
result = var(v, ddof=1)
print(result)
```

```
from numpy import array
from numpy import cov
x = array([1,2,3,4,5,6,7,8,9])
print(x)
y = array([9,8,7,6,5,4,3,2,1])
print(y)
Sigma = cov(x,y)[0,1]
print(Sigma)
```

```
from numpy import array
from numpy import cov
x = array([1,2,3,4,5,6,7,8,9])
print(x)
y = array([9,8,7,6,5,4,3,2,1])
print(y)
Sigma = cov(x,y)[0,1]
print(Sigma)
```

using Pandas

- Pandas are an open source Python package providing fast, flexible, and expressive data structures designed to make working with "relational" or "labeled" data both easy and intuitive
- Handles tabular data sets comprising different variable types (integer, float, double, etc.).
- Can also be used to perform loading data or doing feature engineering on time series data.
- A Python library of data analysis and data manipulation tool built on NumPy
- Fast, intuitive data structure
- Before data is to be understood, one has to understand two key data structures in Pandas:
  - ✓ Series
  - ✓ DataFrames
- Series is a one dimensional indexed array. Elements in the series can be accessed using labels.(1-D labeled NumPy array)

Focus on

- Descriptive statistics, function application
- GroupBy: aggregation and transforming datasets
- Reshaping, pivot tables
- Joining, merging, concatenating
- Time series functionality

Example for Series and Dataframe

Creating a series by passing a list of values, and a custom index label.

```
s = pd.Series([1,2,3,np.nan,5,6],
index=
['A','B','C','D','E','F'])
print s
```

Create a dataframe

```
data = {'Gender': ['F', 'M',
'M'],'Emp_ID': ['E01','E02', 'E03'],
'Age': [25, 27, 25]}

df = pd.DataFrame(data,
columns=['Emp_ID','Gender','Age'])
```

Reading and Writing Data

To read from a csv file:

```
df=pd.read_csv('Data/mtcars.cs
v')
```

To read from a text file:

```
df=pd.read_csv('Data/mtcars.txt',
sep='\t')
```

To read from Excel:

```
df=pd.read_excel('Data/mtcars.xlsx','
Sheet1')
```

To read from multiple sheets of Excel into different dataframes

```
xlsx = pd.ExcelFile('file_name.xls')
sheet1_df = pd.read_excel(xlsx,
'Sheet1')
sheet2_df = pd.read_excel(xlsx,
'Sheet2')
```

To write to a file

```
df.to_csv('Data/mtcars_new.csv',
index=False)
df.to_csv('Data/mtcars_new.txt',
sep='\t',index=False)
```

Example

Read the csv file “sales” into jupyter notebook.

#Set current working directory to the location where ‘sales’ data is available

#import pandas library

```
import pandas as pd
data1 = pd.read_csv(“sales.csv” )
```

#View the imported data under ‘data1’ dataframe

```
data1
```

Note:

1.If the csv file contains any headers(with out values) we can skip the header based on its location in the file

Ex: data1 = read.csv(“sales.csv”, skiprows=1)

2.If the csv file doesn’t contain header then during import, mention the names of columns using the names attribute.

Ex:

```
data1 = read.csv(“sales.csv”, header=None,
names=[“SalesExecutive”,”Name”,”Gender”,”Age”,”Location”,”Sales (in Rs)”])
```

To read select rows(say 3 rows) from dataframe:

data1 = pd.read_csv("sales.csv", nrows=3)

data1

To read dataframe containing 'na' (not avaialable) values,use:

data1 = pd.read_csv(sales.csv", na_values =[" na"])

data1 #All the 'na' values will be turned to NaN in the dataframe.

Pandas has some built-in functions that enable to get better understanding of data using basic statistical summary methods.

describe()-

It returns the quick stats such as count, mean, std (standard deviation), min, first quartile, median, third quartile, max on each column of the dataframe

Ex:

cov() –

Covariance indicates how two variables are related. A positive covariance means the variables are positively related, while a negative covariance means the variables are inversely related.

Ex:

data1.cov()

corr()-

Correlation is another way to determine how two variables are related. It tells whether variables are positively or inversely related. Correlation also tells the degree to which the variables tend to move together.

Correlation ranges between -1 and 1.

data1.corr()

head()-

It displays by default top 5 records. Also it takes user specified number of rows as input.

Ex:

data1.head()

tail()-

It displays the bottom ‘n’ records.

```
data1.tail()
```

To extract column names from the dataframe

```
data1.columns
```

To extract column data types

```
data1.dtypes
```

To get unique column values

```
data1[column_name].unique()
```

#column_name must be in single quotes Ex:data1[‘Age’].unique()

To view by column name

```
data1[column_name]
```

To view by row number

```
data1[0:3]
```

To display only number of rows

```
data1.shape[0]
```

To rename a specific column name

```
df.rename(columns={'old_columnname':'new_columnname'},inplace=True)
```

To rename all column names of DataFrame

```
df.columns =['col1_new_name','col2_new_name'….]
```

To create a new column from existing column

```
df['new_column_name'] = df['existing_column_name'] + 5
```

To create new column from elements of two columns

```
df['new_column_name'] =df['existing_column1'] + '_' + df['existing_column2']
```

To add a list or a new column to DataFrame

```
df['new_column_name'] =pd.Series(mylist)
```

To drop missing rows and columns having missing values

To replace missing values with last valid observation

df.fillna(method='ffill', inplace=True, limit = 1)

Check missing value condition and return Boolean value of true or false for each cell

pd.isnull(df)

To replace all missing values for a given column with its mean

mean=df['column_name'].mean();
df['column_name'].fillna(mean)

To return mean, min, max,sum , count, cumulative sum for each column

df.mean()
df.max()
df.min()
df.sum()
df.count()
df.cumsum()

To apply a function along any axis of the DataFrame

df.apply(np.cumsum)

Merge/Join DataFrames

Pandas provide various facilities for easily combining together Series, DataFrame, and Panel objects with various kinds of set logic for the indexes and relational algebra functionality in the case of join merge-type operations.

```
data2 ={"emp_id": ["1","2","3","4","5"],"emp_name":
["Ajay","Vijay","Atul","Sneha","Ravi"]}
data2_df =pd.DataFrame(data2, columns =[‘emp_id’,’emp_name’])

data3 ={"emp_id": ["6","7","8","9","10"],"emp_name":
["Vimal","Rani","Sirisha","Ashok","Pavan"]}
data3_df =pd.DataFrame(data3, columns = ['emp_id','emp_name'])

df=pd.concat([data2_df,data3_df])
df
print data2_df.append(data3_df)
pd.concat([data2_df, data2_df], axis=1)                    #Joining two
dataframes along columns
```

Pandas offer SQL style merges as well. Left join produces a complete set of records from Table A, with the matching

print pd.merge(data2_df, data3_df, on='emp_id', how='inner')
#inner join

print pd.merge(data2_df, data3_df, on='emp_id', how='outer')
#outer join

data1.groupby(['Location','Sales (in Rs)']).max()
#Splitting data into groups using 'groupby'

Pivot Tables

Pandas provides a function 'pivot_table' to create MS-Excel spreadsheet style pivot tables. It can take following arguments:

- data: DataFrame object,
- values: column to aggregate,
- index: row labels,
- columns: column labels,
- Aggfunc: aggregation function to be used on values, default is NumPy.mean

Ex:

Matplotlib

- Matplotlib is a numerical mathematics extension NumPy and a great package to view or present data in a pictorial or graphical format.
- It enables analysts and decision makers to see analytics presented visually, so they can grasp difficult concepts or identify new patterns.
- The most common way for building a pictorial or graphical format is by using global functions to build and

display a global figure using matplotlib

1. plt.bar – creates a bar chart
2. plt.scatter – makes a scatter plot
3. plt.boxplot – makes a box and whisker plot
4. plt.hist – makes a histogram
5. plt.plot – creates a line plot

Ex:

To create a bar chart of test scores of five students

```
import matplotlib.pyplot as plt
x=np.arange(5)
y=(35,40,25,70,56)
plt.bar(x,y)
plt.show()
```

#show()/close() implies that any follow up plot commands will not use same figure.

```
plt.scatter(x,y)
plt.show()
```

- Histogram, line graph, and boxplot can be used directly on a dataframe.
- A simple histogram can be a great first step in understanding a dataset.

Ex:

```
%matplotlib inline
Import matplotlib.pyplot as plt
data1.histogram()
data1.plot()
data1.boxplot()
```

Customizing a histogram

The hist() function has many options to tune both the calculation and the display:

Example

- Read a csv file of sales data into python using pandas library

```
import os
os.getcwd()
```

Move the sales.csv file into the current working directory (or)

To change the current working directory

```
os.chdir('path of the required directory')
import pandas as pd
data1 =pd.read_csv("sales.csv")
data1
```

Start Exploration

- Start iPython interface in Inline Pylab mode

%pylab inline

plot(arange(5))

- Importing libraries and the data set:

Following are the libraries we will use during this project:

  - numpy
  - matplotlib
  - pandas

- No 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().

Understanding numerical variables:

```
import pandas as pd
import numpy as np
import matplotlib as plt
data1 = pd.read_csv("sales.csv")
data1
data1.head()
data1.tail()
data1.info()
data1.describe()          #Get summary of numerical variables
```

describe() function would provide count, mean, standard deviation (std), min, quartiles(25%,50%,75%) and max in its output

Understanding non numerical variables

- The data set also has two non numerical variables i.e. Gender and Location
- Look at frequency distribution to understand whether they make sense or not.

data1.shape #Get no. of Rows and Columns

data1.dtypes

data1['Gender'].value_counts()

data1[‘Location’].value_counts()

data1[data1['Gender'].str.contains('Male')] *#Get data related to Males*

data1[data1['Gender'].str.contains(‘Female')] *#Get data related to Females*

data1[data1['Gender'].str.contains('Female') &
data1['Location'].str.contains('Hyderabad')] *#Get data related*

Descriptive Statistics

```
data1['Sales ( in Rs)'].sum()       #Get total sale value of company
data1['Sales ( in Rs)'].mean()    # Get average sale value of company
data1['Sales ( in Rs)'].min()      # Get minimum sale value
data1['Sales ( in Rs)'].max()      # Get maximum sale value
data1['Sales ( in Rs)'].median()  # Get median sale value
data1.corr()  #To understand correlation significance of numerical
                variables
```

Distribution Analysis

- After understanding the basic nature of the data, next step is to understand distribution of variables.
- This can be achieved by plotting the histogram on numerical and non-numerical variables.

For numeric variables Age and Sales

data1[‘Age’].hist(bins = 10)

data1['Sales (in Rs)'].hist(bins=10)

data1.boxplot(column = ‘Sales (in Rs)’)

data1.boxplot(column = ‘Age’)

data1.boxplot(column = 'Sales (in Rs)', by = 'Age')

data1.boxplot(column = ‘Sales (in Rs)’, by = ‘Location’)

data1.boxplot(column = ‘Sales (in Rs)’, by =‘Gender’)

Interpretation

Interpretation:

- ✓ Analysis shows the maximum sale is near to 80(i.e.77) by an age 27.
- ✓ Minimum sale value is 11 at an age 20.
- ✓ Median sale is somewhere near to 40(around 34).
- ✓ Age group 24 is performing average sales for the company.
- ✓ Age group 27 is performing best sales for the company.
- ✓ Age has positive correlation over sales.
- ✓ Median sale value of Bangalore is higher than Mumbai.
- ✓ Least and highest sales happen in Bangalore.
- ✓ Region Hyderabad attributes to increased sale value making it a prospective sales horizon.
- ✓ Males have generated least and highest sales
- ✓ Median sale value of Males is higher then Females.
- ✓ Sales distribution appears uniform for all three regions with a sale value ranging from 25 to 75. Hence Bangalore region attributes to outliers in sales with sale values lying below and above the normal sale distribution.