Python for Aspiring Data Scientists

Vol. 1

Venkatesh K. Pappakrishnan, Ph.D.

COPYRIGHTS

ABOUT THE AUTHOR

Venkatesh Pappakrishnan is currently working as a Data scientist in Silicon Valley. Venkatesh has over 8 years of experience in mathematical modeling and algorithms, including 3 years of experience in machine learning and statistics. Venkatesh has numerous accomplishments in the field of Physics as well as in data science. In addition to his interest in research, he is also passionate about teaching. Venkatesh has over 6 years of experience in teaching to different levels of students, from college freshmen to graduate level. He is an expert in teaching hard-to-learn topics to students by using proper analogies and examples to help them understand the concepts easily. Currently, he also serves as an active reviewer of scientific journal articles in both data science and optics such as Data science journal, IEEE, and SPIE.

About his education - Venkatesh received his doctorate in Engineering Physics in 2014 for his work on Quantum Levitation using repulsive Casimir force. He has over 15 peer-reviewed publications, conference articles, and conference presentations in Physics. Venkatesh has primarily worked on developing an analytical model for Casimir force which has no closed form solution and optical cloaking. He is also well experienced in numerical algorithms. Venkatesh also holds a M.S. in Applied Physics, and a M.S. in Electrical Engineering. He received his bachelor's degree in Electrical Engineering from India.

PREFACE

Why should we use python for data science?

Python is one of the most commonly used high-level programming languages among data scientists both for developing proof-of-concepts, and deploying in production. The following are the key advantages in choosing python over any other programming language for data science.

1. The main advantage of choosing python over R, SAS, etc., is that python is based on object oriented programming and almost everything is treated as an object. When you are already familiar with C++ or java, it is easy to learn because the core concepts are the same. Python also has similar syntax to other programming languages such as C, C++, etc., which makes it easier to learn.
2. Python has plenty of open-source libraries available for data science projects. Some of the modules are Pandas, Numpy, Scipy, Statsmodel, and Scikit-learn that greatly reduce the programming effort needed for data wrangling and machine learning modeling process.
3. Although python is mainly used for prototyping, it can very well be used in production. Python is not as powerful as C/C++/java in terms of execution speed, however, it significantly reduces the development time required for building and implementation of a model in production. In addition, python can be extended with C/C++ scripts using wrappers to speed up certain parts of the code.
4. Python also supports multiprocessing, or parallel computing. Python also gained support from the deep learning community with many modules that are developed for python users such as Keras, Tensorflow, Lasagne, etc.
5. Python also comes with variety of visualization options such as Matplotlib, Pandas (with matplotlib as the backend), Seaborn, and Sokeh.
6. Python is becoming increasingly popular among data scientists due to significant advantages over other languages. As a result, the python community is ever expanding with an increased number of volunteers contributing to open-source libraries.

How will this book help you learn faster and retain information for a longer time than other books?

I have written this book in such a way that it is easily digestible for all readers. I have used two of most important and powerful tools in teaching. The first one is Infographics. "A picture is worth a ten thousand words" is a famous saying which has become a cliché in the internet world. It is easy to explain even a difficult concept, and also to retain the information for long time when pictures are used as a tool rather than just words. Sometimes, it is challenging to convey the message with just pictures. Hence, a new type of presentation tool has been born, which is infographics. An Infographic is the combination of pictures (graphics) and words (information) in the right proportion. Too much of either will ruin the purpose of using infographics. In this book, I have tried my best to present the concepts using infographics with right proportion of graphics and text. I would like to solicit your feedback in further improving the book. The second tool is a well-known and commonly used technique, but quite important for explaining concepts which is "examples". Each concept is accompanied by examples that are easy to understand. In my opinion, the combination of these two tools, infographics and examples are quite powerful together to explain a difficult-to-teach subject to anyone. The infographics help in maximum retention of the concepts in our

memory, as we remember pictures better than simple plain text. The examples will help readers to assimilate those difficult concepts easily.

How you should read this book?

Each chapter in the book consists of one or more infographics, followed by plenty of examples. My recommendation is spend at least few minutes to assimilate and understand the infographics to get a big picture of all the methods, and then proceed to the examples to better digest the information. Always refer back to the infographics as and when needed. Finally, after going through all the examples please refer back to the infographics again; everything in the infographics will now make sense, if it had not during your first look. Now that you know each and every method and when to use them, you will be able to retain the infographics in your memory much better than before.

What are you expected to know to read this book?

Basic programming logics and routines such as for loop, while loop, if condition, break, continue, creating functions, print statement, etc. If you have already used any of the low-level programming languages, and some high-level programming languages or tools that use traditional programming styles, then you are off to a good start.

ACKNOWLEDGEMENTS

I would like to thank my wife, parents, and family for their full support and encouragement to complete this book. I would also like to thank my friends for providing constructive feedback to improve the book, especially Jagadheesh, Ravivarman, and Rakesh.

Contents

I About the Author . . . 2
II Preface . . . 3
III Acknowledgements . . . 5
1 Introduction . . . 8
2 Data Structures . . . 10
2.1 Lists . . . 12
2.2 Arrays . . . 14
2.3 Strings . . . 14
2.4 Tuples . . . 18
2.5 Sets . . . 18
2.6 Dictionaries . . . 18
3 Numpy . . . 21
3.1 Import or Export Data . . . 23
3.2 Create an Array . . . 23
3.3 Inspect an Array . . . 31
3.4 Mathematical Operations . . . 33
3.5 Array Manipulation . . . 35
4 Pandas . . . 41
4.1 Create a DataFrame . . . 42
4.2 Data Exploration . . . 49
4.3 Data Visualization . . . 57
4.4 Data Cleansing . . . 70
4.5 Data Selection . . . 85
4.6 Data Manipulation . . . 91
4.7 Data Transformation . . . 102
4.8 Save Data . . . 118
5 Bibiliography . . . 120

List of Figures

1 Data structures in Python 11
2 Numpy 22
3 Methods to create a DataFrame 42
4 Methods to create a DataFrame 43
5 Data exploration techniques 50
6 Data visualization 58
7 Data cleaning 71
8 Data selection 86
9 Data manipulation 92
10 Data transformation 103

1
INTRODUCTION

Python is one of the most powerful tools in helping us to develop proof of concept in a short time. It is incredibly difficult and time consuming to use any of the low-level languages such as C, C++, Java, etc., for developing proof of concept. As mentioned earlier, python has many advantages over other languages or tools for developing machine learning models. Some of the major advantages are; A. It is based on OOPS and it retains the same programming structure as in C, C++, java, etc. B. There are plenty of open source libraries, and C. The ability to use in production. There are only a handful of python libraries that are commonly used by Data scientists. It is important to learn as many as possible to transform yourself into an expert in python, with a focus on data science. Here is the list,

Python modules:

1. Pandas - data loading, cleaning, and quick exploration
2. Numpy - array manipulation
3. Scikit-learn - machine learning algorithms
4. Matplotlib - data visualization
5. Statsmodel - statistics and statistical modeling
6. Seaborn - high-level statistical data visualization based on matplotlib
7. Bokeh - interactive data visualization
8. Scipy - packages for science, mathematics, and engineering

Tools:

1. Pycharm / Spyder - Pycharm IDE is from JetBrains, and it helps you to improve your coding style by giving suggestions based on PEP8 coding standards. Another feature I personally like is 'easy integration' with Github. We can easily perform almost any Git related task directly from Pycharm. Spyder comes with Anaconda package. Although it is easy to use, it is not as versatile as Pycharm.
2. Jupyter notebook - Interactive computing. It also helps to store your code and results in the same place as a notebook. The main advantage is that it can also be used for presentations.
3. Anaconda (or miniconda) package - contains supporting modules that are required for data mining.

For parallel processing:

1. Pyspark - Hadoop framework. It can handle huge amounts of data.
2. Celery - Framework for distributed computing based on the asynchronous task queue. It also supports job scheduling.
3. Dask - Dask can handle huge amounts of data just like Pyspark, and also uses DataFrames similar to Pandas as its data structure. If you are a fan of Pandas for its ease in data manipulation, then Dask is the best tool for you.

4. Multiprocessing - Inbuilt in Python. Simple to use but not as versatile as any of the others in the list. If you want to run a simple job in parallel, then you could go for multiprocessing in Python.

The above are the different modules, tools, and frameworks that are important for performing any data science related task efficiently. It is not an exhaustive list. Most of the modules, tools, and frameworks are open sourced, and the Python community is very active so that new items are being added to the list continuously. The new ones either improve or replace the old ones. So, as a data scientist you should be an active learner as well as open minded to learn anything that is required to complete the project.

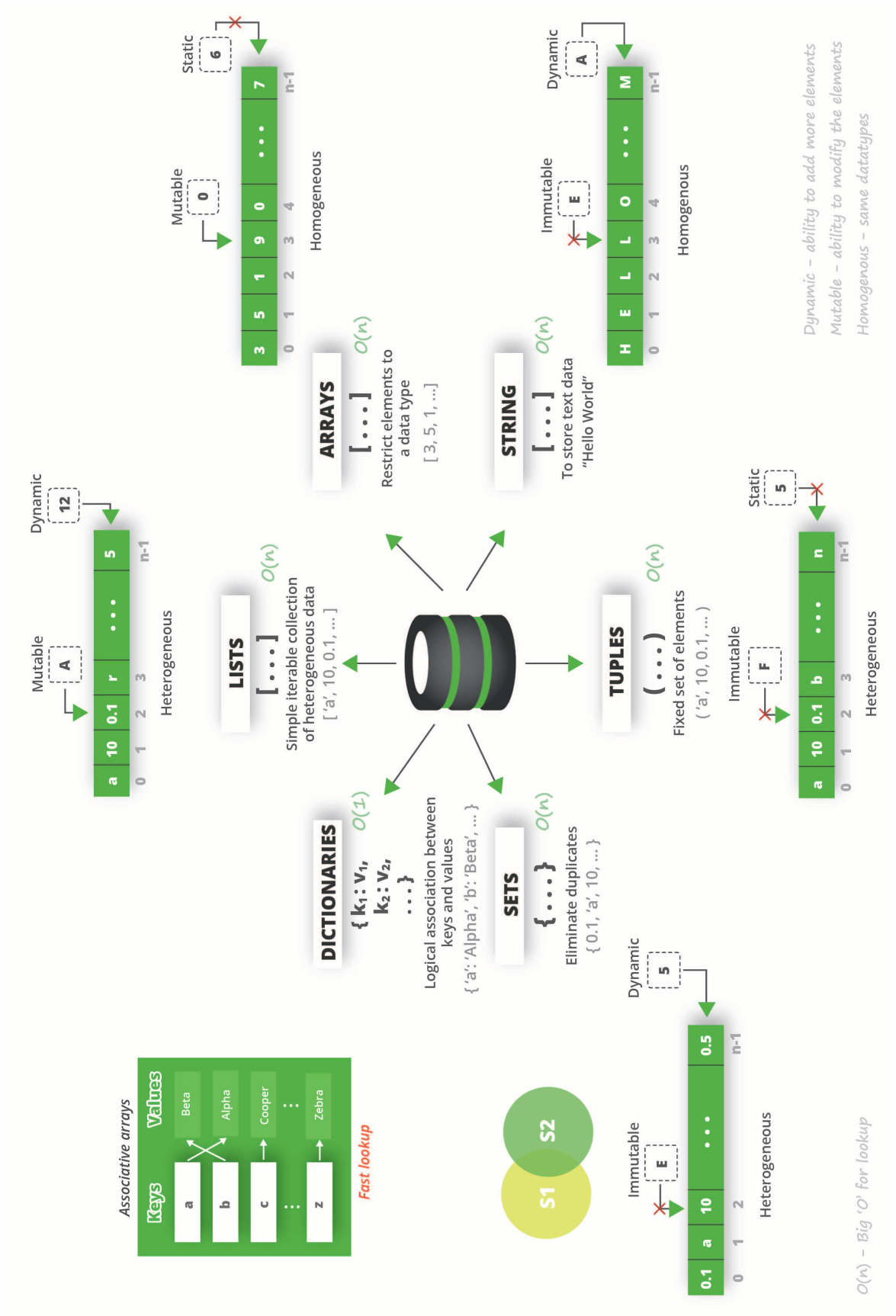
It is rather impossible to cover all the topics in a single book, or at least quite difficult for people to read a book that is more than 1000 pages long. Therefore, I will split the topics into a few books in a series by covering everything with 2-3 topics per book.

In this book, I will be covering the two most important modules in python - Pandas and Numpy. As mentioned above, Pandas is quite powerful and simple to use for data wrangling, and Numpy is important for array manipulation. These two are the most widely used modules among all the modules that exist for data science applications. All of the modules have an underlying data structure to handle the data. The efficiency and easiness to use mainly depends on the data structure. Hence, I will start the book by discussing different data structures in Python, their usage, pros and cons, etc.

2
DATA STRUCTURES

Data structure is a specific way of storing and organizing data in memory such that it can be retrieved and used in a most productive way. The data structure plays an important role in code optimization. The rate at which the data is stored or retrived, from the memory, is quite important in algorithm optimization especially when huge amount of data are involved.

There are several data structures in computer programming out of which only a subset are available in python right out-of-the-box. The following are the data structures in python that are mainly used for data science related projects.



Data structures in Python

```
In [1]: import numpy as np # load numpy module
```

2.1 Lists

- A list is a collection of homogeneous/heterogenous elements (int, float, string, etc.)

Time complexity:

- Read (with index) / append / add: O(1)
- Insert / delete: O(n) - expensive as the items have to be rearranged after insertion or deletion
- Value lookup: O(n)

```
In [2]: list_ = ['hello', 'how', 'are', 'you', 1, 10, 'how', 'well', 'are'] # created a simple
        list
        list_

Out[2]: ['hello', 'how', 'are', 'you', 1, 10, 'how', 'well', 'are']

In [3]: n = len(list_)  # length of the list_
        n

Out[3]: 9
```

Indices are numbered from 0 to n-1 or alternatively from -n to -1

```
In [4]: print(list_[0]) # first element
        print(list_[8]) # last element

        print(list_[-9]) # first element
        print(list_[-1]) # last element

hello
are
hello
are
```

Slicing

- To get a specific set of elements from the list

```
In [5]: list_[:3]  # left-closed type data retrieval

Out[5]: ['hello', 'how', 'are']
```

Only elements with indices 0-2 are retrived and the element with index 3 is skipped

```
In [6]: list_[2:5]  # 2-4 are retrived
```

```
Out[6]: ['are', 'you', 1]

In [7]: list_[2:]  # 2-8 elements are retrieved. Leaving right-end empty implies last element in
        the list

Out[7]: ['are', 'you', 1, 10, 'how', 'well', 'are']

In [8]: list_[:4] # 0-3 elements. Leaving left end  empty implies beginning of the list

Out[8]: ['hello', 'how', 'are', 'you']

In [9]: list_[-1:]  # -1 implies last element of the list

Out[9]: ['are']

In [10]: list_[:-1] # only the last element is skipped

Out[10]: ['hello', 'how', 'are', 'you', 1, 10, 'how', 'well']

In [11]: list_[:-2] # last and last-before elements are skipped (-1 and -2 respectively)

Out[11]: ['hello', 'how', 'are', 'you', 1, 10, 'how']

In [12]: list_[::-1]  # to reverse the order

Out[12]: ['are', 'well', 'how', 10, 1, 'you', 'are', 'how', 'hello']
```

Delete

```
In [13]: list_

Out[13]: ['hello', 'how', 'are', 'you', 1, 10, 'how', 'well', 'are']

In [14]: del list_[4]

         list_ # the list_[4] was deleted and replaced by the next element in the list

Out[14]: ['hello', 'how', 'are', 'you', 10, 'how', 'well', 'are']
```

Replace

```
In [15]: list_[4] = 1232

         list_ # list_[4] is replaced by 1232 and the elements with index >= 4 remains the same

Out[15]: ['hello', 'how', 'are', 'you', 1232, 'how', 'well', 'are']
```

Insert

```
In [16]: list_.insert(4, 111)

         list_ # 111 is inserting into list_[4] and the elements with index >= 4 are moved to the
         right by one

Out[16]: ['hello', 'how', 'are', 'you', 111, 1232, 'how', 'well', 'are']
```

Please note that these simple tricks will be very helpful during projects

2.2 Arrays

A list is a collection of homogeneous items that are contiguously arranged. - Mutable but static (the size cannot be changed or in other words new elements cannot be added to it)

Time complexity:

- Same as that of lists - O(1) with index and O(n) for lookup

```
In [17]: arrays_ = np.array(list_)
         arrays_

Out[17]: array(['hello', 'how', 'are', 'you', '111', '1232', 'how', 'well', 'are'],
               dtype='<U5')
```

Please notice that the numbers in the list_ are converted to strings when we converted the list_ to an array. An array should consist of elements of the same datatype. As converting a string/char to a number is impossible, the numbers are rather converted to strings.

```
In [18]: arrays_[1] = 122  # arrays_[1] element will be replaced with "122"
         arrays_

Out[18]: array(['hello', '122', 'are', 'you', '111', '1232', 'how', 'well', 'are'],
               dtype='<U5')
```

Insert or delete operation cannot be performed on an array as the size cannot be altered. When an array is created a fixed memory location is allocated which cannot be extended. Inserting a value at ith index will overwrite the existing value with the new one and the other elements will remain unchanged.

2.3 Strings

- Immutable, homogeneous, and dynamic

Time complexity:

- Same as that of lists and arrays - O(1) with index and O(n) for lookup

Reading/slicing:

- Same as in lists. Just consider each character in the string as an element of a list.

```
In [19]: string_ = "Hello, I am Jack and I am a data scientist"

         string_[1]  # 'e' from Hello

Out[19]: 'e'

In [20]: string_[:11]  # slicing

Out[20]: 'Hello, I am'

In [21]: string_[1] = 'r' # immutable


         ---------------------------------------------------------------------------

         TypeError                                 Traceback (most recent call last)

         <ipython-input-21-33093e5f46be> in <module>()
    ----> 1 string_[1] = 'r' # immutable


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

Combining strings:

```
In [22]: # You can add more words or lines by adding 2 strings arithmatically.
         string_new = "I am 10 years experienced"

         string_ + ". " + string_new  # very simple but painful to add ". " between every string
         that you want to add

Out[22]: 'Hello, I am Jack and I am a data scientist. I am 10 years experienced'

In [23]: ". ".join([string_, string_new])  # This solves the above problem in joining multiple
         strings

Out[23]: 'Hello, I am Jack and I am a data scientist. I am 10 years experienced'
```

Find

- Returns the index of the leftmost word/character searched

```
In [24]: string_.find("data")  # returns the index of the beginning of the word
```

```
Out[24]: 28

In [25]: string_[28:32]

Out[25]: 'data'

In [26]: string_.find("Amazon")  # Returns -1 if search word is not in the string

Out[26]: -1
```

rfind:

- Returns the index of the rightmost word/character searched

```
In [27]: print(string_.find("am"))
         print(string_.rfind("am"))  # returns the index of the first character of the rightmost
         word
```

```
9
23
```

Count

- Counts the number of occurances of a word/character

```
In [28]: string_.count('am')

Out[28]: 2

In [29]: string_.count('a')

Out[29]: 7
```

Split

- Splits a string, at a specified character, into a list of elements

```
In [30]: string_list = string_.split(" ")
         string_list

Out[30]: ['Hello,', 'I', 'am', 'Jack', 'and', 'I', 'am', 'a', 'data', 'scientist']
```

Startswith

- Checks if the string starts with the search word/character

```
In [31]: string_.startswith('Hello')

Out[31]: True

In [32]: string_.startswith('I') # I is not the beginning of the string

Out[32]: False

In [33]: # If you like to check for a word in the string. Do the following

         string_list  # list of words from string_

         [word_.startswith('am') for word_ in string_list]

Out[33]: [False, False, True, False, False, False, True, False, False, False]
```

So there are two 'am's in the string

```
In [34]: sum([word_.startswith('am') for word_ in string_list]) # to directly get the count

Out[34]: 2
```

Endswith

- Similar to 'startswith' method but here to check at the end of a string

```
In [35]: string_.endswith('scientist') # note that it could be a partial word

Out[35]: True

In [36]: string_.endswith('tist') # note that it could be a partial word

Out[36]: True

In [37]: string_.endswith('data')

Out[37]: False
```

Replace

- Although a specific character element cannot be replaced, word as a whole can be replaced with another word.

```
In [38]: string_.replace("Jack", "John")  # replaces Jack with John

Out[38]: 'Hello, I am John and I am a data scientist'
```

2.4 Tuples

- Immutable and static
- Tuples are used when we want to restrict the user from changing values in the list

Time complexity:

- Same as that of lists and arrays - O(1) with index and O(n) for lookup

```
In [39]: tuples_ = tuple(list_)
         tuples_

Out[39]: ('hello', 'how', 'are', 'you', 111, 1232, 'how', 'well', 'are')
```

Note that the elements are enclosed in parentheses.

```
In [40]: # Another method for creating tuples
         tuples_ = 'hello', 'how', 'are', 1, 1232, 'well'
         tuples_

Out[40]: ('hello', 'how', 'are', 1, 1232, 'well')
```

2.5 Sets

A set is an unordered collection with no duplicate elements. Set is another data type that we often use in data wrangling. It is used to eliminate duplicate values from a list or an array. - Immutable and dynamic

```
In [41]: set_ = set(list_)
         set_

Out[41]: {111, 1232, 'are', 'hello', 'how', 'well', 'you'}

In [42]: # set_[2] = 'lel' will not work as the values cannot be replaced
         set_.add('lel')  # values can be added - dynamic
         set_

Out[42]: {111, 1232, 'are', 'hello', 'how', 'lel', 'well', 'you'}
```

It returns only the unique values from the list_ in no specific order.

2.6 Dictionaries

A dictionary consists of key-value pairs in no specific order. The key-value pairs are stored in an associative array. Each key maps to a value or a list or another dictionary.

The keys are usually hash codes that are generated using a hash function. The idea is to have a unique hash code for every value to be stored. Hence the read operation is much faster than any other data types - O(1) (even for search operations - the best case). Please note that here we sacrifice space complexity, by storing keys in the memory, to achieve better time complexity.

Time complexity:

- Most dict operations O(1) (even for lookup operation)

```
In [43]: # Lets store age of people as dictionary
         dict_ = {"Julie": 32, "Rahul": 23, "Jasmine": 12, "Jack": 15, "Jennifer": 18}
         dict_

Out[43]: {'Julie': 32, 'Rahul': 23, 'Jasmine': 12, 'Jack': 15, 'Jennifer': 18}

In [44]: # To retrieve values
         dict_['Jasmine']

Out[44]: 12

In [45]: # To get list of keys
         dict_.keys()

Out[45]: dict_keys(['Julie', 'Rahul', 'Jasmine', 'Jack', 'Jennifer'])

In [46]: # To get list of values
         dict_.values()

Out[46]: dict_values([32, 23, 12, 15, 18])
```

Other methods to create a dictionary

```
In [47]: # method 2
         dict([('Julie', 32), ('Rahul', 23), ('Jasmine', 12)])

Out[47]: {'Julie': 32, 'Rahul': 23, 'Jasmine': 12}

In [48]: # method 3
         dict(Julie=32, Rahul=23, Jasmine=12)

Out[48]: {'Julie': 32, 'Rahul': 23, 'Jasmine': 12}

In [49]: # method 4 - list comprehension technique
         {x: 2*x for x in range(5)}

Out[49]: {0: 0, 1: 2, 2: 4, 3: 6, 4: 8}
```

To print a dictionary

```
In [50]: for k, v in dict_.items():
             print(k, v)

Julie 32
Rahul 23
Jasmine 12
Jack 15
Jennifer 18
```

To reverse the key-value pairs

```
In [51]: dict_reversed = {v:k for k, v in dict_.items()}  # Note the curly braces
         dict_reversed

Out[51]: {32: 'Julie', 23: 'Rahul', 12: 'Jasmine', 15: 'Jack', 18: 'Jennifer'}
```

3

NUMPY

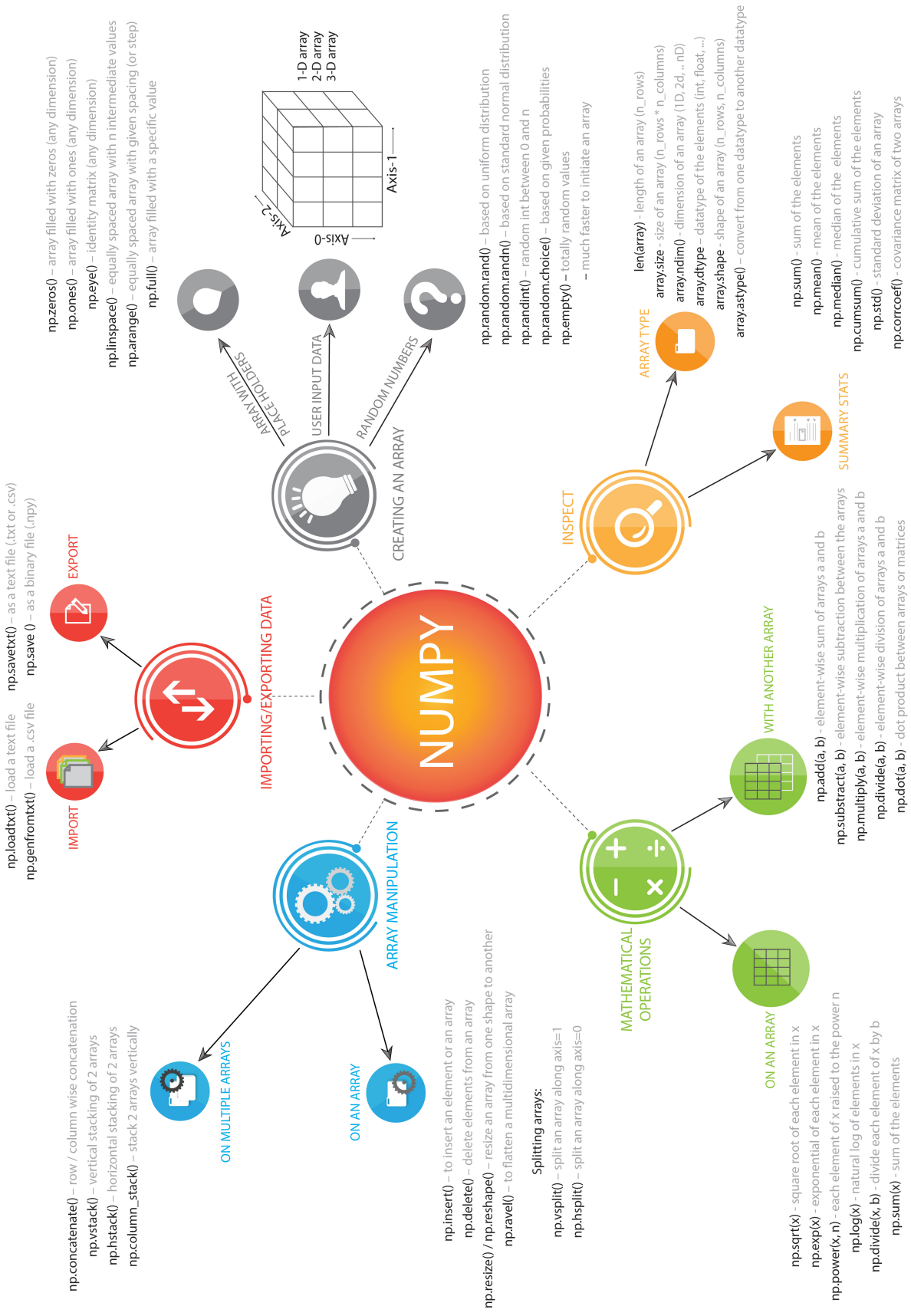
The main objective of numpy module is to handle or create single- or multi- dimensional arrays. As mentioned in the previous chapters, it is one of the backend engines for Pandas and other modules such as sklearn, scipy, etc., to handle array manipulation. The numpy arrays are more convenient to use, highly memory efficient, and allows faster access (both read and write) compared to python's built-in lists.

In data science, array creating and manipulation are most common tasks that you might encounter almost everyday. For example, you might want to create an array filled with zeros or create an array with random numbers for Monte Carlo simulation and so on. Therefore it will be quite helpful if you are familiar with what numpy has to offer you. In general, the more functions you know, that are already built-in, the faster and easier you can perform the task in hand.

There are only 5 major types of operations that you can perform using numpy, which are

1. Import/export from a file,
2. Array creation,
3. Array inpection,
4. Mathematical operations on arrays, and
5. Aggregation of two or more arrays.

The infographics presented below contains everything that you need to know to jump start in numpy.


NUMPY
IMPORTING/EXPORTING DATA
IMPORT
np.loadtxt() – load a text file
np.genfromtxt() – load a .csv file
EXPORT
np.savetxt() – as a text file (.txt or .csv)
np.save () – as a binary file (.npy)
CREATING AN ARRAY
ARRAY WITH PLACE HOLDERS
USER INPUT DATA
RANDOM NUMBERS
np.zeros() – array filled with zeros (any dimension)
np.ones() – array filled with ones (any dimension)
np.eye() – identity matrix (any dimension)
np.linspace() – equally spaced array with n intermediate values
np.arange() – equally spaced array with given spacing (or step)
np.full() – array filled with a specific value
Axis-0
Axis-2
Axis-1
1-D array
2-D array
3-D array
np.random.rand() – based on uniform distribution
np.random.randn() – based on standard normal distribution
np.randint() – random int between 0 and n
np.random.choice() – based on given probabilities
np.empty() – totally random values
– much faster to initiate an array
INSPECT
ARRAY TYPE
len(array) - length of an array (n_rows)
array.size - size of an array (n_rows * n_columns)
array.ndim() - dimension of an array (1D, 2d, .. nD)
array.dtype - datatype of the elements (int, float, ...)
array.shape - shape of an array (n_rows, n_columns)
array.astype() – convert from one datatype to another datatype
SUMMARY STATS
np.sum() - sum of the elements
np.mean() - mean of the elements
np.median() - median of the elements
np.cumsum() - cumulative sum of the elements
np.std() - standard deviation of an array
np.corrcoef() - covariance matrix of two arrays
MATHEMATICAL OPERATIONS
WITH ANOTHER ARRAY
np.add(a, b) - element-wise sum of arrays a and b
np.substract(a, b) - element-wise subtraction between the arrays
np.multiply(a, b) - element-wise multiplication of arrays a and b
np.divide(a, b) - element-wise division of arrays a and b
np.dot(a, b) - dot product between arrays or matrices
ON AN ARRAY
np.sqrt(x) - square root of each element in x
np.exp(x) - exponential of each element in x
np.power(x, n) - each element of x raised to the power n
np.log(x) - natural log of elements in x
np.divide(x, b) - divide each element of x by b
np.sum(x) - sum of the elements
ARRAY MANIPULATION
ON AN ARRAY
np.insert() – to insert an element or an array
np.delete() – delete elements from an array
np.resize() / np.reshape() – resize an array from one shape to another
np.ravel() – to flatten a multidimensional array
Splitting arrays:
np.vsplit() – split an array along axis=1
np.hsplit() – split an array along axis=0
ON MULTIPLE ARRAYS
np.concatenate() – row / column wise concatenation
np.vstack() – vertical stacking of 2 arrays
np.hstack() – horizontal stacking of 2 arrays
np.column_stack() - stack 2 arrays vertically

Numpy

```
In [1]: import numpy as np  # load numpy module
        import copy  # to copy arrays
```

3.1 Import or Export Data

3.1.1 Import data

```
In [2]: # Load text file
        array_ = np.loadtxt("sample.txt", delimiter=",", dtype=int)
        array_

Out[2]: array([44,  5,  2, 76,  0, 12,  5, 67,  0])

In [3]: # Load .csv file
        np.genfromtxt("sample.csv", delimiter=",", dtype=int)

Out[3]: array([[ 2,  3, 56, 34, 12, 34, 56],
               [ 0, 45,  6,  7,  1,  2,  9]])
```

3.1.2 Export data

```
In [4]: # Save as txt file
        np.savetxt("output.txt", array_, delimiter=" ")

In [5]: # Save as .csv file
        np.savetxt("output.csv", array_, delimiter=",")
```

3.2 Create an Array

3.2.1 User input data

One way to create an array is to input data by yourself to the numpy module.

One dimensional arrays

- created using a list

```
In [6]: array_ = np.array([1, 2, 6, 7, 8, 1])
        array_

Out[6]: array([1, 2, 6, 7, 8, 1])

In [7]: array_ = np.array([1, "a", 6, 7, 8, 1])
        array_

Out[7]: array(['1', 'a', '6', '7', '8', '1'], dtype='<U11')
```

When a char is included in an array with other elements being int or float, the datatype of all the elements will be converted to match a common datatype. Arrays are homogenous in nature and they can only store elements with similar datatype. Hence, in the above case, the numbers are converted to chars as the reverse is not possible.

Two dimensional arrays

- created using list of lists

```
In [8]: array2_ = np.array([[1, 2, 3], [4, 5, 6]])
        array2_

Out[8]: array([[1, 2, 3],
               [4, 5, 6]])

In [9]: array2_[0][0]

Out[9]: 1

In [10]: array2_ = np.array([(1, 2, 3), (4, 5, 6)])
         array2_

Out[10]: array([[1, 2, 3],
                [4, 5, 6]])
```

Two dimensional arrays can be created using a list of lists or list of tuples. Remember that tuples are immutable and the same rule applies here. When a list of tuples are used to create an array then it's elements cannot be modified.

Multi-dimensional arrays

- created using list-of-list-of-lists

```
In [11]: array3_ = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]], [[1, 8, 5], [4,
         2, 9]]])
         array3_

Out[11]: array([[[ 1,  2,  3],
                 [ 4,  5,  6]],

                [[ 7,  8,  9],
                 [10, 11, 12]],

                [[ 1,  8,  5],
                 [ 4,  2,  9]]])
```

Assume 3D array as n layers of stacked 2D-arrays back-to-back with index from 0 to n-1. If you pay close attention, when each element of a list is replaced by a list (instead of an element) we add another dimension to the data.

- 1-D: List of elements
- 2-D: List-of-lists with elements

- 3-D: Lists-of-lists-of-lists with elements and so on.

For data retrival, the first index represents the index of the highest dimension and the last index represents the index of the lowest dimension

```
In [12]: array3_.shape
         # It is important to know the dimension of an array before trying to access it elements
         to avoid IndexError

Out[12]: (3, 2, 3)

In [13]: array3_[0]

Out[13]: array([[1, 2, 3],
                [4, 5, 6]])

In [14]: array3_[1]

Out[14]: array([[ 7,  8,  9],
                [10, 11, 12]])

In [15]: array3_[2]

Out[15]: array([[1, 8, 5],
                [4, 2, 9]])

In [16]: array3_[0]

Out[16]: array([[1, 2, 3],
                [4, 5, 6]])
```

Shallow and deep copy of an array

The shallow and deep copy are relevant only for compound objects (objects that contain other objects)

```
In [17]: comp = [[0, 1, 2], 3, 5, 7]

In [18]: # Shallow copy
         a = np.copy(comp)
         a

Out[18]: array([list([0, 1, 2]), 3, 5, 7], dtype=object)
```

Only shallow copy exists inside numpy module. To deepcopy an array we need to use another module "copy"

```
In [19]: # Deep copy
         b = copy.deepcopy(comp)
         b
```

```
Out[19]: [[0, 1, 2], 3, 5, 7]
```

```
In [20]: print("id(a):", id(a))
         print("id(b):", id(b))
         print("id(comp):", id(comp))
```

```
id(a): 2166905174976
id(b): 2166905129416
id(comp): 2166905130952
```

The objects ids are different for all the arrays

Let's modify comp and check how other arrays are affected

```
In [21]: # append an element to the list inside the array
         comp[0].append(3)
```

```
In [22]: print(comp)
         print(a)
         print(b)
```

```
[[0, 1, 2, 3], 3, 5, 7]
[list([0, 1, 2, 3]) 3 5 7]
[[0, 1, 2], 3, 5, 7]
```

As expected the array b (deep copied) remains untouched whereas the array a (shallow copied) is modified due to the changes made to array comp. Please refer below for the object ids of the arrays. For objects comp[0] and a[0] the ids are the same which implies that they both point to the same memory location.

```
In [23]: print(id(comp[0]))
         print(id(a[0]))
         print(id(b[0]))
```

```
2166905193032
2166905193032
2166905192904
```

```
In [24]: # append an element to the list inside the array
         comp.append(3)
```

```
In [25]: print(comp)
         print(a)
         print(b)
```

```
[[0, 1, 2, 3], 3, 5, 7, 3]
[list([0, 1, 2, 3]) 3 5 7]
[[0, 1, 2], 3, 5, 7]
```

Please note that when an element is appended to comp (deep copied), a and b will remain unchanged.

To understand this, we should know how the data are stored. Each element in the list acts as an object pointing to a memory location for the value. When we shallow copy an array, a new object is created with objects inside the array (objects for elements) referring to the same memory location and when any of the objects is modified it gets reflected in the copied array. However, when a new element is appended to the parent array, the object of which was not present when copied, the changes will not get reflected in the copied array.

3.2.2 Array with placeholders

Zeros

- array or matrix with all zeros

```
In [26]: np.zeros(5)

Out[26]: array([0., 0., 0., 0., 0.])

In [27]: np.zeros([5, 5])

Out[27]: array([[0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.]])
```

Ones

- array or matrix with all 1s

```
In [28]: np.ones(5)

Out[28]: array([1., 1., 1., 1., 1.])

In [29]: np.ones([5, 5])

Out[29]: array([[1., 1., 1., 1., 1.],
                [1., 1., 1., 1., 1.],
                [1., 1., 1., 1., 1.],
                [1., 1., 1., 1., 1.],
                [1., 1., 1., 1., 1.]])
```

Identity matrix - array or matrix with diagonal elements as 1

```
In [30]: # Identity matrix (I) or diagonal matrix
         np.eye(5)

Out[30]: array([[1., 0., 0., 0., 0.],
                [0., 1., 0., 0., 0.],
                [0., 0., 1., 0., 0.],
                [0., 0., 0., 1., 0.],
                [0., 0., 0., 0., 1.]])

In [31]: np.eye(5, 1)

Out[31]: array([[1.],
                [0.],
                [0.],
                [0.],
                [0.]])
```

linspace

- array of equally spaced elements with given n

```
In [32]: np.linspace(start=0, stop=10, num=11) # or simply np.linspace(0, 10, 11)

Out[32]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

arange

- array of equally spaced elements with given stepsize

```
In [33]: np.arange(start=0, stop=10, step=1) # or simply np.arange(0, 10, 1)

Out[33]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

full

- array or matrix filled with specific values

```
In [34]: np.full(shape=5, fill_value=8) # or simply np.full(5, 8)

Out[34]: array([8, 8, 8, 8, 8])

In [35]: np.full(shape=(5, 5), fill_value=8) # or simply np.full((5, 5), 8)

Out[35]: array([[8, 8, 8, 8, 8],
                [8, 8, 8, 8, 8],
                [8, 8, 8, 8, 8],
                [8, 8, 8, 8, 8],
                [8, 8, 8, 8, 8]])
```

3.2.3 Random number generator

Random number generator plays an important role in statistics and data science. Many algorithms, such as Monte Carlo, Markov chain, etc., requires random numbers for solving an complex equation numerically (simulation) or predict the next state or sequence of states probabilitistically. Random numbers has applications in numerous fields and it is one of the basic elements for any algorithm or simulation.

In python, you can choose how a random number should be generated. Should it follow a specific probability distribution function such as gaussian distribution, uniform distribution, etc., or user-given probabilities for the given values or totally random?

rand

- returns n random samples from an uniform distribution over (0, 1)

```
In [36]: np.random.rand(10)

Out[36]: array([0.42001413, 0.7568704 , 0.79644949, 0.63733904, 0.31788908,
                0.69618985, 0.90308158, 0.34790605, 0.833274  , 0.64138867])
```

randn

- returns n random samples from a standard normal distribution (mean=0 and S.D.=1)

```
In [37]: # array of random float numbers between -inf and inf - from standard normal distribution
         np.random.randn(10)

Out[37]: array([-1.29125034, -1.49725343,  0.4283142 ,  0.50363909,  1.18678241,
                 0.75859435, -0.26479496, -0.361355  ,  1.44108636, -0.41559579])

In [38]: # random floats between 0 and n (just multiply by n)
         np.random.randn(10) * 10

Out[38]: array([ 16.68555649, -17.89799993,  -4.9921352 ,   3.03904667,
                 -6.80544991, -14.11503242,  14.6578837 ,  -3.43065088,
                  3.07683341,   5.68022153])
```

randint

- returns n random integers from an uniform distribution over (0, n)

```
In [39]: # array of random float numbers between 0 and 1
         np.random.randint(0, 10, size=5)

Out[39]: array([3, 3, 2, 3, 7])

In [40]: # array of random float numbers between 0 and 1
         np.random.randint(0, 10, size=(5, 5))
```

```
Out[40]: array([[7, 4, 1, 1, 0],
                [2, 9, 7, 6, 2],
                [6, 0, 5, 6, 6],
                [4, 7, 1, 9, 4],
                [6, 2, 9, 2, 2]])
```

choice

- to pick values based on user-given probabilities

```
In [41]: np.random.choice(a = 5, size = 3)
```

```
Out[41]: array([0, 4, 4])
```

a - array or int(np.arange(n) is used to create an array) size - number of output samples

```
In [42]: np.random.choice(5, 3, p=[0.1, 0.5, 0.3, 0.08, 0.02])
```

```
Out[42]: array([1, 1, 1], dtype=int64)
```

p=[0.1, 0.5, 0.3, 0.08, 0.02] are the probabilities for the numbers from 0-4

```
In [43]: # with a user-defined list
         list_ = [10, 3, 6, 7, 8]

         np.random.choice(list_, 10, p=[0.1, 0.5, 0.3, 0.08, 0.02])
```

```
Out[43]: array([3, 6, 3, 3, 7, 8, 3, 3, 6, 3])
```

empty

- to create an empty array or matrix (with random values and with random datatype)
- the datatype can be specified or will be assigned when values are added to it
- much faster to initialize an array compared to creating an array with preset values like zeros() ot ones()

```
In [44]: np.empty(5) # Totally random values
```

```
Out[44]: array([0.1 , 0.5 , 0.3 , 0.08, 0.02])
```

```
In [45]: np.empty((3, 3))
```

```
Out[45]: array([[0.000e+000, 0.000e+000, 0.000e+000],
                [0.000e+000, 0.000e+000, 6.146e-321],
                [0.000e+000, 0.000e+000, 0.000e+000]])
```

3.3 Inspect an Array

For the study, let's create an array and insert NaN into it.

```
In [46]: array_ = np.random.randint(0, 10, size=10)
         array2_ = np.eye(4, 5)

         array2_[2][2] = np.NaN # NaN is inserted for study

In [47]: array2_

Out[47]: array([[ 1.,  0.,  0.,  0.,  0.],
                [ 0.,  1.,  0.,  0.,  0.],
                [ 0.,  0., nan,  0.,  0.],
                [ 0.,  0.,  0.,  1.,  0.]])
```

3.3.1 Array Type

shape

- number of rows and columns returned as a tuple

```
In [48]: array_.shape

Out[48]: (10,)

In [49]: array2_.shape

Out[49]: (4, 5)
```

size

- provides the size of an array (row * col) or row (1D)

```
In [50]: array_.size

Out[50]: 10

In [51]: array2_.size

Out[51]: 20
```

len

- number of rows in an array (2D, 3D,...) or length of an array (1D)

```
In [52]: len(array_)

Out[52]: 10
```

```
In [53]: len(array2_)

Out[53]: 4

In [54]: array_

Out[54]: array([8, 8, 1, 4, 0, 7, 4, 0, 8, 6])
```

ndim

- dimension of an array

```
In [55]: array_.ndim

Out[55]: 1

In [56]: array2_.ndim

Out[56]: 2
```

dtype

- datatype of the elements

```
In [57]: array_.dtype

Out[57]: dtype('int32')

In [58]: array2_.dtype

Out[58]: dtype('float64')
```

astype

- to convert an array from one datatype into another

Although it is not an inspection type function, it is appropriate to place it here.

```
In [59]: array_.astype(np.float).dtype # converted from int to float

Out[59]: dtype('float64')

In [60]: array2_.astype(np.object).dtype # converted from float to object

Out[60]: dtype('O')
```

3.3.2 Summary statistics

```
In [61]: a = [11, 5, 7, 12, 2, 3, 15, 7, 9, 3, 5, 9]
         b = [4, 5, 8, 12, 3, 3, 12, 8, 9, 5, 6, 8]

In [62]: np.sum(a) # sum of all the elements in the array

Out[62]: 88

In [63]: np.mean(a) # mean of the elements in the array

Out[63]: 7.333333333333333

In [64]: np.median(a) # median of the elements in the array

Out[64]: 7.0

In [65]: np.cumsum(a) # cumulative sum of the elements

Out[65]: array([11, 16, 23, 35, 37, 40, 55, 62, 71, 74, 79, 88], dtype=int32)
```

For multidimensional array use axis=0 for cumsum along y-axis or axis=1 for cumsum along x-axis

```
In [66]: np.std(a)

Out[66]: 3.8369548110737797

In [67]: np.corrcoef(a, b) # Covariance matrix

Out[67]: array([[1.       , 0.7955227],
                [0.7955227, 1.       ]])
```

The correlation coefficients are arranged in a matrix as follows corr(a, a) corr(a, b) corr(b, a) corr(b, b)

3.4 Mathematical Operations

Let's create two simple arrays for the study

```
In [68]: a = np.array([10, 5, 2, 1, 6, 3])
         b = np.array([5, 0.5, 8, 1, 3, 7])
```

3.4.1 On an array

```
In [69]: np.sqrt(a) # square root of each element in the array

Out[69]: array([3.16227766, 2.23606798, 1.41421356, 1.        , 2.44948974,
                1.73205081])

In [70]: np.exp(a) # exponential of each element in the array
```

```
Out[70]: array([2.20264658e+04, 1.48413159e+02, 7.38905610e+00, 2.71828183e+00,
                4.03428793e+02, 2.00855369e+01])

In [71]: np.power(a, 2) # nth power of each element in the list

Out[71]: array([100,  25,   4,   1,  36,   9], dtype=int32)

In [72]: np.log(a) # log base e of each element in the list

Out[72]: array([2.30258509, 1.60943791, 0.69314718, 0.        , 1.79175947,
                1.09861229])

In [73]: np.log10(a) # log base e of each element in the list

Out[73]: array([1.        , 0.69897   , 0.30103   , 0.        , 0.77815125,
                0.47712125])

In [74]: np.divide(a, 2) # divide each element with n

Out[74]: array([5. , 2.5, 1. , 0.5, 3. , 1.5])

In [75]: np.sum(a)

Out[75]: 27
```

3.4.2 With another array

```
In [76]: np.add(a, b)

Out[76]: array([15. ,  5.5, 10. ,  2. ,  9. , 10. ])

In [77]: a + b # element-wise addition

Out[77]: array([15. ,  5.5, 10. ,  2. ,  9. , 10. ])

In [78]: np.subtract(a, b)

Out[78]: array([ 5. ,  4.5, -6. ,  0. ,  3. , -4. ])

In [79]: a - b # element-wise addition

Out[79]: array([ 5. ,  4.5, -6. ,  0. ,  3. , -4. ])

In [80]: np.multiply(a, b)

Out[80]: array([50. ,  2.5, 16. ,  1. , 18. , 21. ])

In [81]: a*b # element-wise multiplication
```

```
Out[81]: array([50. ,  2.5, 16. ,  1. , 18. , 21. ])

In [82]: np.divide(a, b)

Out[82]: array([ 2.        , 10.        ,  0.25      ,  1.        ,  2.        ,
                 0.42857143])

In [83]: a / b

Out[83]: array([ 2.        , 10.        ,  0.25      ,  1.        ,  2.        ,
                 0.42857143])

In [84]: np.dot(a, b) # matrix multiplication

Out[84]: 108.5

In [85]: np.dot(a.reshape(2, 3), b.reshape(3, 2)) # matrix multiplication

Out[85]: array([[96. , 24. ],
                [62. , 27.5]])
```

3.5 Array Manipulation

3.5.1 On an array

Array slicing / subsetting

```
In [86]: a

Out[86]: array([10,  5,  2,  1,  6,  3])

In [87]: # Getting data using index
         a[0]

Out[87]: 10

In [88]: # Slicing
         a[:3]

Out[88]: array([10,  5,  2])

In [89]: # boolean indexing
         a[a > 5]

Out[89]: array([10,  6])

In [90]: a > 5 # passed to the array as an index. Elements with boolean index false are dropped

Out[90]: array([ True, False, False, False,  True, False])

In [91]: array_

Out[91]: array([8, 8, 1, 4, 0, 7, 4, 0, 8, 6])

In [92]: comp

Out[92]: [[0, 1, 2, 3], 3, 5, 7, 3]
```

Insert

```
In [93]: np.insert(arr=array_, obj=0, values=[1, 2, 3])

Out[93]: array([1, 2, 3, 8, 8, 1, 4, 0, 7, 4, 0, 8, 6])

In [94]: comp.insert(0, [1, 2])
         comp

Out[94]: [[1, 2], [0, 1, 2, 3], 3, 5, 7, 3]
```

Delete

```
In [95]: array_

Out[95]: array([8, 8, 1, 4, 0, 7, 4, 0, 8, 6])

In [96]: np.delete(array_, [1, 2, 3])  # elements with index 1, 2, 3 are deleted

Out[96]: array([8, 0, 7, 4, 0, 8, 6])
```

To remove specific values from an array

```
In [97]: np.delete(array_, np.where(array_==7), axis=0)

Out[97]: array([8, 8, 1, 4, 0, 4, 0, 8, 6])
```

Resize

- to resize an array into desired dimension
- it changes the data itself

```
In [98]: array_.resize(2, 5)  # it changes the data itself

In [99]: array_  # the array_ object itself is modified

Out[99]: array([[8, 8, 1, 4, 0],
                [7, 4, 0, 8, 6]])
```

Reshape

- to reshape an 1D array into desired dimension
- it doesn't change the data

```
In [100]: array_ = np.array([1, 2, 9, 0, 9, 9, 9, 5, 2, 3])
          array_

Out[100]: array([1, 2, 9, 0, 9, 9, 9, 5, 2, 3])
```

```
In [101]: array_.reshape(2, 5)

Out[101]: array([[1, 2, 9, 0, 9],
                 [9, 9, 5, 2, 3]])

In [102]: array_  # the array_ object remains unmodified

Out[102]: array([1, 2, 9, 0, 9, 9, 9, 5, 2, 3])
```

This is the only difference between resize and reshape

```
In [103]: array_ = np.arange(1, 10, step=0.1).reshape(9, 10)
          array_

Out[103]: array([[1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9],
                 [2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9],
                 [3. , 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9],
                 [4. , 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9],
                 [5. , 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9],
                 [6. , 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9],
                 [7. , 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, 7.9],
                 [8. , 8.1, 8.2, 8.3, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9],
                 [9. , 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 9.7, 9.8, 9.9]])

In [104]: array_.reshape(-1, 1)[:5] # n elements in the array is transformed into an array with n
          rows and 1 column

Out[104]: array([[1. ],
                 [1.1],
                 [1.2],
                 [1.3],
                 [1.4]])
```

The above technique is used for transforming target values in 1-D horizontal array to a 1-D vertical array before passing it to the machine learning models. The -1 represents size of an array which is n.

```
In [105]: array_.reshape(array_.size, 1)[:5] # n elements in the array is transformed into an
          array with n rows and 1 column

Out[105]: array([[1. ],
                 [1.1],
                 [1.2],
                 [1.3],
                 [1.4]])
```

Transpose

```
In [106]: array_.T
```

```
Out[106]: array([[1. , 2. , 3. , 4. , 5. , 6. , 7. , 8. , 9. ],
                 [1.1, 2.1, 3.1, 4.1, 5.1, 6.1, 7.1, 8.1, 9.1],
                 [1.2, 2.2, 3.2, 4.2, 5.2, 6.2, 7.2, 8.2, 9.2],
                 [1.3, 2.3, 3.3, 4.3, 5.3, 6.3, 7.3, 8.3, 9.3],
                 [1.4, 2.4, 3.4, 4.4, 5.4, 6.4, 7.4, 8.4, 9.4],
                 [1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5, 9.5],
                 [1.6, 2.6, 3.6, 4.6, 5.6, 6.6, 7.6, 8.6, 9.6],
                 [1.7, 2.7, 3.7, 4.7, 5.7, 6.7, 7.7, 8.7, 9.7],
                 [1.8, 2.8, 3.8, 4.8, 5.8, 6.8, 7.8, 8.8, 9.8],
                 [1.9, 2.9, 3.9, 4.9, 5.9, 6.9, 7.9, 8.9, 9.9]])
```

Ravel

- flattens a multidimensional array
- sometimes it is mandatory to flatten an array before passing it to a machine learning algorithm (target values)

```
In [107]: np.ravel(array_) # from n-D to 1-D
```

```
Out[107]: array([1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2. , 2.1, 2.2,
                 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 3. , 3.1, 3.2, 3.3, 3.4, 3.5,
                 3.6, 3.7, 3.8, 3.9, 4. , 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8,
                 4.9, 5. , 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, 6. , 6.1,
                 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9, 7. , 7.1, 7.2, 7.3, 7.4,
                 7.5, 7.6, 7.7, 7.8, 7.9, 8. , 8.1, 8.2, 8.3, 8.4, 8.5, 8.6, 8.7,
                 8.8, 8.9, 9. , 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 9.7, 9.8, 9.9])
```

3.5.2 On multiple arrays

```
In [108]: array1 = np.arange(1, 11, 1).reshape(2, 5)
          array2 = np.arange(11, 16, 1).reshape(1, 5)
```

```
In [109]: print(array1)
          print(array2)
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]]
[[11 12 13 14 15]]
```

Concatenate

- to concatenate 2 or more arrays

```
In [110]: array3 = np.concatenate((array1, array2), axis=0)  # row-wise operation
          array3

Out[110]: array([[ 1,  2,  3,  4,  5],
                 [ 6,  7,  8,  9, 10],
                 [11, 12, 13, 14, 15]])

In [111]: np.concatenate([array3, array2.T[:3]], axis=1)  # column-wise operation

Out[111]: array([[ 1,  2,  3,  4,  5, 11],
                 [ 6,  7,  8,  9, 10, 12],
                 [11, 12, 13, 14, 15, 13]])
```

The concatenation operations shown above for row-wise concatenation and column-wise concatenation can be also achieved using vstack and hstack operations, respectively.

```
In [112]: np.vstack((array1, array2))

Out[112]: array([[ 1,  2,  3,  4,  5],
                 [ 6,  7,  8,  9, 10],
                 [11, 12, 13, 14, 15]])

In [113]: np.hstack([array3, array2.T[:3]])

Out[113]: array([[ 1,  2,  3,  4,  5, 11],
                 [ 6,  7,  8,  9, 10, 12],
                 [11, 12, 13, 14, 15, 13]])
```

To combine 2 arrays vertically without using transpose

```
In [114]: np.column_stack((array1[0], array2[0]))

Out[114]: array([[ 1, 11],
                 [ 2, 12],
                 [ 3, 13],
                 [ 4, 14],
                 [ 5, 15]])
```

Splitting arrays

- To split an array into two in the desired format. This is just the reverse operation to array concatenations.

Vertical split Vertical split is similar to cutting an array along axis = 1 which is the first axis (similar to split with axis = 0).

```
In [115]: np.vsplit(array1, 2)
```

```
Out[115]: [array([[1, 2, 3, 4, 5]]), array([[ 6,  7,  8,  9, 10]])]

In [116]: np.vsplit(array3, 3)

Out[116]: [array([[1, 2, 3, 4, 5]]),
           array([[ 6,  7,  8,  9, 10]]),
           array([[11, 12, 13, 14, 15]])]
```

Horizontal split Horizontal split is similar to cutting an array along axis = 0 which is the second axis (similar to split with axis = 1)

```
In [117]: array4 = np.column_stack((array1[0], array2[0]))
          array4

Out[117]: array([[ 1, 11],
                 [ 2, 12],
                 [ 3, 13],
                 [ 4, 14],
                 [ 5, 15]])

In [118]: np.hsplit(array4, 2)

Out[118]: [array([[1],
                  [2],
                  [3],
                  [4],
                  [5]]), array([[11],
                  [12],
                  [13],
                  [14],
                  [15]])]
```

The above operation gives 2 vertical arrays as the output.

4

PANDAS

Pandas is one of the most important modules in python that every aspiring data scientist should master. It is widely used among data scientists during almost every step in a data science project. Pandas is primarily used for data munging/wrangling or ETL operation (Extract-Transform-Load) but also has several functions for basic data exploration. Pandas has several build-in functions that are optimized to run fast as well as easy to use. Pandas allows us to develop proof-of-concept of our ideas in a much shorter time than what it would normally take without it.

The data structures that are mainly used in Pandas are DataFrames and Series. The Pandas DataFrame is similar to a MS excel table with two-dimensions (columns and rows). We could directly apply build-in/custom functions, on the rows or the columns, for data exploration, data cleaning, data transformation, and so on.

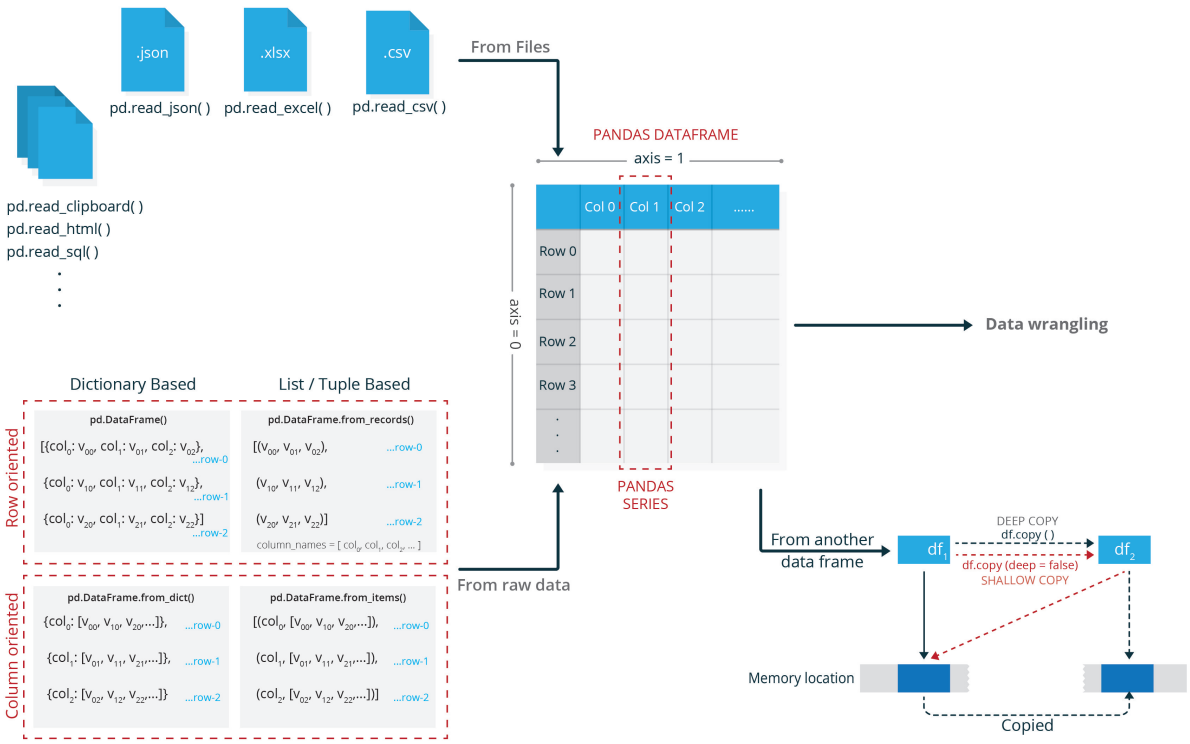
Topics:

I have categorized Pandas's functionality into 8 based on it's application in data science.

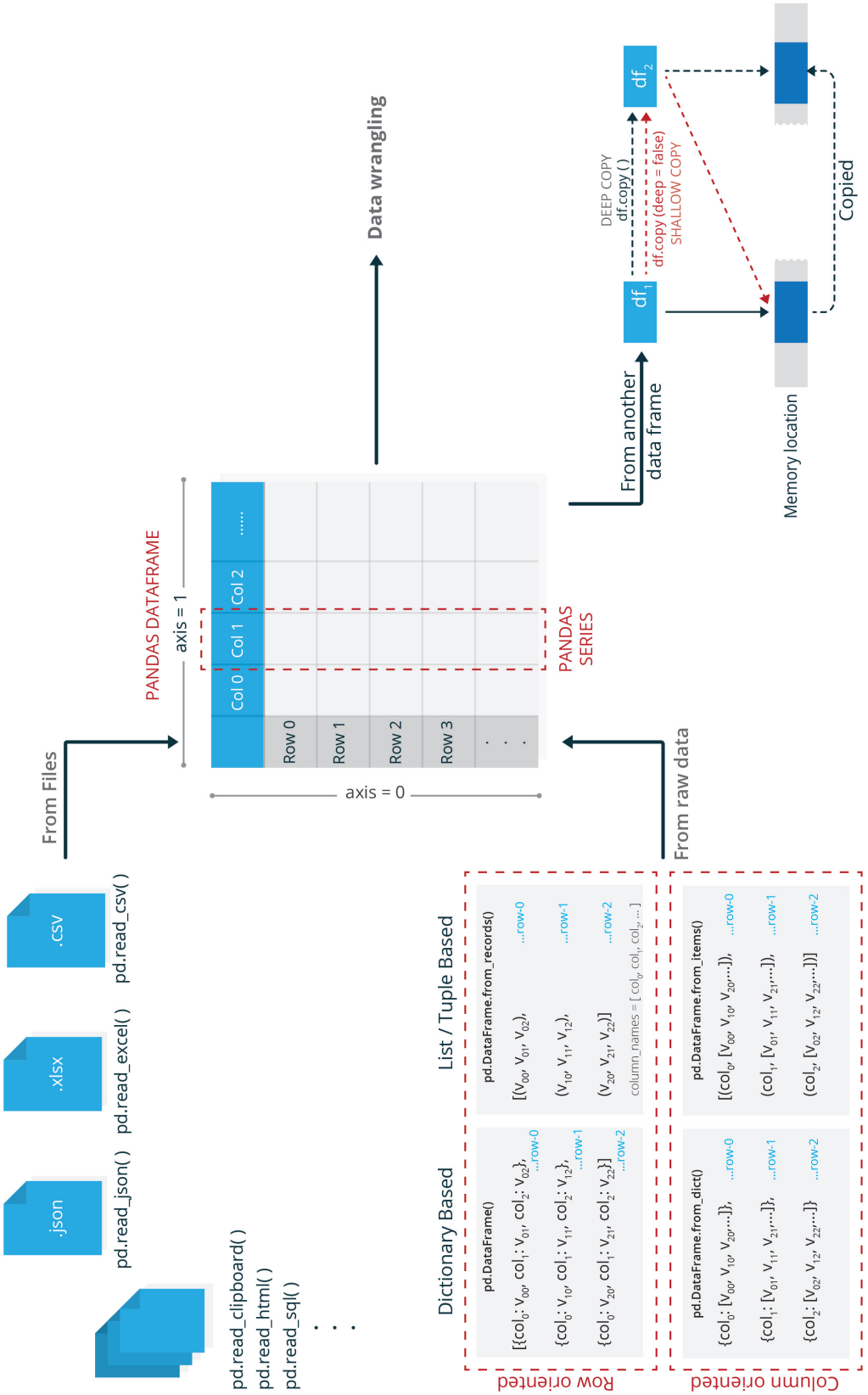
1. Create a DataFrame
2. Data exploration
3. Data visualization
4. Data cleaning
5. Data selection
6. Data manipulation
7. Data transformation
8. Save Data

```
In [1]: # Lets import the required mdoules for loading data

        import Pandas as pd          # loads Pandas module
        import os                    # os module to get anything realted to operating system
        import json                  # module that support json
        import numpy as np           # numpy to handle arrays
        import sys                           # for OS related functionalities
        import matplotlib.pyplot as plt      # visualization
        import traceback                     # to print exceptions
        import time                          # time module to get time and other related tasks
        import warnings                      # module to handle warnings

        warnings.simplefilter("ignore")      # filters out the warnings
```

4.1 Create a DataFrame



Methods to create a DataFrame



Methods to create a DataFrame

4.1.1 From files

Pandas has the capability to read data from any file format in which data are usually stored. The data are generally saved in formats such as .csv, .xlsx (excel), pickle, json, html, etc. Surprisingly, we can also read contents that are copied to our clipboard.

For practice lets download pima-indians dataset from UCI Machine Learning repository (open source). Link: https://archive.ics.uci.edu/ml/datasets/pima+indians+diabetes

Please download the dataset, place it in a folder of your choice, and add the path to path variable as below

```
In [3]:  path = "../data/"
```

Attributes of the data:

1. Number of times pregnant
2. Plasma glucose concentration a 2 hours in an oral glucose tolerance test
3. Diastolic blood pressure (mm Hg)
4. Triceps skin fold thickness (mm)
5. 2-Hour serum insulin (mu U/ml)
6. Body mass index (weight in kg/(height in m)^2)
7. Diabetes pedigree function
8. Age (years)
9. Class variable (0 or 1)

The data set comes with no labels (column names) in the .csv file that you have just downloaded. So let's compile a list of labels to use for creating a DataFrame.

```
In [4]: # Column names
        cols = ['preg_count', 'glucose', 'BP', 'skin_thick', 'insulin', 'BMI', 'pedigree',
        'age', 'class']
```

1. Read the data in .csv format

```
In [5]: df = pd.read_csv(path + "pima-indians-diabetes.csv", header=None, names=cols)
        df.head()
```

Out[5]:

	preg_count	glucose	BP	skin_thick	insulin	BMI	pedigree	age	class
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

If there is already a row with labels (columns names) in the file, we can just give the row number to the <"header"> variable in the above read function. The default is 0.

```
In [6]: # list column names
        df.columns
```

```
Out[6]: Index(['preg_count', 'glucose', 'BP', 'skin_thick', 'insulin', 'BMI',
               'pedigree', 'age', 'class'],
              dtype='object')
```

```
In [7]: # rename columns
        df.columns = ['preg_count', 'glucose', 'BP', 'skin_thick', 'insulin', 'BMI', 'pedigree',
        'age', 'class']
```

2. Read the data in .xlsx format

```
In [8]: df = pd.read_excel(path + "pima_indians.xlsx")
        df.head()
```

Out[8]:

	preg_count	glucose	BP	skin_thick	insulin	BMI	pedigree	age	class
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

3. Read the data in json format

```
In [9]: df = pd.read_json(path + "pima_indians.json")
        df.head()
```

Out[9]:

	BMI	BP	age	class	glucose	insulin	pedigree	preg_count	skin_thick
0	33.6	72	50	1	148	0	0.627	6	35
1	26.6	66	31	0	85	0	0.351	1	29
10	37.6	92	30	0	110	0	0.191	4	0
100	39.0	72	33	1	163	0	1.222	1	0
101	26.1	60	22	0	151	0	0.179	1	0

Note: JSON is a serialized dictionary. Both json and dictionary are similar in format. We can access/select the data the same way we access/select in a multi-level dictionary. Simply, dictionary is a data structure whereas JSON is a data format. The dictionary cannot be stored as a file as it is an abstract object. The JSON format is created to store dictionaries as a serialized object and transfer to different nodes over network on request. Another major difference between them is that the dictionaries can have keys in any datatype like string, int, float, etc., whereas JSON can have only string as keys.

```
In [10]: # open() creates a file object that will be used by json load and "with" ensures that
         the file will be closed after
         with open(path + "pima_indians.json") as jsonfile:
             data = json.load(jsonfile)
```

```
In [11]: data['BMI']['0']
```

```
Out[11]: 33.6
```

```
In [12]: # 4. Read from a pickle file
         df = pd.read_pickle(path + "df.pkl")
         df.head()
```

Out[12]:

	preg_count	glucose	BP	skin_thick	insulin	BMI	pedigree	age	class
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Pickle is one of the most commonly used data format to store an object in a file. It is used to serialize and then deserialize an object. In data science, objects such as DataFrame, trained machine learning model, config file, etc., are saved as pickle file which can be easily loaded (unpickled) whenever needed.

4.1.2 Reading from variables

DataFrames can also be created by directly passing data in certain formats (as listed below).

1. Dictionary form - row oriented:

```
In [13]: dict_row = [{'BMI': 21.1, 'insulin': 50, 'BP': 66},
                     {'BMI': 24.4, 'insulin': 72, 'BP': 54},
                     {'BMI': 20.9, 'insulin': 24, 'BP': 78}]

         pd.DataFrame(dict_row)
```

Out[13]:

	BMI	BP	insulin
0	21.1	66	50
1	24.4	54	72
2	20.9	78	24

2. Dictionary form - column oriented:

```
In [14]: dict_column = {'BMI': [21.1, 24.4, 20.9],
                        'insulin': [50, 72, 24],
                        'BP': [66, 54, 78]}

         pd.DataFrame.from_dict(dict_column)
```

Out[14]:

	BMI	BP	insulin
0	21.1	66	50
1	24.4	54	72
2	20.9	78	24

3. List form - row oriented:

```
In [15]: records_ = [(21.1, 50, 66),
                     (24.4, 72, 54),
                     (20.9, 24, 78)]

         column_labels = ['BMI', 'insulin', 'BP']
         pd.DataFrame.from_records(records_, columns=column_labels)
```

```
Out[15]:     BMI  insulin  BP
         0  21.1       50  66
         1  24.4       72  54
         2  20.9       24  78
```

4. List form - column oriented:

```
In [16]: item_ = [('BMI', [21.1, 24.4, 20.9]),
                  ('insulin', [50, 72, 24]),
                  ('BP', [66, 54, 78])]

         pd.DataFrame.from_items(item_)
```

```
Out[16]:     BMI  insulin  BP
         0  21.1       50  66
         1  24.4       72  54
         2  20.9       24  78
```

4.1.3 Copying a DataFrame

To test new methods (or techniques) we may have to make a copy of an existing DataFrame so that the actual data remains unchanged even if the method/technique didn't work as expected.

There are 3 methods to copy a DataFrame.

Simple copy

- It is just like copying a variable but their objects remain the same

```
In [17]: df1 = df # copies the DataFrame df to df1 with the same object ID (binds to the same
         object)

         print(id(df)) # prints the object ID
         print(id(df1))
```

```
2201019746400
2201019746400
```

Making any changes to df1 will reflect on df as well.

Shallow copy

- Only structure of the parent DataFrame is copied. The data and indices for both the DataFrames points to the same memory location. So modifying the objects in one DataFrame will reflect on the other DataFrame.

```
In [18]: df2 = df.copy(deep=False) # shallow copy

         print(id(df))
         print(id(df2)) # New object is created but the data points to the location of the df's
         data location

         # df and df2 points to the same memory location for data
```

```
2201019746400
2201091484976
```

Deep copy

- Both the data and the indices are copied and stored in a different memory location. So the objects in the new DataFrame points to a different memory location that that of the parent DataFrame. The DataFrames are totally disconnected, the changes we make on one DataFrame will not get reflected on the other.

```
In [19]: # To avoid the above issue, we can use copy() method (as follows)
         df1 = df.copy(deep=True) # deep copy

         print(id(df))
         print(id(df1))
```

```
2201019746400
2201091718560
```

To clarify further, the difference between shallow and deep copy - the deep copy creates new id for each object (by copying it to a different memory location) that is stored in a DataFrame whereas the shallow copy merely copies the ids from the parent DataFrame to a new variable (with different id).

4.1.4 Pandas Series

```
In [20]: # Create a Pandas series from a list
         dfs = pd.Series([5, 6, 7, 12, 34, 'hello', 'john'], name = "info") # - Creates a series
         from a list
         dfs

Out[20]: 0        5
         1        6
```

```
2        7
3       12
4       34
5    hello
6     john
Name: info, dtype: object
```

```
In [21]: # Each column in a DataFrame is a Pandas series
         df['BMI'][:5]
```

```
Out[21]: 0    33.6
         1    26.6
         2    23.3
         3    28.1
         4    43.1
         Name: BMI, dtype: float64
```

```
In [22]: type(df['BMI'])
```

```
Out[22]: Pandas.core.series.Series
```

Note:

```
In [23]: # If we use double square braces instead of single one then it refers to a DataFrame
         print(type(df['BMI']))
         print(type(df[['BMI']]))
```

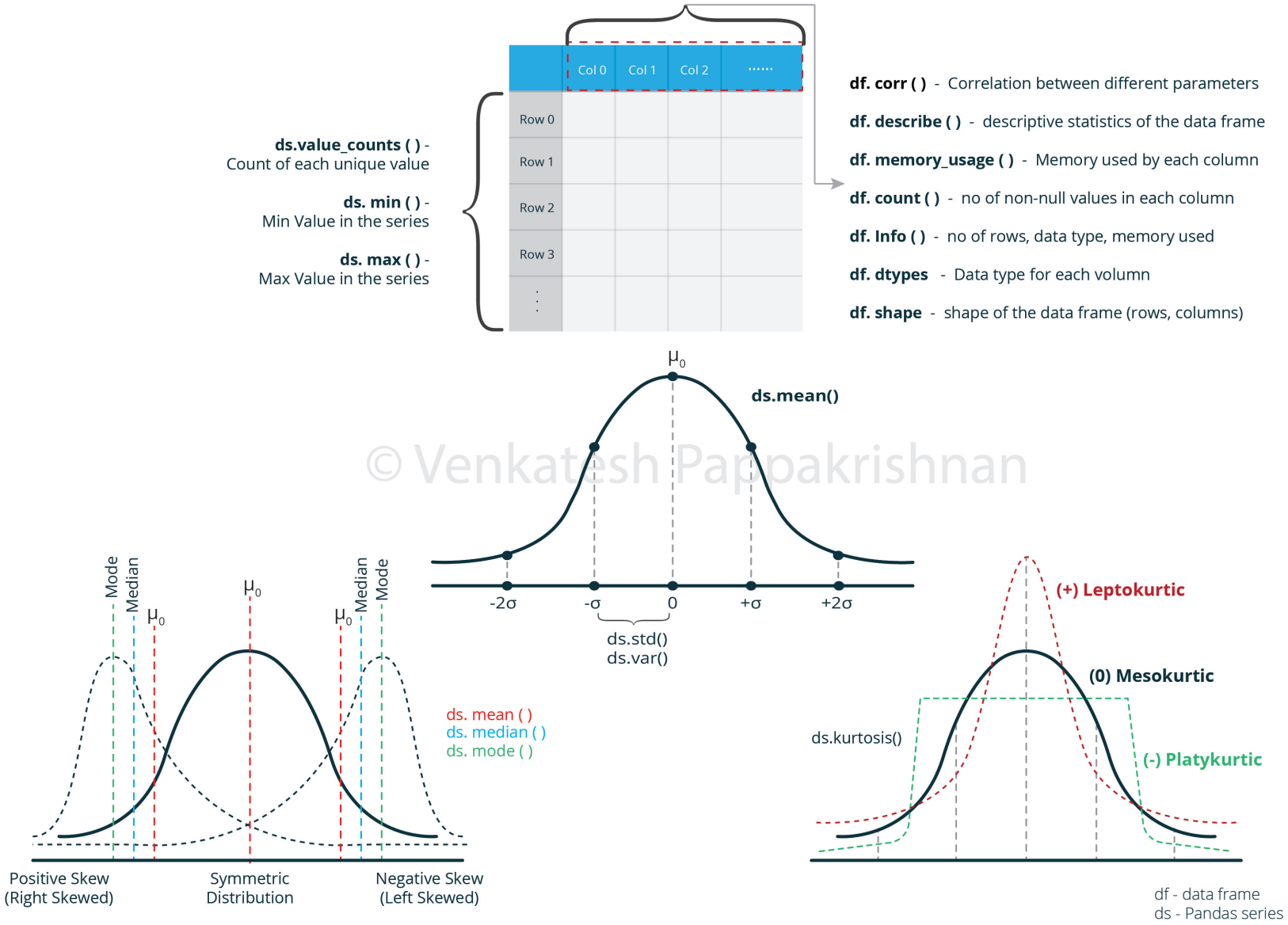
```
<class 'Pandas.core.series.Series'>
<class 'Pandas.core.frame.DataFrame'>
```

4.2 Data Exploration

It is imperative to understand the given data before taking any steps to clean it. A bit of domain knowledge related to the data is definitely critical to distinguish between acceptable and unacceptable values. However, basic exploration and cleaning can be still performed with minimal domain knowledge but at the risk of loosing important information (possibly contained in outliers).

There are some basic statistical functions that are the in-built in Pandas to carry out this process. To give you a flavor of how data exploration is done, I will show you how to find basic information about the dataset such as dtypes, size, shape, etc., descriptive statistics, correlation coefficient matrix, and different statistical moments.

This is one of the most time consuming steps in a data science project. "Garbage-in Garbage-out" - It is important to use clean data for modeling. However, it is not a straight line from point A (unclean data) to point B (clean data). We can only clean the bad data only if we know how to find that in the first place. For some datasets, it is trivial to find bad data and clean them but for most of the real datasets that you will be using in projects, it is non-trivial to spot the bad data points. You may have to apply various techniques to find them.



Data exploration techniques

For statistical analysis and inference, normality for the data is one of the main assumptions. However, it is not always the case with the data. In such cases, we may have to transform the data such that it's distribution becomes close to bell curve. When the data are skewed left or right (refer to the above picture) we can apply simple transformation techniques based on **Tukey's ladder of power**

Similarly, for kurtosis analysis (data spikiness) when the data doesn't fall under mesokurtosis then a custom mathematical function should be applied to expand or shrink the space around mean to make it close to mesokurtic state.

Shape: Dimension of a DataFrame

```
In [24]: # The first element represents number of rows whereas the second element represents
         number of columns
         df.shape

Out[24]: (768, 9)
```

rows - 768 columns (index not included) - 9

Data types:

```
In [25]: # data types for each column in the DataFrame
         df.dtypes

Out[25]: preg_count      int64
         glucose         int64
         BP              int64
         skin_thick      int64
         insulin         int64
         BMI           float64
         pedigree      float64
         age             int64
         class           int64
         dtype: object
```

Information: Info gives more information than 'dtypes' such as number of non-null elements for each column/feature and memory used by the DataFrame.

```
In [26]: df.info()

<class 'Pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
preg_count    768 non-null int64
glucose       768 non-null int64
BP            768 non-null int64
skin_thick    768 non-null int64
```

```
insulin       768 non-null int64
BMI           768 non-null float64
pedigree      768 non-null float64
age           768 non-null int64
class         768 non-null int64
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

Memory usage:

```
In [27]: df.memory_usage() # in bytes

Out[27]: Index           80
         preg_count    6144
         glucose       6144
         BP            6144
         skin_thick    6144
         insulin       6144
         BMI           6144
         pedigree      6144
         age           6144
         class         6144
         dtype: int64

In [28]: df.memory_usage().sum()

Out[28]: 55376
```

Value Counts:

- provides the frequency of each unique value in the column

This makes senses only for categorical or nominal values

```
In [29]: df['preg_count'].value_counts().sort_index()

Out[29]: 0     111
         1     135
         2     103
         3      75
         4      68
         5      57
         6      50
         7      45
         8      38
         9      28
```

```
10     24
11     11
12      9
13     10
14      2
15      1
17      1
Name: preg_count, dtype: int64
```

For instance, the number of people with 0 pregnancy count is 111 and so on

4.2.1 Statistical operations

Summary statistics

```
In [30]: # To get statistics for all the columns at the same time
         df.describe()
```

Out[30]:

	preg_count	glucose	BP	skin_thick	insulin	BMI	\
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	

	pedigree	age	class
count	768.000000	768.000000	768.000000
mean	0.471876	33.240885	0.348958
std	0.331329	11.760232	0.476951
min	0.078000	21.000000	0.000000
25%	0.243750	24.000000	0.000000
50%	0.372500	29.000000	0.000000
75%	0.626250	41.000000	1.000000
max	2.420000	81.000000	1.000000

Categorical variables are ignored for the above table

Statistical moments

1. Mean (1st moment)
2. Variance (2nd moment)
3. Skewness (3rd moment)
4. Kurtosis (4th moment)

1. Mean

```
In [31]: # Applying mean() to the DataFrame returns mean of each column (Pandas series)
         df.mean()

Out[31]: preg_count      3.845052
         glucose       120.894531
         BP             69.105469
         skin_thick     20.536458
         insulin        79.799479
         BMI            31.992578
         pedigree        0.471876
         age            33.240885
         class           0.348958
         dtype: float64

In [32]: df['preg_count'].mean() # returns the mean of 'preg_count' column

Out[32]: 3.8450520833333335

In [33]: # Mean of each row
         df.mean(axis=1)[:5]

Out[33]: 0    38.469667
         1    26.550111
         2    34.663556
         3    35.807444
         4    51.043111
         dtype: float64
```

2. Variance and standard deviation

```
In [34]: df.var()

Out[34]: preg_count       11.354056
         glucose        1022.248314
         BP              374.647271
         skin_thick      254.473245
         insulin       13281.180078
         BMI              62.159984
         pedigree          0.109779
         age             138.303046
         class             0.227483
         dtype: float64

In [35]: df.std()
```

```
Out[35]: preg_count      3.369578
         glucose        31.972618
         BP             19.355807
         skin_thick     15.952218
         insulin       115.244002
         BMI             7.884160
         pedigree        0.331329
         age            11.760232
         class           0.476951
         dtype: float64
```

3. Skewness Skewness is the measure of the symmetry of a distribution compared to standard normal distribution

- +ive - right skewed (mean is to the right of mode/median). Long tail in the +ive direction.
- 0 - symmetric
- -ive - left skewed (mean is to the left of mode/median). Long tail in the -ive direction.

```
In [36]: df.skew()
```

```
Out[36]: preg_count    0.901674
         glucose       0.173754
         BP           -1.843608
         skin_thick    0.109372
         insulin       2.272251
         BMI          -0.428982
         pedigree      1.919911
         age           1.129597
         class         0.635017
         dtype: float64
```

4. Kurtosis Kurtosis is a measure of the flatness or peakedness of a distribution compared to the normal distribution.

- +ive - Leptokurtosis (sharper/spikier peak compared to the normal dist.)
- 0 - Mesokurtic (normal dist.)
- -ive - Platykurtic (flatter peak compared to the normal dist.) eg. Uniform distribution

```
In [37]: df.kurtosis()
```

```
Out[37]: preg_count    0.159220
         glucose       0.640780
         BP            5.180157
         skin_thick   -0.520072
         insulin       7.214260
         BMI           3.290443
```

```
pedigree      5.594954
age           0.643159
class        -1.600930
dtype: float64
```

min / max / median

```
In [38]: # min of each column
         df.min()

Out[38]: preg_count     0.000
         glucose        0.000
         BP             0.000
         skin_thick     0.000
         insulin        0.000
         BMI            0.000
         pedigree       0.078
         age           21.000
         class          0.000
         dtype: float64

In [39]: # max of each column
         df.max()

Out[39]: preg_count     17.00
         glucose       199.00
         BP            122.00
         skin_thick     99.00
         insulin       846.00
         BMI            67.10
         pedigree        2.42
         age            81.00
         class           1.00
         dtype: float64

In [40]: # median of each column
         df.median()

Out[40]: preg_count      3.0000
         glucose       117.0000
         BP             72.0000
         skin_thick     23.0000
         insulin        30.5000
         BMI            32.0000
         pedigree        0.3725
         age            29.0000
         class           0.0000
         dtype: float64
```

Correlation

```
In [41]: df.corr()
```

```
Out[41]:             preg_count   glucose        BP  skin_thick   insulin       BMI  \
         preg_count    1.000000  0.129459  0.141282   -0.081672 -0.073535  0.017683
         glucose       0.129459  1.000000  0.152590    0.057328  0.331357  0.221071
         BP            0.141282  0.152590  1.000000    0.207371  0.088933  0.281805
         skin_thick   -0.081672  0.057328  0.207371    1.000000  0.436783  0.392573
         insulin      -0.073535  0.331357  0.088933    0.436783  1.000000  0.197859
         BMI           0.017683  0.221071  0.281805    0.392573  0.197859  1.000000
         pedigree     -0.033523  0.137337  0.041265    0.183928  0.185071  0.140647
         age           0.544341  0.263514  0.239528   -0.113970 -0.042163  0.036242
         class         0.221898  0.466581  0.065068    0.074752  0.130548  0.292695

                      pedigree       age     class
         preg_count  -0.033523  0.544341  0.221898
         glucose      0.137337  0.263514  0.466581
         BP           0.041265  0.239528  0.065068
         skin_thick   0.183928 -0.113970  0.074752
         insulin      0.185071 -0.042163  0.130548
         BMI          0.140647  0.036242  0.292695
         pedigree     1.000000  0.033561  0.173844
         age          0.033561  1.000000  0.238356
         class        0.173844  0.238356  1.000000
```

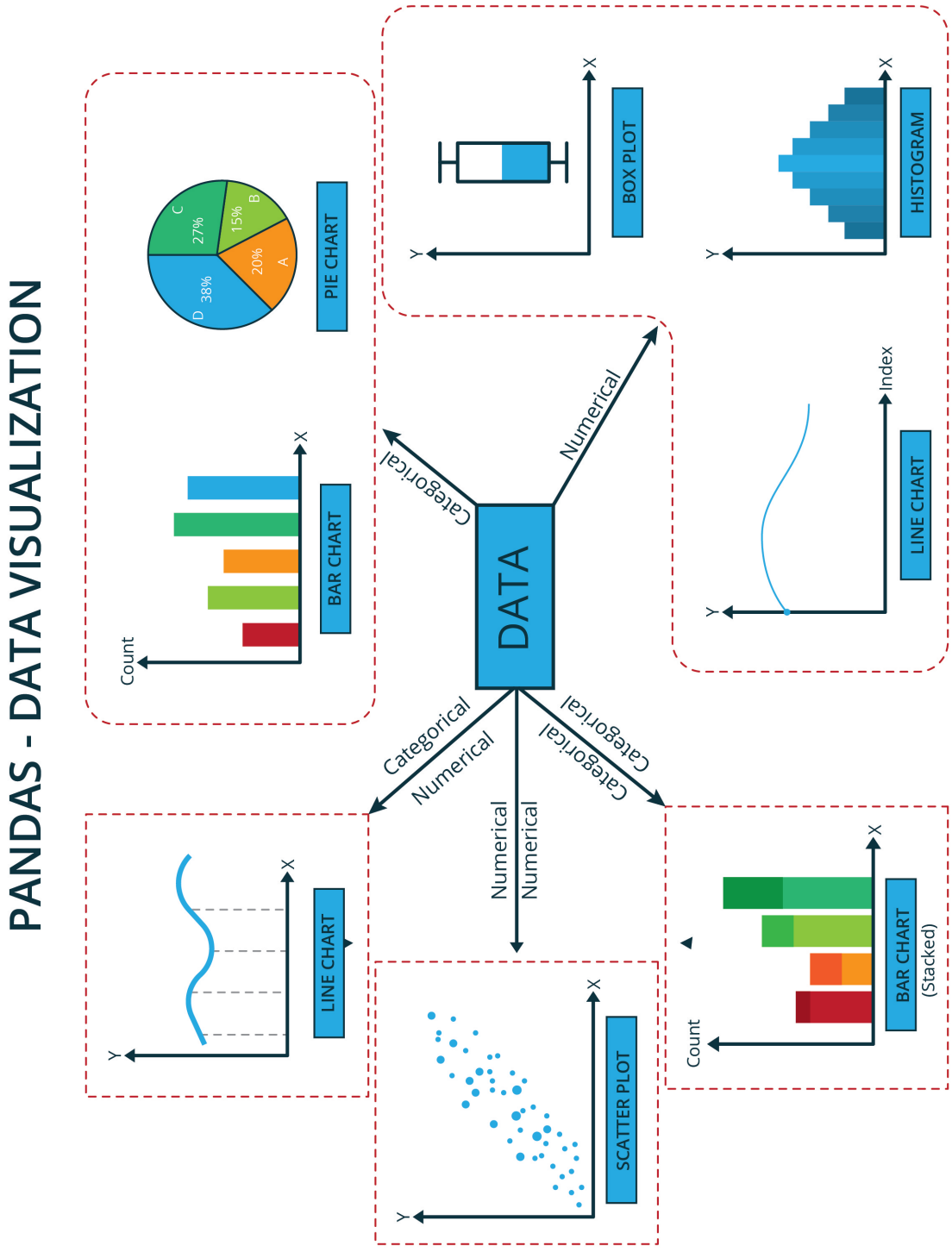
The correlation matrix provides the correlation coefficient between each of the columns in the DataFrame. The default is pearson's correlation coefficient method but we can also choose 'kendall' or 'spearman' based on the datatype.

4.3 Data Visualization

Data visualization is critical for data exploration to better understand what lies within the data. It is also important for presenting your results or findings to anyone, even to people with no technical background. Using appropriate visual aid it is easy to make anyone understand the analysis and results without which it will be definitely challenging to convey the message.

"A picture is worth a ten thousand words" – Fred R. Barnard

In this section, we are going to use visualizations only for data exploration. Visualization as a tool for presentation is out of the scope of this book but, if possible, I will discuss in another book that is solely dedicated for presenting the data to any audience.



Data visualization

Please pick a GUI of your interest or one that is suitable for the environment to display the plots.
Use %matplotlib to activate matplotlib interactive support. The % represents a magic command in ipython notebook.

```
In [42]: # this is to display the plot in the jupyter notebook itself
         % matplotlib inline

In [43]: # uses the default which is TkAgg backend
         %matplotlib

         # uses a different backend
         %matplotlib qt
```

```
Using matplotlib backend: Qt5Agg
```

```
In [44]: # List of backends that can be used
         %matplotlib --list
```

```
Available matplotlib backends: ['tk', 'gtk', 'gtk3', 'wx', 'qt4', 'qt5', 'qt', 'osx', 'nbagg',
 'notebook', 'agg', 'svg', 'pdf', 'ps', 'inline', 'ipympl', 'widget']
```

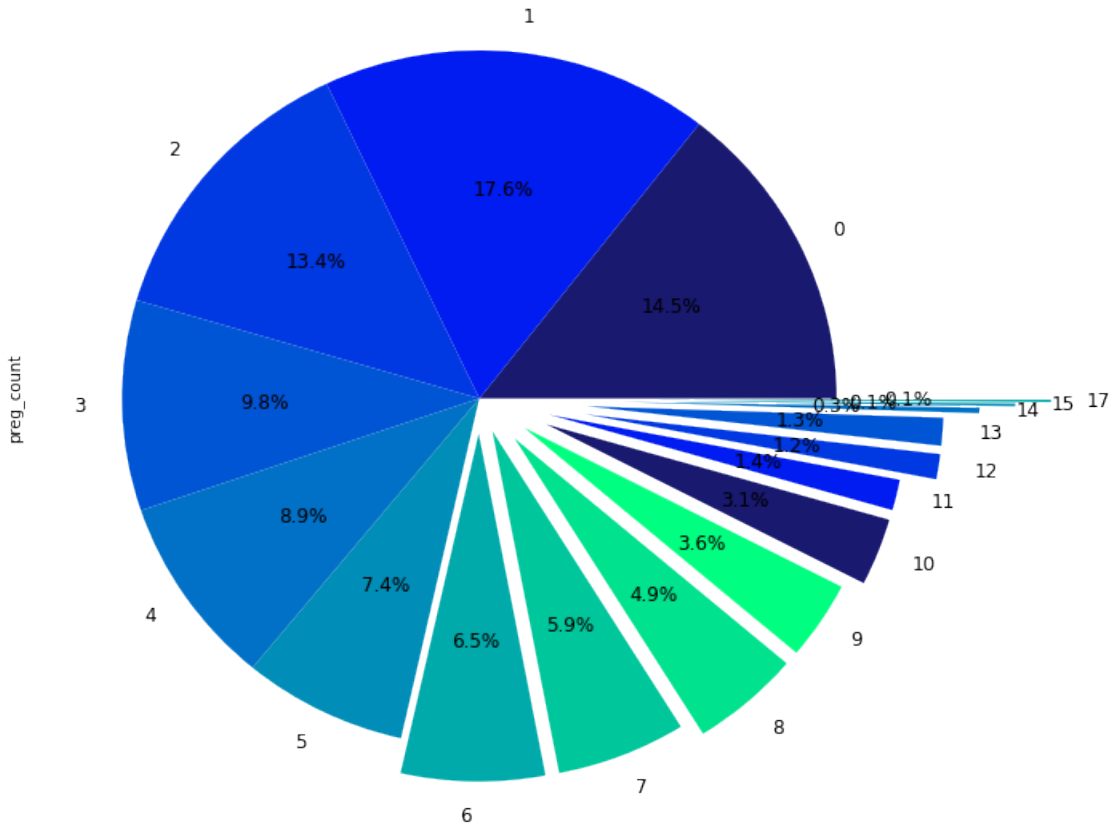
Please explore the various options if you are interested

4.3.1 Piechart

```
In [45]: % matplotlib inline

In [46]: df['preg_count'].value_counts().sort_index()
```

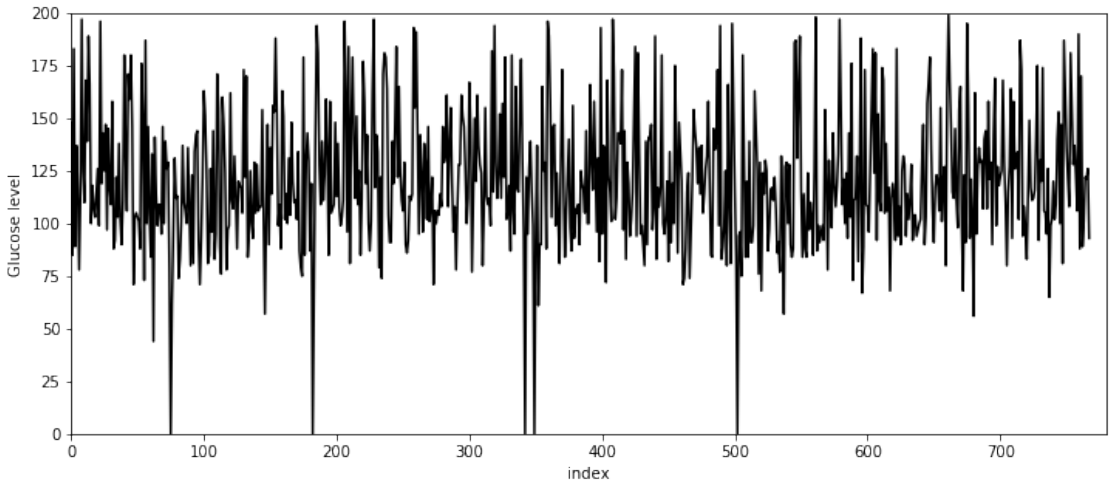
```
Out[46]: 0     111
         1     135
         2     103
         3      75
         4      68
         5      57
         6      50
         7      45
         8      38
         9      28
         10     24
         11     11
         12      9
         13     10
         14      2
         15      1
         17      1
         Name: preg_count, dtype: int64
```

```
In [47]: colors = ['#191970', '#001CF0', '#0038E2', '#0055D4', '#0071C6', '#008DB8', '#00AAAA',
                   '#00C69C', '#00E28E', '#00FF80', ]

         explode = (0, 0, 0, 0, 0, 0, 0.1, 0.1, 0.15, 0.15, 0.2, 0.2, 0.3, 0.3, 0.4, 0.5, 0.6)

In [48]: dff = df['preg_count'].value_counts().sort_index()

         dff.plot(kind='pie', colors=colors, explode=explode, autopct='%1.1f%%', fontsize=12,
         figsize=(10, 10))

         # All the options are self explanatory
         # Kind - type of plot
         # colors - colors for each pie
         # explode - distance from the centre and so on

Out[48]: <matplotlib.axes._subplots.AxesSubplot at 0x2007cf40ef0>
```



4.3.2 Lineplot

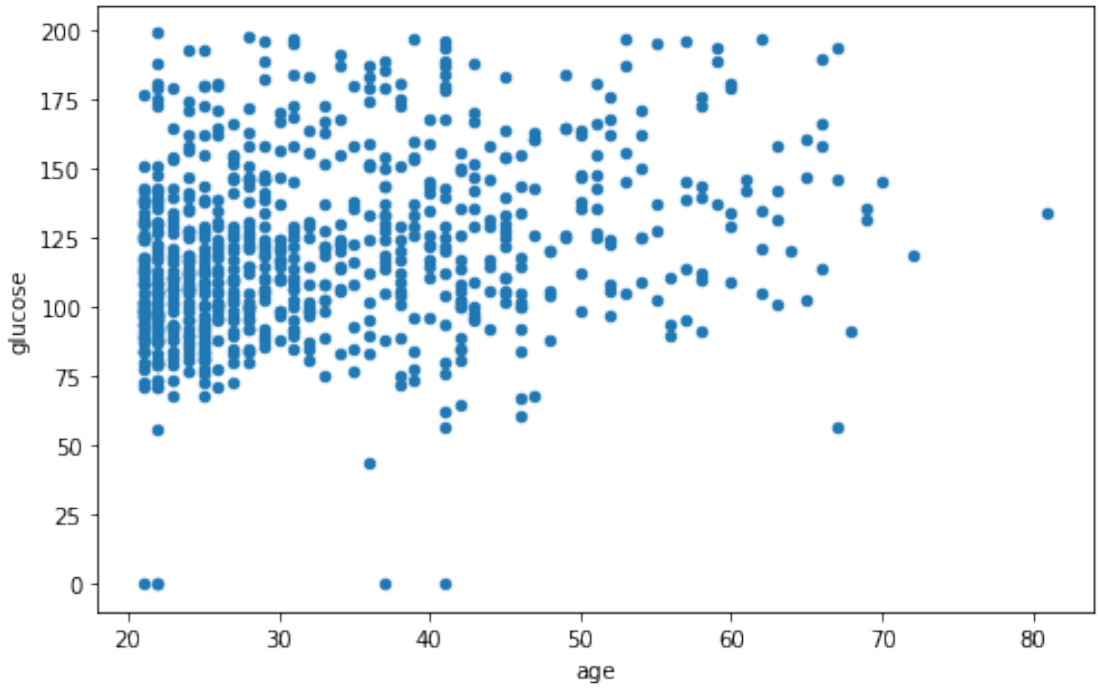
```
In [49]: # Plotting with index along the x-axis
         df['glucose'].plot(figsize=(12, 5), color='black') # color and figsize changed

         plt.xlim(0, 780) # range for x-axis
         plt.ylim(0, 200) # range for x-axis
         plt.xlabel('index')
         plt.ylabel('Glucose level'); # ";" prevents object info from displaying
```



4.3.3 Scatterplot

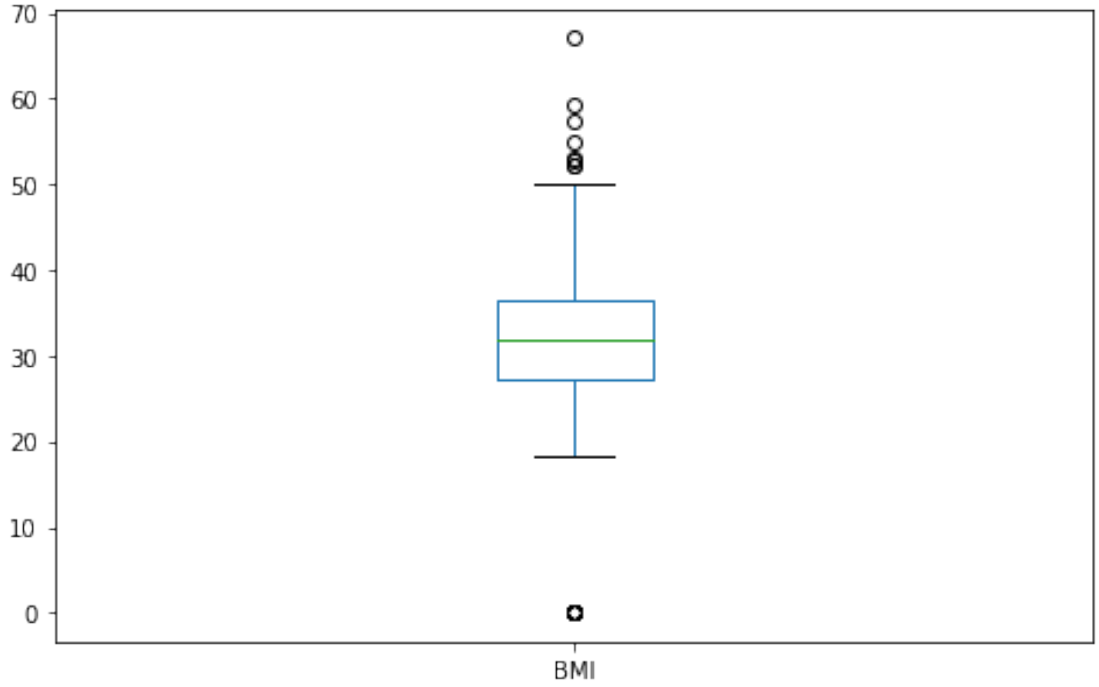
```
In [50]: # plotting one variable against the other
         df.plot.scatter('age', 'glucose', figsize=(8, 5))

         # The x and y labels are automatically taken from the column names
```

```
Out[50]: <matplotlib.axes._subplots.AxesSubplot at 0x2007d694320>
```



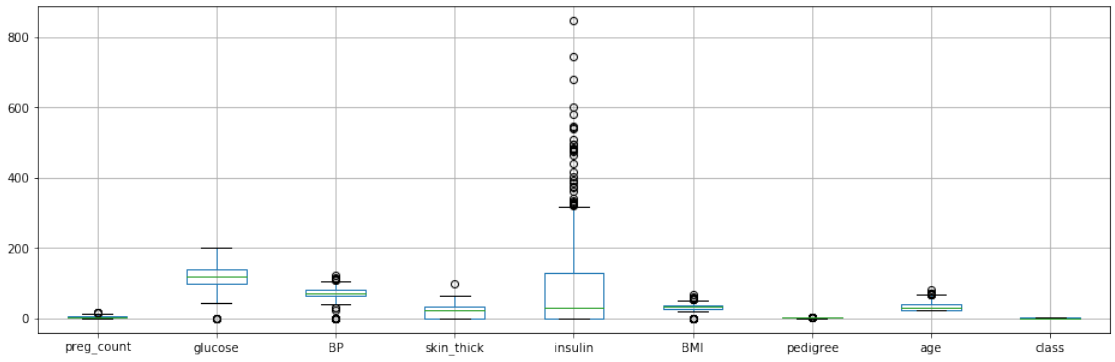
4.3.4 Boxplot

```
In [51]: # Box plot of a column
         df['BMI'].plot.box(figsize=(8, 5));
```



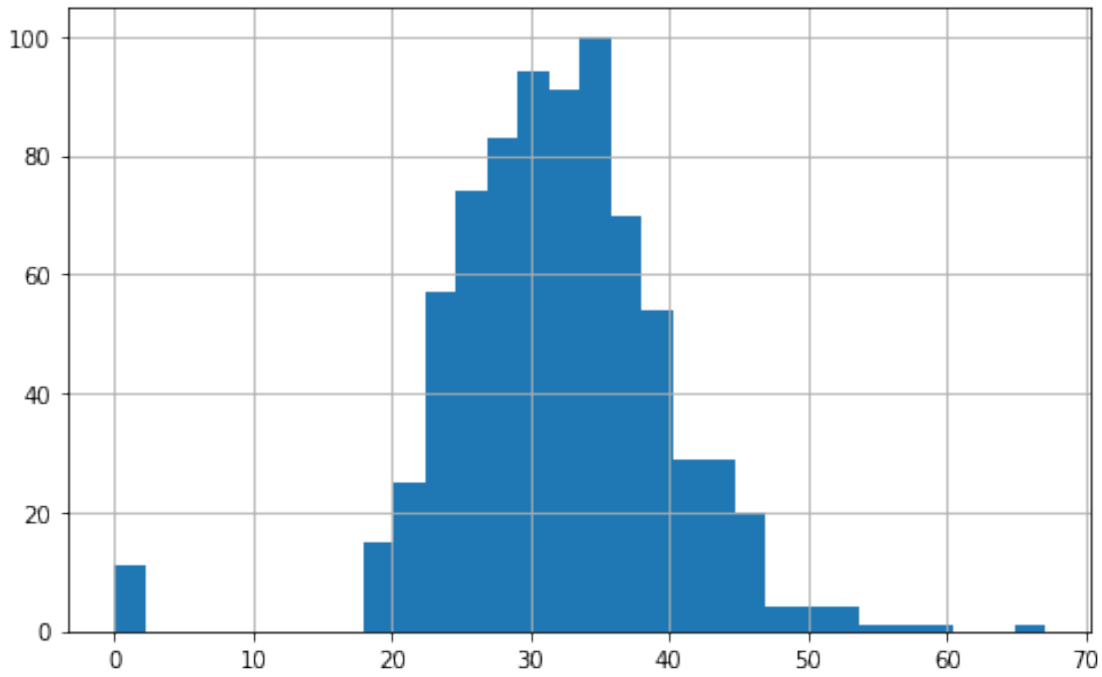
```
In [52]: # Box plot of all the columns with numerical data
         df.boxplot(figsize=(16, 5)) # or df.plot.box()
```

```
Out[52]: <matplotlib.axes._subplots.AxesSubplot at 0x2007d5d9898>
```

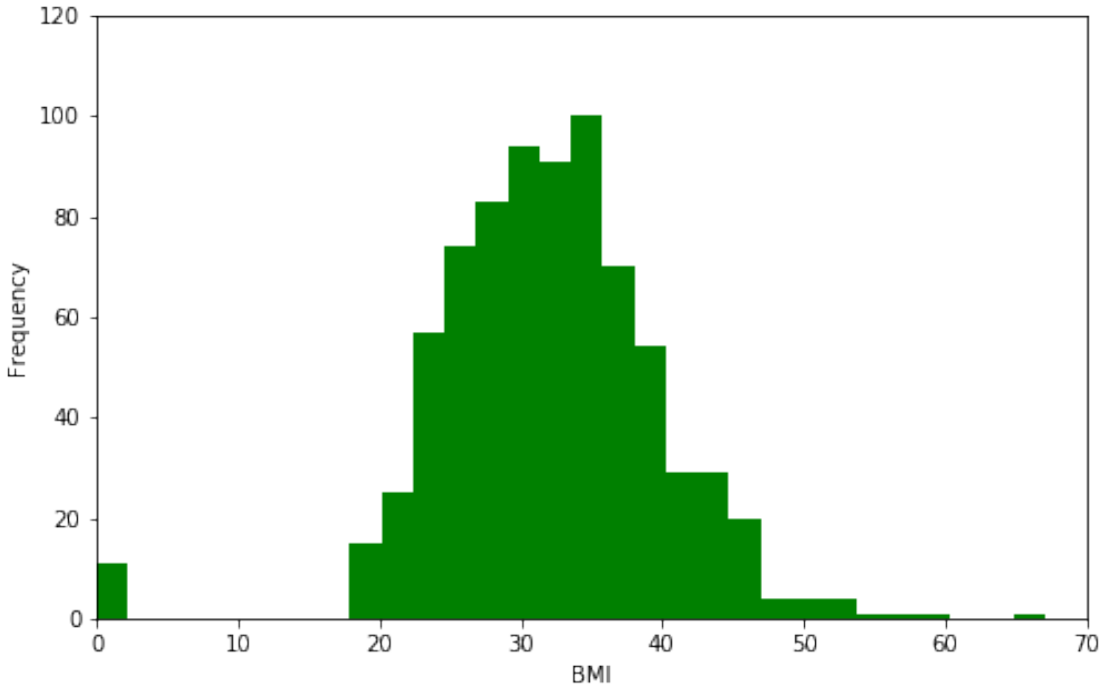


4.3.5 Histogram

```
In [53]: df['BMI'].hist(bins=30, figsize=(8, 5)); # we can specify the number of bins
```



```
In [54]: ax = df['BMI'].hist(bins=30, grid=False, color='green', figsize=(8, 5)) # grid turned
         off and color changed

         ax.set_xlabel('BMI')
         ax.set_ylabel('Frequency')

         ax.set_xlim(0, 70) # limiting display range to 0-70 for the x-axis
         ax.set_ylim(0, 120); # limiting display range to 0-120 for the y-axis
```



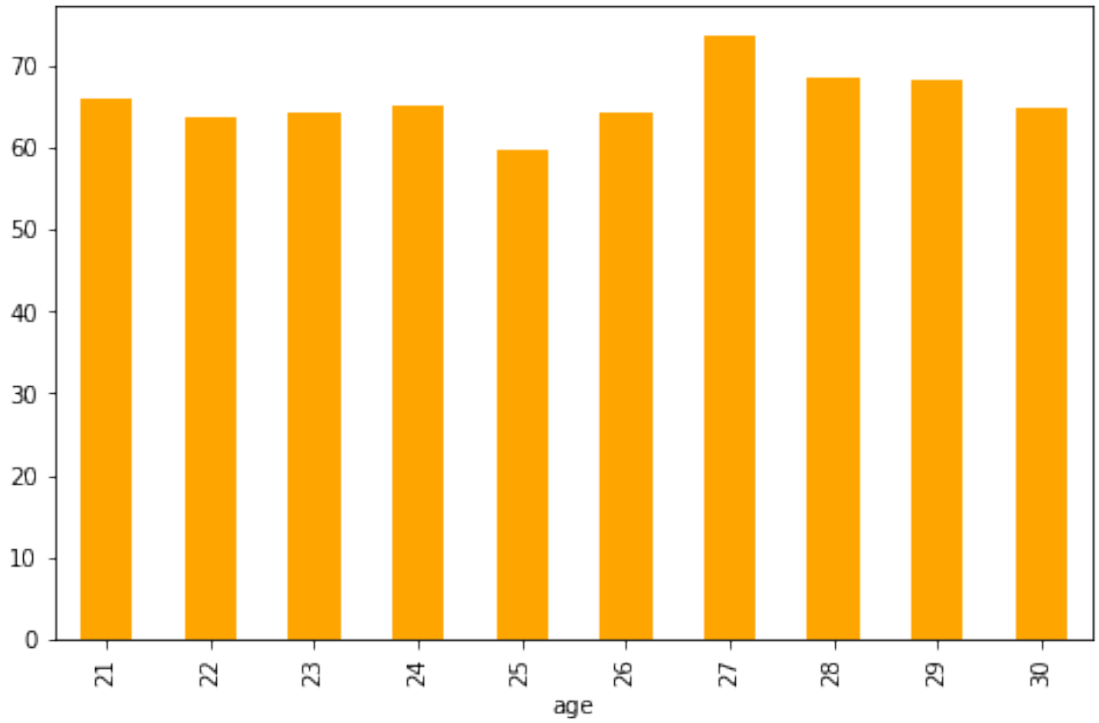
As we mostly use figsize = (8, 5) for most of the pictures, it will convenient to default figsize as (8, 5) and only modify the figsize when necessary. This way we can avoid the need for passing the figure dimensions explicitly everytime we plot.

```
In [55]: from matplotlib import pylab

In [56]: pylab.rcParams['figure.figsize'] = (8, 5)
```

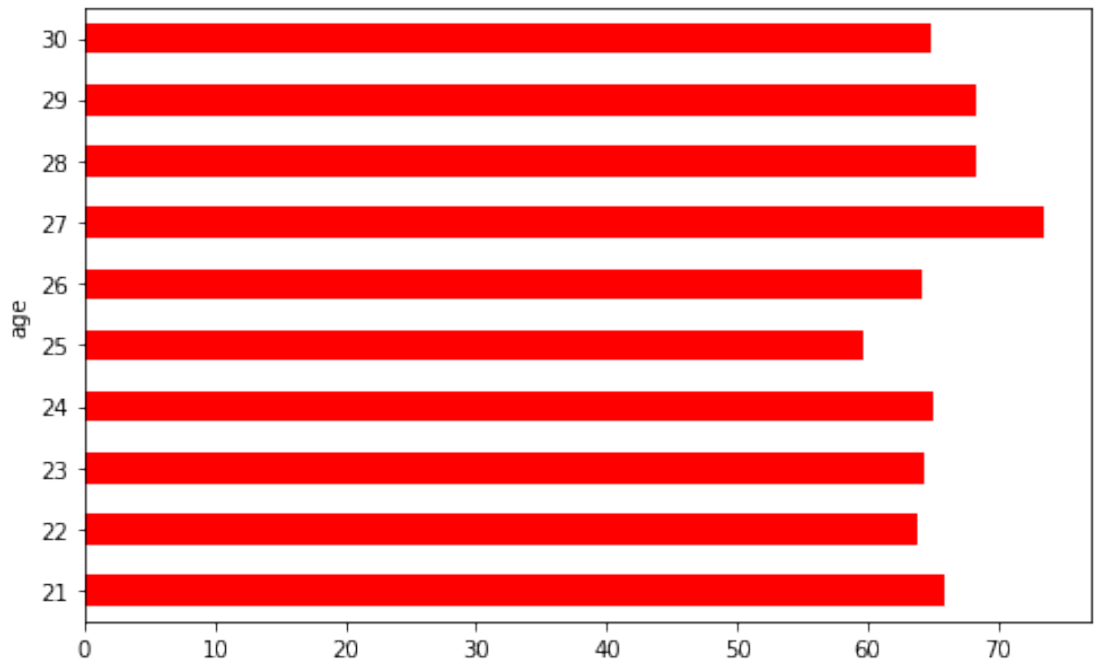
4.3.6 Barplot

The bar charts are used to visualize categorical data (nominal or ordinal values) and the height shows the value it represents

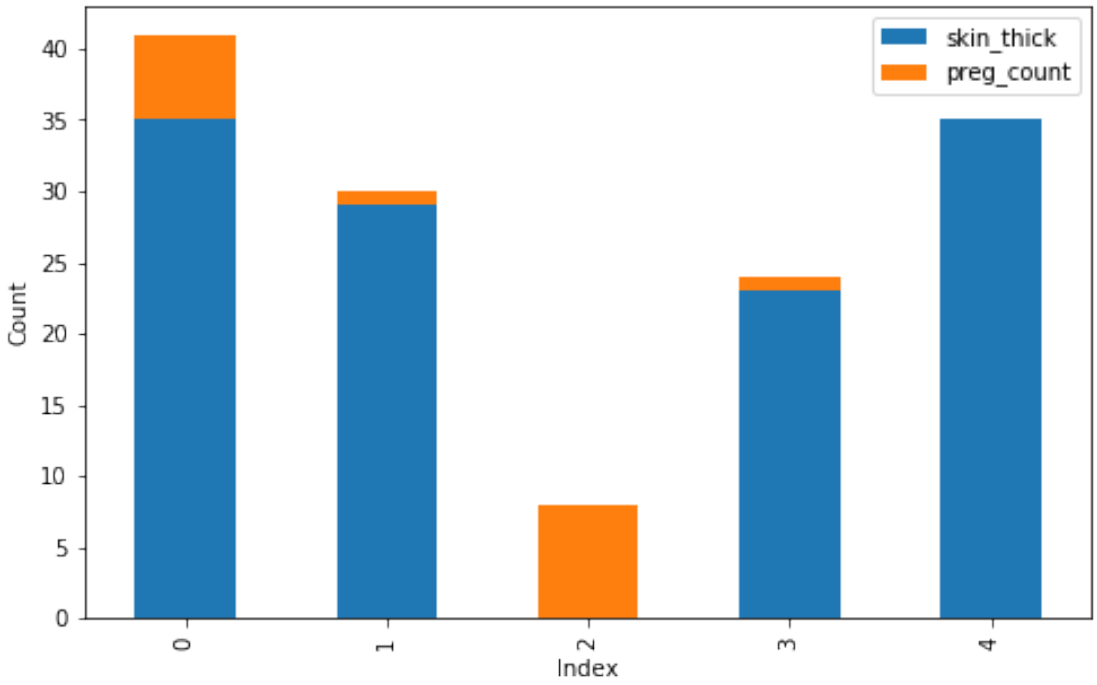
```
In [57]: df_avg_BP = df.groupby('age')['BP'].mean()
         df_avg_BP[:10].plot.bar(color='orange');
```




```
In [58]: # To plot horizontally
         df_avg_BP[:10].plot.barh(color='red');
```



```
In [59]: # To stack values from multiple columns
         ax = df[['skin_thick', 'preg_count']][:5].plot.bar(stacked=True)

         ax.set_xlabel("Index")
         ax.set_ylabel("Count");
```

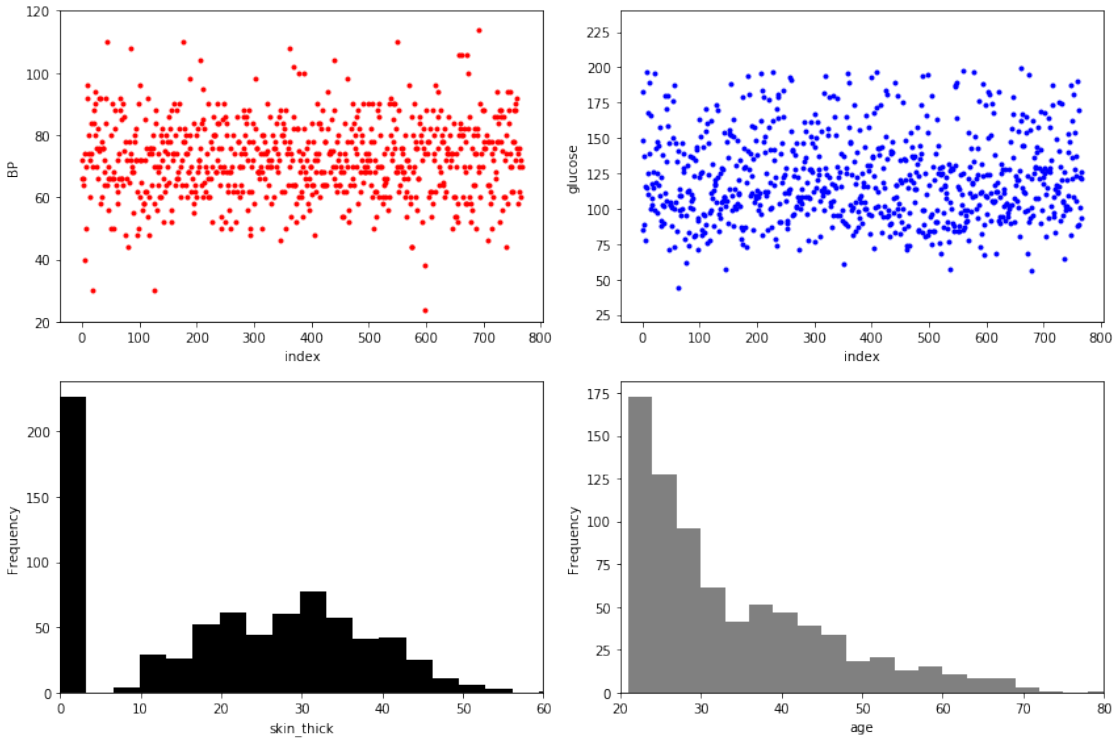


Although, the above plot doesn't make much sense, it serves the purpose as an example. Stacked bar plots are generally used to determine % contribution from each variable for every value in the categorical variable under consideration (index here).

4.3.7 Multiple Plots

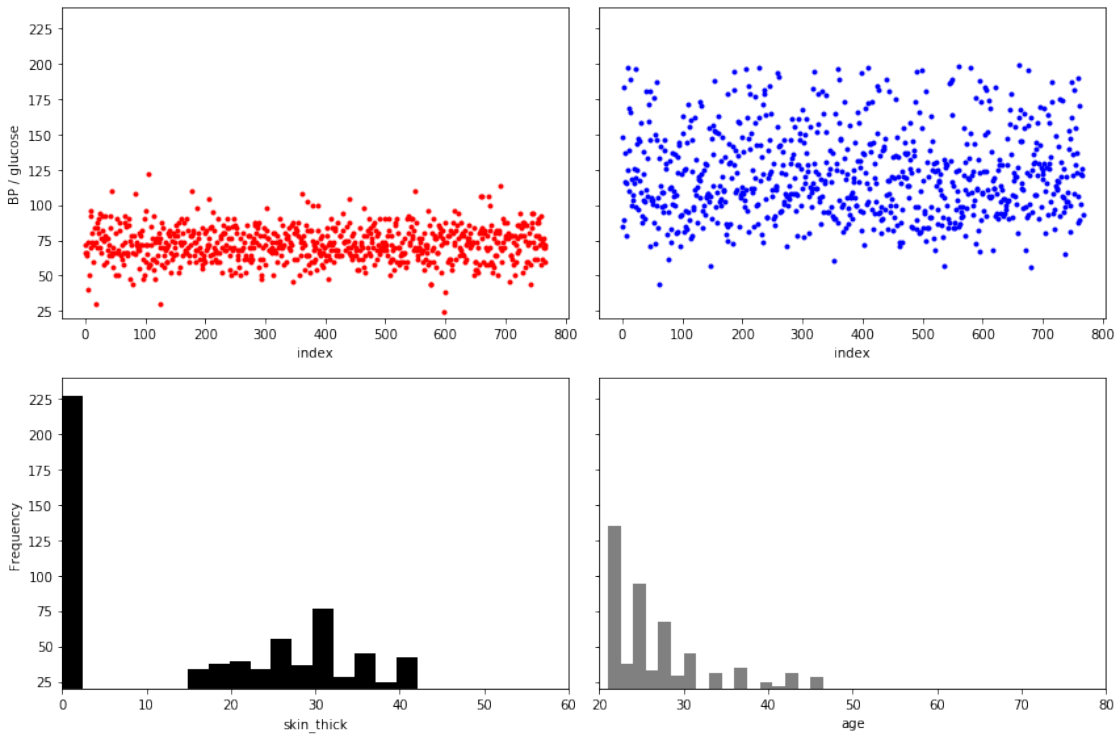
```
In [60]: import matplotlib.pyplot as plt

In [61]: fig, axes = plt.subplots(2, 2, figsize=(12, 8))
         # or fig, (ax1, ax2, ax3, ax4) = plt.subplots(2, 2, figsize=(12, 8))

         # axes is the axes object(s). It can be a single object or an array of objects.
         # In this case, it is an array of dimension 2-by-2

         df['BP'].plot(ax = axes[0][0], style='.', color='red') # top left
         df['glucose'].plot(ax = axes[0][1], style='.', color='blue') # top right
```

```
df['skin_thick'].plot.hist(bins=30, ax = axes[1][0], color='black') # bottom left
df['age'].plot.hist(bins=20, ax = axes[1][1], color='gray') # bottom right

axes[0][0].set_xlabel('index')
axes[0][1].set_xlabel('index')
axes[1][0].set_xlabel('skin_thick')
axes[1][1].set_xlabel('age')

axes[0][0].set_ylabel('BP')
axes[0][1].set_ylabel('glucose')

axes[0][0].set_ylim(20, 120)
axes[0][1].set_ylim(20, 240)

axes[1][0].set_xlim(0, 60)
axes[1][1].set_xlim(20, 80)

fig.tight_layout()
```




```
In [62]: # shared y axis
         fig, axes = plt.subplots(2, 2, figsize=(12, 8), sharey=True)
```

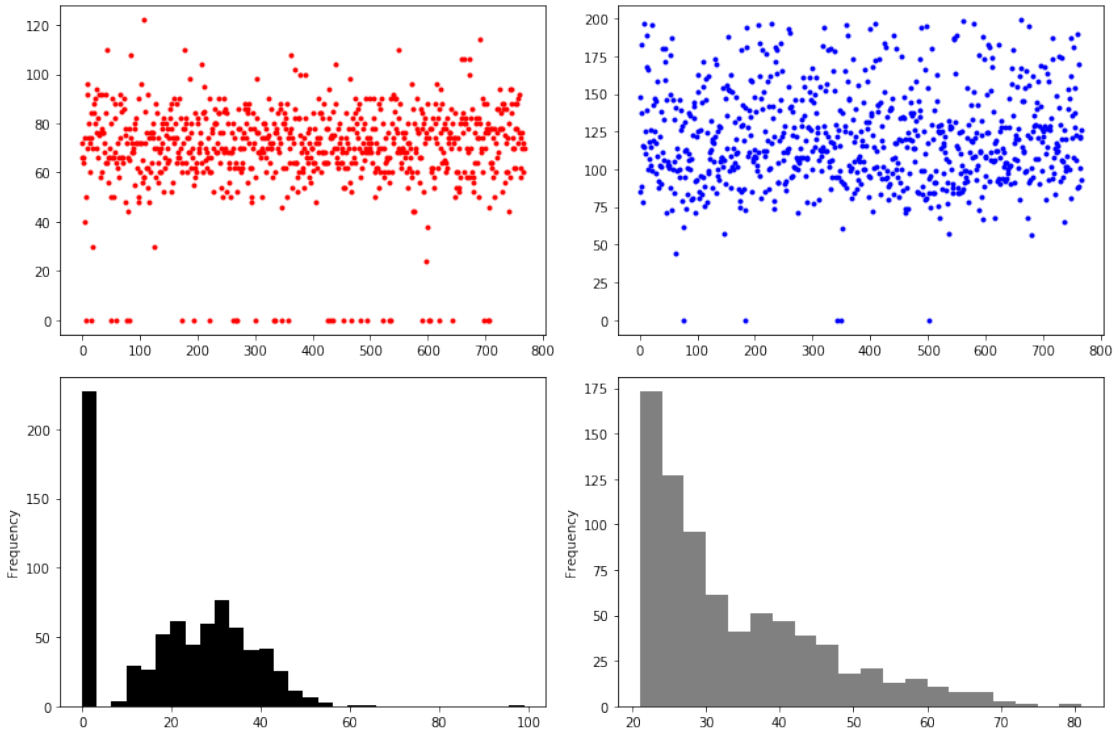
```
df['BP'].plot(ax = axes[0][0], style='.', color='red') # top left
df['glucose'].plot(ax = axes[0][1], style='.', color='blue') # top right

df['skin_thick'].plot.hist(bins=40, ax = axes[1][0], color='black') # bottom left
df['age'].plot.hist(bins=40, ax = axes[1][1], color='gray') # bottom right

axes[0][0].set_xlabel('index')
axes[0][1].set_xlabel('index')
axes[1][0].set_xlabel('skin_thick')
axes[1][1].set_xlabel('age')

axes[0][0].set_ylabel('BP / glucose')

axes[0][0].set_ylim(20, 120)
axes[0][1].set_ylim(20, 240)

axes[1][0].set_xlim(0, 60)
axes[1][1].set_xlim(20, 80)

fig.tight_layout()
```



Also, we can apply sharex=True if the x-axes of the plots have similar range.

4.3.8 Multiple Plots - Another method

```
In [63]: fig, ax = plt.subplots(2, 2, figsize=(12, 8))

         plt.subplot(2, 2, 1)
         df['BP'].plot(style='.', color='red')

         plt.subplot(2, 2, 2)
         df['glucose'].plot(style='.', color='blue') # bottom right

         plt.subplot(2, 2, 3)
         df['skin_thick'].plot.hist(bins=30, color='black') # top right

         plt.subplot(2, 2, 4)
         df['age'].plot.hist(bins=20, color='gray') # bottom left

         plt.tight_layout()
         plt.show()
```



These are the mainly used methods for visualizing data using Pandas and matplotlib and will be helpful to quicky go through the data exploration process. However, there are other modules such as seaborn and bokeh which are quite important as well but they applicable when data visualization is intended for presentation. There are more advanced techniques in matplotlib and other modules which are only required

when the plot mandates lot of customization to match your taste; those are out of the scope of this book, so please explore it on your own.

4.4 Data Cleansing

Unclean data renders only useless and inaccurate models. Garbage-in, garbage-out (GIGO)

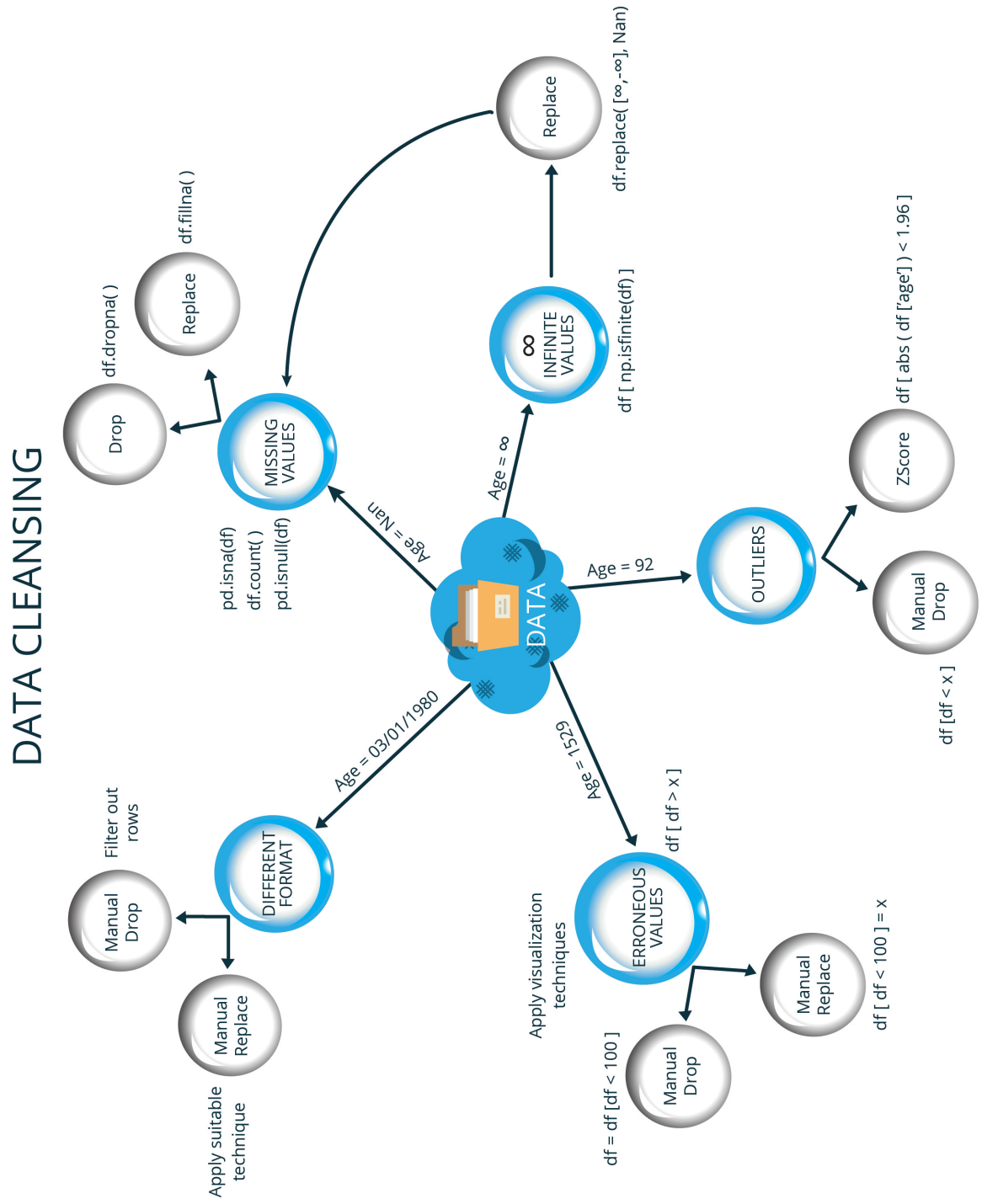
It is meaningless to spend any time in modeling, if your data is not clean. Data cleaning is the most important task in the entire predicive modeling work flow. Clean data is critical for training models to achieve good predictive power.

Data scientists usually spend 70% of their time in understanding and cleaning the data which shows the seriousness of the task among data scientists. On contrary to the common misconception that modeling is the most time consuming task, it just involves 30% of the work.

Data integrity is questionable when any of the following exists in the data

1. Missing values (NaNs),
2. Infinite values,
3. Outliers,
4. Erroneous values, and
5. Values in different format.

For each type, we need to apply suitable method(s) to clean the data. Let's discuss different methods and techniques to clean the data.



Data cleaning

4.4.1 Missing values (NaNs)

Identify missing values

```
In [64]: df.count()

Out[64]: preg_count    768
         glucose       768
         BP            768
         skin_thick    768
         insulin       768
         BMI           768
         pedigree      768
         age           768
         class         768
         dtype: int64

In [65]: len(df)

Out[65]: 768
```

The count on each column is equal to the length of the DataFrame. So there are no missing values.
For study, let's introduce some null values into the data and fill them with appropriate values.

```
In [66]: # Introducing NaNs in the dataset
         for cols in df.columns:
             # randomly pick 25 rows (without replacement) for each parameter and replace the
         values with NaNs
             index_ = np.random.choice(df.index, size=25, replace = 0)
             df.loc[index_, cols] = np.NaN

In [67]: df.index

Out[67]: RangeIndex(start=0, stop=768, step=1)

In [68]: df.count() / df.shape[0] # % of rows with non-null values

Out[68]: preg_count    0.967448
         glucose       0.967448
         BP            0.967448
         skin_thick    0.967448
         insulin       0.967448
         BMI           0.967448
         pedigree      0.967448
         age           0.967448
         class         0.967448
         dtype: float64
```

```
In [69]: df.shape[0] - df.count() # number of rows with non-null values
```

```
Out[69]: preg_count    25
         glucose       25
         BP            25
         skin_thick    25
         insulin       25
         BMI           25
         pedigree      25
         age           25
         class         25
         dtype: int64
```

Now that we have injected some missing values into our dataset we can go ahead and impute them with appropriate values or drop them altogether.

Other methods to find missing values

a. pd.isna()

```
In [70]: pd.isna(df).head() # return a boolean DataFrame with True for missing values and False
         for non-null values
```

```
Out[70]:    preg_count  glucose     BP  skin_thick  insulin    BMI  pedigree    age  \
         0       False    False  False       False    False  False     False  False
         1       False    False   True       False     True  False     False  False
         2       False    False  False       False    False  False     False  False
         3       False    False  False       False    False  False     False  False
         4       False    False  False       False    False  False     False  False

            class
         0  False
         1  False
         2  False
         3  False
         4  False
```

```
In [71]: pd.isna(df).sum()
```

```
Out[71]: preg_count    25
         glucose       25
         BP            25
         skin_thick    25
         insulin       25
         BMI           25
         pedigree      25
```

```
age           25
class         25
dtype: int64
```

b. pd.isnull()

```
In [72]: pd.isnull(df).head() # same as pd.isna()
```

```
Out[72]:    preg_count  glucose     BP  skin_thick  insulin    BMI  pedigree    age  \
         0       False    False  False       False    False  False     False  False
         1       False    False   True       False     True  False     False  False
         2       False    False  False       False    False  False     False  False
         3       False    False  False       False    False  False     False  False
         4       False    False  False       False    False  False     False  False

            class
         0  False
         1  False
         2  False
         3  False
         4  False
```

Just in case, if you want to reverse the condition and filter only non-null values

```
In [73]: pd.notnull(df).head()
```

```
Out[73]:    preg_count  glucose     BP  skin_thick  insulin   BMI  pedigree   age  \
         0        True     True   True        True     True  True      True  True
         1        True     True  False        True    False  True      True  True
         2        True     True   True        True     True  True      True  True
         3        True     True   True        True     True  True      True  True
         4        True     True   True        True     True  True      True  True

            class
         0   True
         1   True
         2   True
         3   True
         4   True
```

Handle missing values There are two methods to handle missing data, they are 1. drop the rows or 2. impute missing values

```
In [74]: df_null = df[df['BMI'].isnull()].copy(deep=True)
         df_null.head()
```

```
Out[74]:      preg_count  glucose    BP  skin_thick  insulin  BMI  pedigree   age  \
         5           5.0    116.0  74.0         0.0      0.0  NaN     0.201  30.0
         94          2.0    142.0  82.0        18.0     64.0  NaN     0.761  21.0
         107         4.0    144.0  58.0        28.0    140.0  NaN     0.287  37.0
         115         4.0    146.0  92.0         0.0      0.0  NaN     0.539  61.0
         162         0.0    114.0  80.0        34.0    285.0  NaN     0.167  27.0

              class
         5      0.0
         94     0.0
         107    0.0
         115    1.0
         162    0.0

In [75]: df_null.count()

Out[75]: preg_count    23
         glucose       25
         BP            25
         skin_thick    25
         insulin       25
         BMI            0
         pedigree      23
         age           23
         class         25
         dtype: int64
```

a. Drop missing values Drop the rows with missing values

- df1.dropna(how='any') - drop the rows with atleast one missing values
- df1.dropna(how='all') - drop the rows with all missing values
- df1.dropna(how='any', axis = 1) - drop the columns with atleast one missing values
- df1.dropna(how='all', axis = 1) - drop the columns with all missing values

```
In [76]: df_null.dropna(how='all').head()

Out[76]:      preg_count  glucose    BP  skin_thick  insulin  BMI  pedigree   age  \
         5           5.0    116.0  74.0         0.0      0.0  NaN     0.201  30.0
         94          2.0    142.0  82.0        18.0     64.0  NaN     0.761  21.0
         107         4.0    144.0  58.0        28.0    140.0  NaN     0.287  37.0
         115         4.0    146.0  92.0         0.0      0.0  NaN     0.539  61.0
         162         0.0    114.0  80.0        34.0    285.0  NaN     0.167  27.0

              class
         5      0.0
         94     0.0
```

```
107    0.0
115    1.0
162    0.0
```

```
In [77]: df_null.count()
```

```
Out[77]: preg_count    23
         glucose       25
         BP            25
         skin_thick    25
         insulin       25
         BMI            0
         pedigree      23
         age           23
         class         25
         dtype: int64
```

None of the rows were dropped (max count = 25) as there are no rows with all missing values

```
In [78]: df_null.dropna(how='any')
```

```
Out[78]: Empty DataFrame
         Columns: [preg_count, glucose, BP, skin_thick, insulin, BMI, pedigree, age, class]
         Index: []
```

All the rows were dropped as atleast one of the values in each row are missing

```
In [79]: df_null.dropna(how='all', axis = 1).head()  # axis = 1 refers to columns. By default
         axis = 0.
```

```
Out[79]:      preg_count  glucose    BP  skin_thick  insulin  pedigree   age  class
         5           5.0    116.0  74.0         0.0      0.0     0.201  30.0    0.0
         94          2.0    142.0  82.0        18.0     64.0     0.761  21.0    0.0
         107         4.0    144.0  58.0        28.0    140.0     0.287  37.0    0.0
         115         4.0    146.0  92.0         0.0      0.0     0.539  61.0    1.0
         162         0.0    114.0  80.0        34.0    285.0     0.167  27.0    0.0
```

'BMI' column is dropped as all the values are missing

```
In [80]: df_null.dropna(how='any', axis = 1).head()
```

```
Out[80]:      glucose    BP  skin_thick  insulin  class
         5      116.0  74.0         0.0      0.0    0.0
         94     142.0  82.0        18.0     64.0    0.0
         107    144.0  58.0        28.0    140.0    0.0
         115    146.0  92.0         0.0      0.0    1.0
         162    114.0  80.0        34.0    285.0    0.0
```

3 columns have atleast one missing value and hence they are dropped

Drop values from selected columns

```
In [81]: index_glucose = df[df["glucose"].isnull()].index
         index_glucose

Out[81]: Int64Index([  7,  10,  35,  47, 105, 142, 208, 229, 290, 331, 352, 378, 398,
                     416, 430, 432, 486, 491, 575, 587, 594, 650, 683, 723, 763],
                    dtype='int64')

In [82]: index_preg_count = df[df["preg_count"].isnull()].index
         index_preg_count

Out[82]: Int64Index([ 37, 295, 309, 314, 359, 366, 380, 383, 394, 404, 437, 468, 473,
                     500, 534, 544, 564, 659, 663, 664, 677, 696, 699, 722, 747],
                    dtype='int64')

In [83]: set(index_glucose) & set(index_preg_count)

Out[83]: set()
```

No rows with missing value for both "glucose" and "preg_count"

b. Replace missing values

```
In [84]: df_null.count()

Out[84]: preg_count    23
         glucose       25
         BP            25
         skin_thick    25
         insulin       25
         BMI            0
         pedigree      23
         age           23
         class         25
         dtype: int64
```

Replace with a constant value

```
In [85]: df_null.fillna(value=5).head() # replace NaNs with a constant value 5

Out[85]:      preg_count  glucose    BP  skin_thick  insulin  BMI  pedigree   age  \
         5           5.0    116.0  74.0         0.0      0.0  5.0     0.201  30.0
         94          2.0    142.0  82.0        18.0     64.0  5.0     0.761  21.0
         107         4.0    144.0  58.0        28.0    140.0  5.0     0.287  37.0
         115         4.0    146.0  92.0         0.0      0.0  5.0     0.539  61.0
         162         0.0    114.0  80.0        34.0    285.0  5.0     0.167  27.0
```

```
     class
5      0.0
94     0.0
107    0.0
115    1.0
162    0.0
```

Replace with a statistical measure - mean, mode, or median

```
In [86]: df_null['BMI'] = df_null['BMI'].fillna( df['BMI'].mean() ) # applying the changes to the
         row
         df_null['BMI'].head()

Out[86]: 5      32.004038
         94     32.004038
         107    32.004038
         115    32.004038
         162    32.004038
         Name: BMI, dtype: float64

In [87]: df_null['BP'] = df_null['BP'].fillna( df['BP'].mean() ) # applying the changes to the
         row
         df_null['BP'].head()

Out[87]: 5      74.0
         94     82.0
         107    58.0
         115    92.0
         162    80.0
         Name: BP, dtype: float64
```

Replace missing values with values from another column or a Pandas series

```
In [88]: df_null[df_null["pedigree"].isnull()]

Out[88]:      preg_count  glucose    BP  skin_thick  insulin        BMI  pedigree  \
         232         1.0     79.0  80.0        25.0     37.0  32.004038       NaN
         615         3.0    106.0  72.0         0.0      0.0  32.004038       NaN

               age  class
         232  22.0    0.0
         615  27.0    0.0

In [90]: df_null['pedigree'] = df_null['pedigree'].fillna( df['insulin'] ) # applying the changes
         to the row
         df_null.loc[(232, 615), :]
```

```
Out[90]:      preg_count  glucose    BP  skin_thick  insulin        BMI  pedigree  \
         232         1.0     79.0  80.0        25.0     37.0  32.004038      37.0
         615         3.0    106.0  72.0         0.0      0.0  32.004038       0.0

               age  class
         232  22.0    0.0
         615  27.0    0.0
```

Similarly, the missing values can be replaced with any desired value. Please remember that domain knowledge plays a major role here. The above operation of filling 'insulin' readings with 'pedigree' is merely for explaining the concept otherwise it is simply nonsensical.

4.4.2 Impute infinite values

The technique is to convert infinite values to NaNs and then apply any of the methods from the previous section to replace the missing values.

```
In [91]: # Lets create a dummy DataFrame
         dict_ = {'a': np.arange(0, 3, 0.5),
                  'b': np.arange(5, 8, 0.5),
                  'c': np.arange(4, 7, 0.5)}

         dff = pd.DataFrame(dict_)
         dff
```

```
Out[91]:      a    b    c
         0  0.0  5.0  4.0
         1  0.5  5.5  4.5
         2  1.0  6.0  5.0
         3  1.5  6.5  5.5
         4  2.0  7.0  6.0
         5  2.5  7.5  6.5
```

```
In [92]: # Lets add infinite values to the DataFrame
         dff = dff.replace([2, 5, 7, 6], -np.inf)
         dff
```

```
Out[92]:           a         b         c
         0  0.000000      -inf  4.000000
         1  0.500000  5.500000  4.500000
         2  1.000000      -inf      -inf
         3  1.500000  6.500000  5.500000
         4      -inf      -inf      -inf
         5  2.500000  7.500000  6.500000
```

Methods to convert infinite values to missing values

Method: 1

```
In [93]: dff[np.isfinite(dff)]
```

```
Out[93]:      a    b    c
         0  0.0  NaN  4.0
         1  0.5  5.5  4.5
         2  1.0  NaN  NaN
         3  1.5  6.5  5.5
         4  NaN  NaN  NaN
         5  2.5  7.5  6.5
```

Method: 2

```
In [94]: dff = dff.replace([np.inf, -np.inf], np.nan)
         dff
```

```
Out[94]:      a    b    c
         0  0.0  NaN  4.0
         1  0.5  5.5  4.5
         2  1.0  NaN  NaN
         3  1.5  6.5  5.5
         4  NaN  NaN  NaN
         5  2.5  7.5  6.5
```

4.4.3 Outliers

Let's create a DataFrame with 'names' and 'age' for practice.

```
In [95]: dict_ = { "Names": ['John', 'Paul', "Mark", "Sarah", "Morgan", "Mike", "Ram",
                             "Jennifer", "Amy", "Chris", "Mitchel", "Alex"],
                   "Age": [25, 45, 12, 95, 6, 21, 34, 24, 41, 31, 19, 16]
                  }

         dff = pd.DataFrame(dict_)
```

```
In [96]: dff
```

```
Out[96]:     Age     Names
         0    25      John
         1    45      Paul
         2    12      Mark
         3    95     Sarah
         4     6    Morgan
         5    21      Mike
         6    34       Ram
         7    24  Jennifer
```

```
8    41       Amy
9    31     Chris
10   19   Mitchel
11   16      Alex
```

In the above dataset eveyones' age is between 5-45 except Sarah. She is the oldest one with significant difference with the next oldest person (Paul). These eccentric values are called outliers and it is recommended to remove/adjust them to not skew the statistical measures such as mean, standard deviation, etc., which are used by machine learning models during training.

First we assume that the data is normally distributed to compute z-score for each data point and then eliminate the ones that are outside threshold.

```
In [97]: from scipy import stats

In [98]: stats.zscore(dff['Age'])

Out[98]: array([-0.25843784,  0.64047639, -0.84273209,  2.88776198, -1.11240636,
                -0.43822069,  0.14607356, -0.30338355,  0.46069354,  0.01123643,
                -0.52811211, -0.66294925])
```

For two-sided test the z-scores for the thresholds are: - 95-percentile: 1.96 - 99-percentile: 2.58

So the z-score for Sarah's age is higher than 99-percentile limit. Hence it can be removed.

```
In [99]: # Lets use 95 percentile value
         dff = dff[abs(stats.zscore(dff['Age'])) < 1.96]

         # after removing rows the corresponding indices will be missing so it is recommended to
         reset index
         dff = dff.reset_index(drop=True)
         dff

Out[99]:     Age     Names
         0    25      John
         1    45      Paul
         2    12      Mark
         3     6    Morgan
         4    21      Mike
         5    34       Ram
         6    24  Jennifer
         7    41       Amy
         8    31     Chris
         9    19   Mitchel
         10   16      Alex
```

What if you want to apply this to all the columns?

dff = dff[(abs(stats.zscore(x))<1.96).all(axis=1)]

x - is the place holder for the columns/rows all - to apply the condition for all the columns/rows in the DataFrame axis = 1 - for columns

4.4.4 Erroneous values

The erroneous values are different from outliers in a sense that erroneous values are improbable values whereas outliers are probably but extremely unlikely.

```
In [100]: dict_ = { "Names": ['John', 'Paul', "Mark", "Sarah", "Morgan", "Mike", "Ram",
                              "Jennifer", "Amy", "Chris", "Mitchel", "Alex"],
                    "Age": [25, 45, 12, 167, 6, 21, 34, 24, 410, 31, 19, 16]
                   }

          dff = pd.DataFrame(dict_)
          dff
```

```
Out[100]:     Age     Names
          0    25      John
          1    45      Paul
          2    12      Mark
          3   167     Sarah
          4     6    Morgan
          5    21      Mike
          6    34       Ram
          7    24  Jennifer
          8   410       Amy
          9    31     Chris
          10   19   Mitchel
          11   16      Alex
```

What are improbable/incorrect values from the above DataFrame?

Yes, the age of Sarah and Amy. It could never be 167 and 410. No one ever lives that long. So it is clearly incorrect data. There are many methods to clean such erroneous data.

1. Treat them as outliers and apply the previous method
2. Manually find them, replace them with NaNs, and then fill the NaNs with a statistical measure (mean, mode, etc.)
3. Find them all and drop them

```
In [101]: dff[dff['Age'] < 100] # drops the rows with Age >= 100
```

```
Out[101]:     Age     Names
          0    25      John
          1    45      Paul
          2    12      Mark
          4     6    Morgan
          5    21      Mike
          6    34       Ram
          7    24  Jennifer
          9    31     Chris
```

```
          10   19   Mitchel
          11   16      Alex

In [102]: dff.loc[dff['Age']>100, "Age"] = np.NaN # replace Age >=100 with a NaN
          dff

Out[102]:      Age     Names
          0   25.0      John
          1   45.0      Paul
          2   12.0      Mark
          3    NaN     Sarah
          4    6.0    Morgan
          5   21.0      Mike
          6   34.0       Ram
          7   24.0  Jennifer
          8    NaN       Amy
          9   31.0     Chris
          10  19.0   Mitchel
          11  16.0      Alex
```

Now we can apply one of the methods we know for replacing missing values.

4.4.5 Values in different format

In data sets you receive, you may find data that are enterted in different format than expected. For example, instead age you might find date of birth. The one advantage over other formats is that the required values can be extracted from them rather than dropping them all.

```
In [103]: dict_ = { "Names": ['John', 'Paul', "Mark", "Sarah", "Morgan", "Mike", "Ram",
                              "Jennifer", "Amy", "Chris", "Mitchel", "Alex"],
                    "Age": [25, 45, 12, 0.1, 6, 21, "03/20/1980", 24, 410, 31, 19, 16]
                   }

          dff = pd.DataFrame(dict_)
          dff

Out[103]:            Age     Names
          0           25      John
          1           45      Paul
          2           12      Mark
          3          0.1     Sarah
          4            6    Morgan
          5           21      Mike
          6   03/20/1980       Ram
          7           24  Jennifer
          8          410       Amy
          9           31     Chris
```

```
          10          19   Mitchel
          11          16      Alex
```

Look at the age of Sarah and Ram. The Sarah's age is incorrect and in the place of Ram's age his date of bith (DOB) was recorded. Clearly, both are incorrect however, we can do nothing other than dropping Sarah's information whereas Ram's age can be computed from his DOB.

The first indication of having values in different format is having an unexpected typesetting for the columns. For example, below we have 'object' as the dtype for 'Age'.

```
In [104]: dff.dtypes

Out[104]: Age      object
          Names    object
          dtype: object
```

The next step is to directly force the typesetting as following. If it succeeds without any traceback, we are good to proceed to the next step and if it fails, then we need to handle it accordingly.

```
In [105]: # Method: 1
          try:
              dff['Age'].astype(int)
          except:
              traceback.print_exc()

Traceback (most recent call last):
  File "<ipython-input-105-b91e3f797455>", line 3, in <module>
...
  File "Pandas/_libs/src/util.pxd", line 91, in util.set_value_at_unsafe
ValueError: invalid literal for int() with base 10: '03/20/1980'


In [106]: # Method: 2
          all(isinstance(x, int) for x in dff['Age'])

Out[106]: False
```

Filter out the rows that do not match the expected format

```
In [218]: # Method: 1
          index_ = [not isinstance(x, int) for x in dff['Age']]
          dff[index_]

Out[218]:           Age  Names
          3         0.1  Sarah
          6  03/20/1980    Ram
```

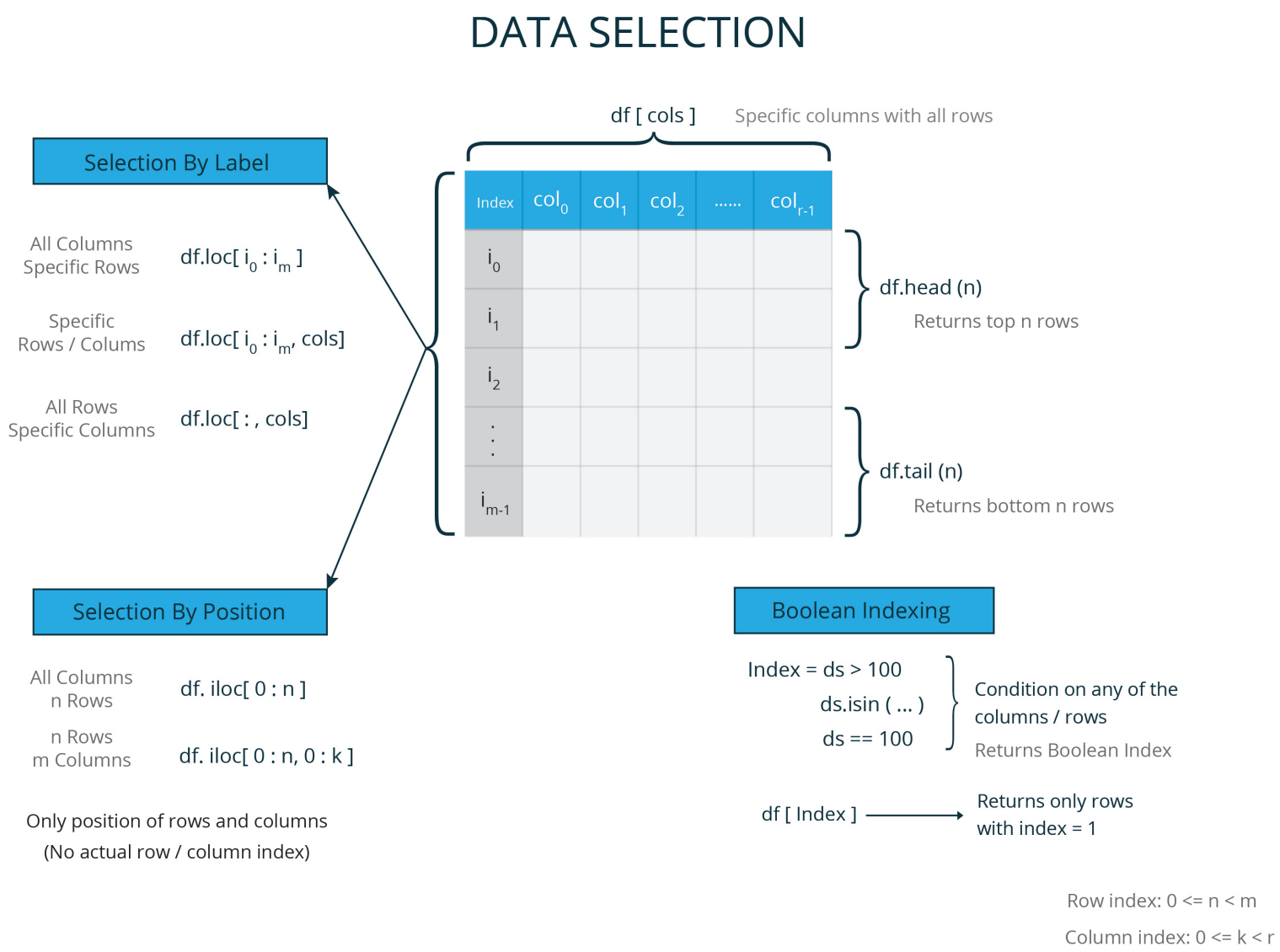
```
In [219]: # Method: 2
          index_ = [type(x)!=int for x in dff['Age']]
          dff[index_]

Out[219]:          Age  Names
          3        0.1  Sarah
          6  03/20/1980    Ram
```

Now apply suitable method of your choice to convert them to appropriate values or drop them.

4.5 Data Selection

Data selection techniques are helpful for choosing specific columns or rows for

a. Viewing - sneak preview of the data,
b. Apply a function to only certain columns or rows, and
c. Filter out unwanted rows or columns for the analysis.



Data selection

```
In [107]: # Head
          df.head(3) # displays first n rows (5 is the default)
```

Out[107]:

	preg_count	glucose	BP	skin_thick	insulin	BMI	pedigree	age	class
0	6.0	148.0	72.0	35.0	0.0	33.6	0.627	50.0	1.0
1	1.0	85.0	NaN	29.0	NaN	26.6	0.351	31.0	0.0
2	8.0	183.0	64.0	0.0	0.0	23.3	0.672	32.0	1.0

```
In [108]: # Tail
          df.tail(3) # displays last n rows (5 is the default)
```

Out[108]:

	preg_count	glucose	BP	skin_thick	insulin	BMI	pedigree	age	\
765	5.0	121.0	72.0	23.0	112.0	NaN	0.245	30.0	
766	1.0	126.0	60.0	0.0	0.0	30.1	0.349	47.0	
767	1.0	93.0	70.0	31.0	0.0	30.4	0.315	23.0	

	class
765	0.0
766	1.0
767	0.0

```
In [109]: # By columns
          df['BMI'][:5]
```

```
Out[109]: 0    33.6
          1    26.6
          2    23.3
          3    28.1
          4    43.1
          Name: BMI, dtype: float64
```

```
In [110]: # By slicing - displays first 5 rows
          df[:5]
```

Out[110]:

	preg_count	glucose	BP	skin_thick	insulin	BMI	pedigree	age	class
0	6.0	148.0	72.0	35.0	0.0	33.6	0.627	50.0	1.0
1	1.0	85.0	NaN	29.0	NaN	26.6	0.351	31.0	0.0
2	8.0	183.0	64.0	0.0	0.0	23.3	0.672	32.0	1.0
3	1.0	89.0	66.0	23.0	94.0	28.1	0.167	21.0	0.0
4	0.0	137.0	40.0	35.0	168.0	43.1	2.288	33.0	1.0

```
In [111]: # By slicing - displays last 5 rows
          df[-5:]
```

Out[111]:

	preg_count	glucose	BP	skin_thick	insulin	BMI	pedigree	age	\
763	10.0	NaN	76.0	48.0	180.0	32.9	0.171	NaN	
764	2.0	122.0	70.0	27.0	0.0	36.8	0.340	27.0	

```
765         5.0    121.0  72.0         23.0    112.0   NaN     0.245  30.0
766         1.0    126.0  60.0          0.0      0.0  30.1     0.349  47.0
767         1.0     93.0  70.0         31.0      0.0  30.4     0.315  23.0

     class
763    0.0
764    0.0
765    0.0
766    1.0
767    0.0
```

4.5.1 Selection by Label

```
In [225]: df.loc[0] # returns row with 0 as index

Out[225]: preg_count      6.000
          glucose       148.000
          BP             72.000
          skin_thick     35.000
          insulin         0.000
          BMI            33.600
          pedigree        0.627
          age            50.000
          class           1.000
          Name: 0, dtype: float64

In [226]: df.loc[0, 'BMI'] # returns 'BMI' with 0 as index

Out[226]: 33.600000000000001
```

The value we get is not exactly 33.6 but rather a minisculely higher value by one trillionth. This is due to floating point error which is common and you should be aware of that.

```
In [227]: print(df.loc[0, 'BMI'].astype(np.float16)) # 16 bits
          print(df.loc[0, 'BMI'].astype(np.float32)) # 32 bits
          print(df.loc[0, 'BMI'].astype(np.float64)) # 64 bits

33.594
33.6
33.6


In [228]: print(sys.getsizeof(df.loc[0, 'BMI'].astype(np.float16))) # 16 bits
          print(sys.getsizeof(df.loc[0, 'BMI'].astype(np.float32))) # 32 bits
          print(sys.getsizeof(df.loc[0, 'BMI'].astype(np.float64))) # 64 bits

26
28
```

```
In [229]: df.loc[0, {'BMI', 'glucose'}]

Out[229]: glucose    148.0
          BMI         33.6
          Name: 0, dtype: float64

In [230]: df.loc[0:5, {'BMI', 'glucose'}]

Out[230]:    glucose   BMI
          0    148.0  33.6
          1     85.0  26.6
          2    183.0  23.3
          3     89.0  28.1
          4    137.0  43.1
          5    116.0  25.6
```

4.5.2 Selection by position

```
In [231]: df.iloc[10] # returns 11th row (it may not be the row with index 10)

Out[231]: preg_count      4.000
          glucose       110.000
          BP             92.000
          skin_thick      0.000
          insulin         0.000
          BMI            37.600
          pedigree        0.191
          age            30.000
          class           0.000
          Name: 10, dtype: float64

In [232]: df.iloc[3:5,:]

Out[232]:    preg_count  glucose    BP  skin_thick  insulin   BMI  pedigree   age  class
          3         1.0     89.0  66.0         NaN     94.0  28.1     0.167  21.0    0.0
          4         0.0    137.0  40.0        35.0    168.0  43.1     2.288  33.0    1.0

In [233]: df.iloc[3:5, 0:2] # cant use columns names here like {'BMI', 'glucose'}

Out[233]:    preg_count  glucose
          3         1.0     89.0
          4         0.0    137.0
```

4.5.3 Data filtering

```
In [234]: # returns only rows with preg_count > 10
          df[df['preg_count'] > 10].head()
```

Out[234]:

	preg_count	glucose	BP	skin_thick	insulin	BMI	pedigree	age	class
24	11.0	143.0	94.0	33.0	146.0	36.6	0.254	51.0	1.0
28	13.0	145.0	82.0	19.0	110.0	22.2	0.245	NaN	0.0
36	11.0	138.0	76.0	0.0	0.0	33.2	0.420	35.0	0.0
72	13.0	126.0	90.0	0.0	0.0	43.4	0.583	42.0	1.0
86	13.0	106.0	72.0	54.0	0.0	36.6	0.178	45.0	0.0

```
In [235]: # returns only rows with class = 1
          df[df['class'] == 1].head()
```

Out[235]:

	preg_count	glucose	BP	skin_thick	insulin	BMI	pedigree	age	class
0	6.0	148.0	72.0	35.0	0.0	33.6	0.627	50.0	1.0
2	8.0	183.0	64.0	0.0	0.0	23.3	0.672	32.0	1.0
4	0.0	137.0	40.0	35.0	168.0	43.1	2.288	33.0	1.0
6	3.0	78.0	50.0	32.0	88.0	31.0	0.248	26.0	1.0
8	2.0	197.0	70.0	45.0	543.0	30.5	0.158	53.0	1.0

```
In [236]: # We can even select specific values to be segregated
          df[df['preg_count'].isin([2, 5])].head()
```

Out[236]:

	preg_count	glucose	BP	skin_thick	insulin	BMI	pedigree	age	class
5	5.0	116.0	74.0	0.0	0.0	25.6	0.201	30.0	0.0
8	2.0	197.0	70.0	45.0	543.0	30.5	0.158	53.0	1.0
14	5.0	166.0	72.0	19.0	175.0	25.8	0.587	NaN	1.0
29	5.0	117.0	92.0	0.0	0.0	34.1	0.337	38.0	0.0
30	5.0	109.0	75.0	26.0	0.0	36.0	0.546	60.0	0.0

```
In [237]: # Negation
          df[~df['preg_count'].isin([2, 5])].head()
```

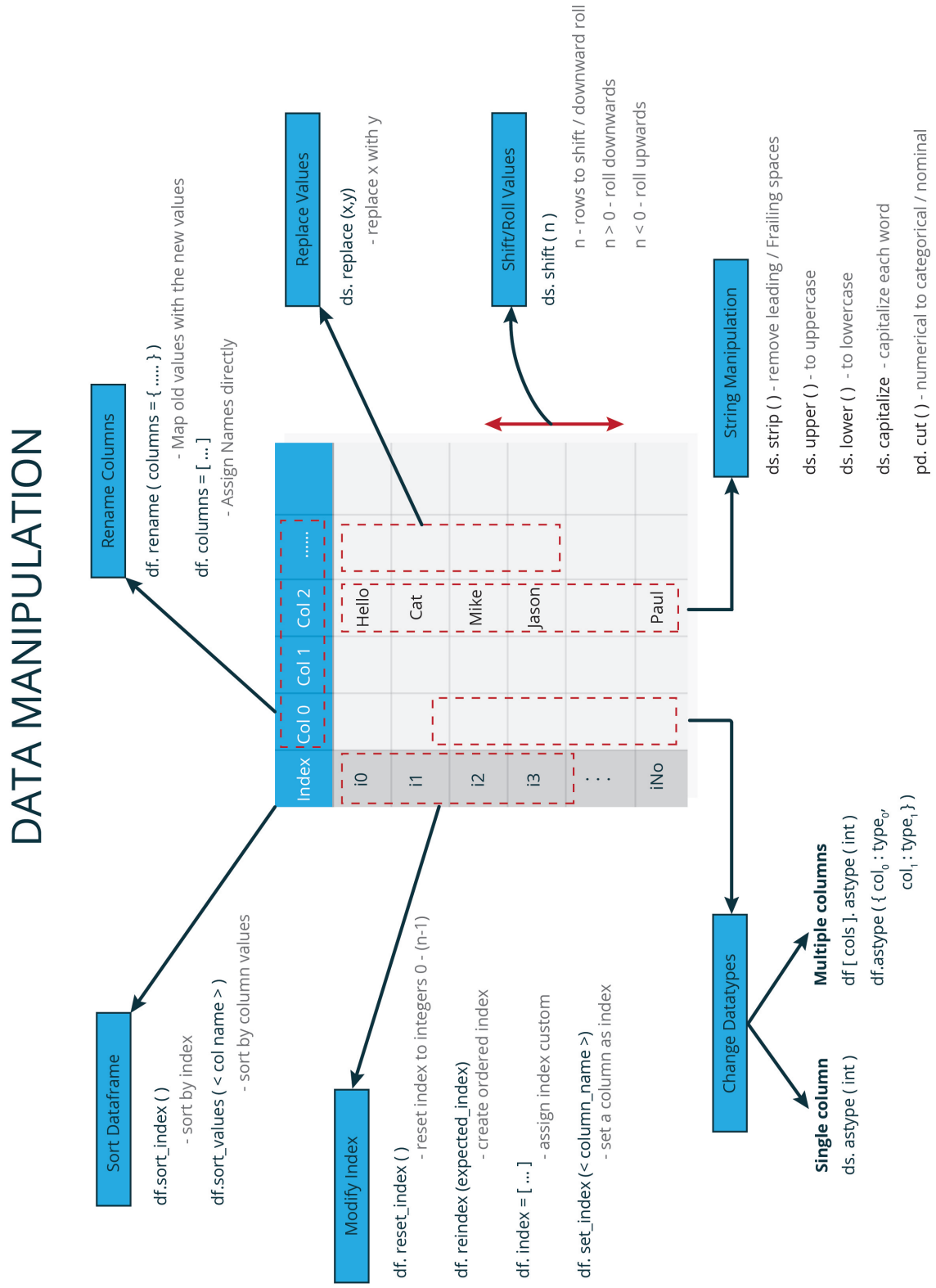
Out[237]:

	preg_count	glucose	BP	skin_thick	insulin	BMI	pedigree	age	class
0	6.0	148.0	72.0	35.0	0.0	33.6	0.627	50.0	1.0
1	1.0	85.0	66.0	29.0	0.0	26.6	0.351	31.0	0.0
2	8.0	183.0	64.0	0.0	0.0	23.3	0.672	32.0	1.0
3	1.0	89.0	66.0	NaN	94.0	28.1	0.167	21.0	0.0
4	0.0	137.0	40.0	35.0	168.0	43.1	2.288	33.0	1.0

The tilt sign preceding the condition represents negation. Therefore rows with preg_count not equal to either 2 or 5 is returned

4.6 Data Manipulation

Data manipulation techniques allows us to modify the data such that it is suitable for the analysis.



Data manipulation

4.6.1 String Manipulation

- upper() / lower() / capitalize() / strip()

```
In [239]: dict_ = { "Names": ['John', 'Paul ', "Mark", "SaRah", "Morgan ", "Mike",
                              " Ram", " JenNifer", "Amy", "ChrIs", "Mitchel", "Alex"],
                    "Age": [25, 45, 12, 167, 6, 21, 34, 24, 410, 31, 19, 16]
                   }

          dff = pd.DataFrame(dict_)
          dff
```

Out[239]:

	Age	Names
0	25	John
1	45	Paul
2	12	Mark
3	167	SaRah
4	6	Morgan
5	21	Mike
6	34	Ram
7	24	JenNifer
8	410	Amy
9	31	ChrIs
10	19	Mitchel
11	16	Alex

Please note that there are leading and trailing spaces inserted into the names and also the names have few upper case letters in it, these will make the word-search task challenging. So it is highly recommended to remove the leading/trailing spaces and fix the inconsistency in alphabet casing.

Strip

```
In [240]: # To remove the leading and trailing spaces
          dff["Names"] = dff["Names"].str.strip()
          print(dff["Names"].values)
```

```
['John' 'Paul' 'Mark' 'SaRah' 'Morgan' 'Mike' 'Ram' 'JenNifer' 'Amy'
 'ChrIs' 'Mitchel' 'Alex']
```

Upper

```
In [241]: # Convert all the characterst to upper case:
          dff["Names"].str.upper()
```

```
Out[241]: 0         JOHN
          1         PAUL
```

```
2         MARK
3        SARAH
4       MORGAN
5         MIKE
6          RAM
7     JENNIFER
8          AMY
9        CHRIS
10     MITCHEL
11        ALEX
Name: Names, dtype: object
```

Lower

```
In [242]: # Convert all the characterst to upper case:
          dff["Names"] = dff["Names"].str.lower()
          dff["Names"]
```

```
Out[242]: 0         john
          1         paul
          2         mark
          3        sarah
          4       morgan
          5         mike
          6          ram
          7     jennifer
          8          amy
          9        chris
          10     mitchel
          11        alex
          Name: Names, dtype: object
```

Capitalize

```
In [243]: # Capitalize each word
          dff["Names"] = [x.capitalize() for x in dff["Names"]]
          dff["Names"]
```

```
Out[243]: 0         John
          1         Paul
          2         Mark
          3        Sarah
          4       Morgan
          5         Mike
          6          Ram
          7     Jennifer
```

```
8          Amy
9        Chris
10     Mitchel
11        Alex
Name: Names, dtype: object
```

Binning

- Binning is used to convert numerical values into categorical/nominal values

```
In [325]: df_cut = pd.cut(df['BMI'], bins=np.arange(0, 75, step=5), right=False) #without label
          for the bins

          df_cut_label = pd.cut(df['BMI'], bins=np.arange(0, 75, step=5), labels=range(14),
          right=False) # With labels for the bins

In [326]: df_cut = pd.DataFrame(df_cut)
          df_cut_label = pd.DataFrame(df_cut_label)

In [327]: display("df_cut.head()", "df_cut_label.head()")

Out[327]: df_cut.head()
                  BMI
          0  [30, 35)
          1  [25, 30)
          2  [20, 25)
          3  [25, 30)
          4  [40, 45)

          df_cut_label.head()
            BMI
          0   6
          1   5
          2   4
          3   5
          4   8
```

Labels: $0 < 1 < 2 < \ldots < 13$

4.6.2 To modify the index

Let's say we drop the rows with age > 100, then the index will be out of sequence. In such cases, it is suggested to reindex the series or DataFrame again such that it is in sequence again. In some cases, we shouldn't reindex the dataset if the index carries important information such as patient ID or product ID.

```
In [244]: dff = dff[dff['Age']<100]
          dff
```

```
Out[244]:     Age     Names
          0    25      John
          1    45      Paul
          2    12      Mark
          4     6    Morgan
          5    21      Mike
          6    34       Ram
          7    24  Jennifer
          9    31     Chris
          10   19   Mitchel
          11   16      Alex
```

Rows 3, 8 are dropped. There are few techniques to clean this up.

Reset index

```
In [245]: # Create an ordered index - by dropping the current index
          dff.reset_index(drop=True) # reorder index from 0 - n-1

          # Make sure you are drop=True otherwise the previous indices will be added to the
          DataFrame as a new column
```

```
Out[245]:    Age     Names
          0   25      John
          1   45      Paul
          2   12      Mark
          3    6    Morgan
          4   21      Mike
          5   34       Ram
          6   24  Jennifer
          7   31     Chris
          8   19   Mitchel
          9   16      Alex
```

Reindex

```
In [246]: # Create an ordered index - by retaining current index and add missing indices (with NaN
          as values by default)
          index_ = np.arange(0, 12, step=1)
          dff.reindex(index=index_)
```

```
Out[246]:     Age     Names
          0   25.0      John
          1   45.0      Paul
          2   12.0      Mark
          3    NaN       NaN
```

```
4    6.0    Morgan
5   21.0      Mike
6   34.0       Ram
7   24.0  Jennifer
8    NaN       NaN
9   31.0     Chris
10  19.0   Mitchel
11  16.0      Alex
```

The missing 3rd, 8th row are added to the DataFrame with NaNs

```
In [248]: dff.reindex(index=index_, fill_value=0) # with a specific fill value for NaNs
```

```
Out[248]:     Age     Names
          0    25      John
          1    45      Paul
          2    12      Mark
          3     0         0
          4     6    Morgan
          5    21      Mike
          6    34       Ram
          7    24  Jennifer
          8     0         0
          9    31     Chris
          10   19   Mitchel
          11   16      Alex
```

Index Insert a custom index by replacing the current index

```
In [249]: new_index = ["P"+str(i) for i in range(len(dff))]
          new_index
```

```
Out[249]: ['P0', 'P1', 'P2', 'P3', 'P4', 'P5', 'P6', 'P7', 'P8', 'P9']
```

```
In [250]: dff.index = new_index
          dff
```

```
Out[250]:     Age     Names
          P0   25      John
          P1   45      Paul
          P2   12      Mark
          P3    6    Morgan
          P4   21      Mike
          P5   34       Ram
          P6   24  Jennifer
          P7   31     Chris
          P8   19   Mitchel
          P9   16      Alex
```

Set index To set a column from the DataFrame as the index

```
In [251]: dff.set_index('Age').head(3)

Out[251]:     Names
          Age
          25   John
          45   Paul
          12   Mark

In [252]: dff.set_index('Names').head(3)

Out[252]:        Age
          Names
          John    25
          Paul    45
          Mark    12
```

4.6.3 Sort DataFrames

Sort on index

```
In [253]: dff.sort_index(ascending=False) # set ascending=True if you want in ascending order

Out[253]:     Age     Names
          P9   16      Alex
          P8   19   Mitchel
          P7   31     Chris
          P6   24  Jennifer
          P5   34       Ram
          P4   21      Mike
          P3    6    Morgan
          P2   12      Mark
          P1   45      Paul
          P0   25      John
```

Sort by values

```
In [254]: dff.sort_values(by='Age', ascending=True)

Out[254]:     Age     Names
          P3    6    Morgan
          P2   12      Mark
          P9   16      Alex
          P8   19   Mitchel
          P4   21      Mike
          P6   24  Jennifer
```

```
P0   25      John
P7   31     Chris
P5   34       Ram
P1   45      Paul
```

4.6.4 Change Datatypes

For a single column

```
In [255]: # Type setting columns
          dff['Age'].astype(str)

Out[255]: P0    25
          P1    45
          P2    12
          P3     6
          P4    21
          P5    34
          P6    24
          P7    31
          P8    19
          P9    16
          Name: Age, dtype: object
```

Now the values in the 'Age' column are in string format (note that dtype=object for the column)

For multiple columns

```
In [256]: dff = dff.astype({"Age": "category", "Names": str})
          dff.head(3)

Out[256]:    Age Names
          P0  25  John
          P1  45  Paul
          P2  12  Mark

In [257]: dff.info()

<class 'Pandas.core.frame.DataFrame'>
Index: 10 entries, P0 to P9
Data columns (total 2 columns):
Age      10 non-null category
Names    10 non-null object
dtypes: category(1), object(1)
memory usage: 890.0+ bytes
```

Note that the datatype of "Age" column has changed to 'category' type

4.6.5 Shifting values

- to shift values by 'n' rows up or down

```
In [258]: # shift values
          dff['Age'].shift(1) # Shifts one row downwards

Out[258]: P0     NaN
          P1    25.0
          P2    45.0
          P3    12.0
          P4     6.0
          P5    21.0
          P6    34.0
          P7    24.0
          P8    31.0
          P9    19.0
          Name: Age, dtype: category
          Categories (10, int64): [6, 12, 16, 19, ..., 25, 31, 34, 45]

In [259]: # shift values
          dff['Age'].shift(-1) # Shifts one row upwards

Out[259]: P0    45.0
          P1    12.0
          P2     6.0
          P3    21.0
          P4    34.0
          P5    24.0
          P6    31.0
          P7    19.0
          P8    16.0
          P9     NaN
          Name: Age, dtype: category
          Categories (10, int64): [6, 12, 16, 19, ..., 25, 31, 34, 45]
```

4.6.6 Replace values

```
In [260]: dff['Age'] = dff['Age'].astype(int) # converting back to int datatype
          dff['Age'].head()

Out[260]: P0    25
          P1    45
          P2    12
          P3     6
          P4    21
          Name: Age, dtype: int64
```

```
In [261]: # replace values
          dff['Age'].replace(25, 45) # - replaces 25 with 45

Out[261]: P0    45
          P1    45
          P2    12
          P3     6
          P4    21
          P5    34
          P6    24
          P7    31
          P8    19
          P9    16
          Name: Age, dtype: int64
```

The values can also be replaced based on a condition using functions which will be discussed in the following sections

4.6.7 Rename columns index

```
In [262]: # Method-1: Rename column names with a dictionary
          dff.rename(columns = {"Names": "Des noms"})

Out[262]:     Age  Des noms
          P0   25       John
          P1   45       Paul
          P2   12       Mark
          P3    6     Morgan
          P4   21       Mike
          P5   34        Ram
          P6   24   Jennifer
          P7   31      Chris
          P8   19    Mitchel
          P9   16       Alex

In [112]: # Method-2: Rename column names with a list (length should be equal to the no. of
          columns)
          dff.columns = ["Age", "Names"]
          dff.head(3)

Out[112]:   Age Names
          0  25  John
          1  45  Paul
          2  12  Mark
```

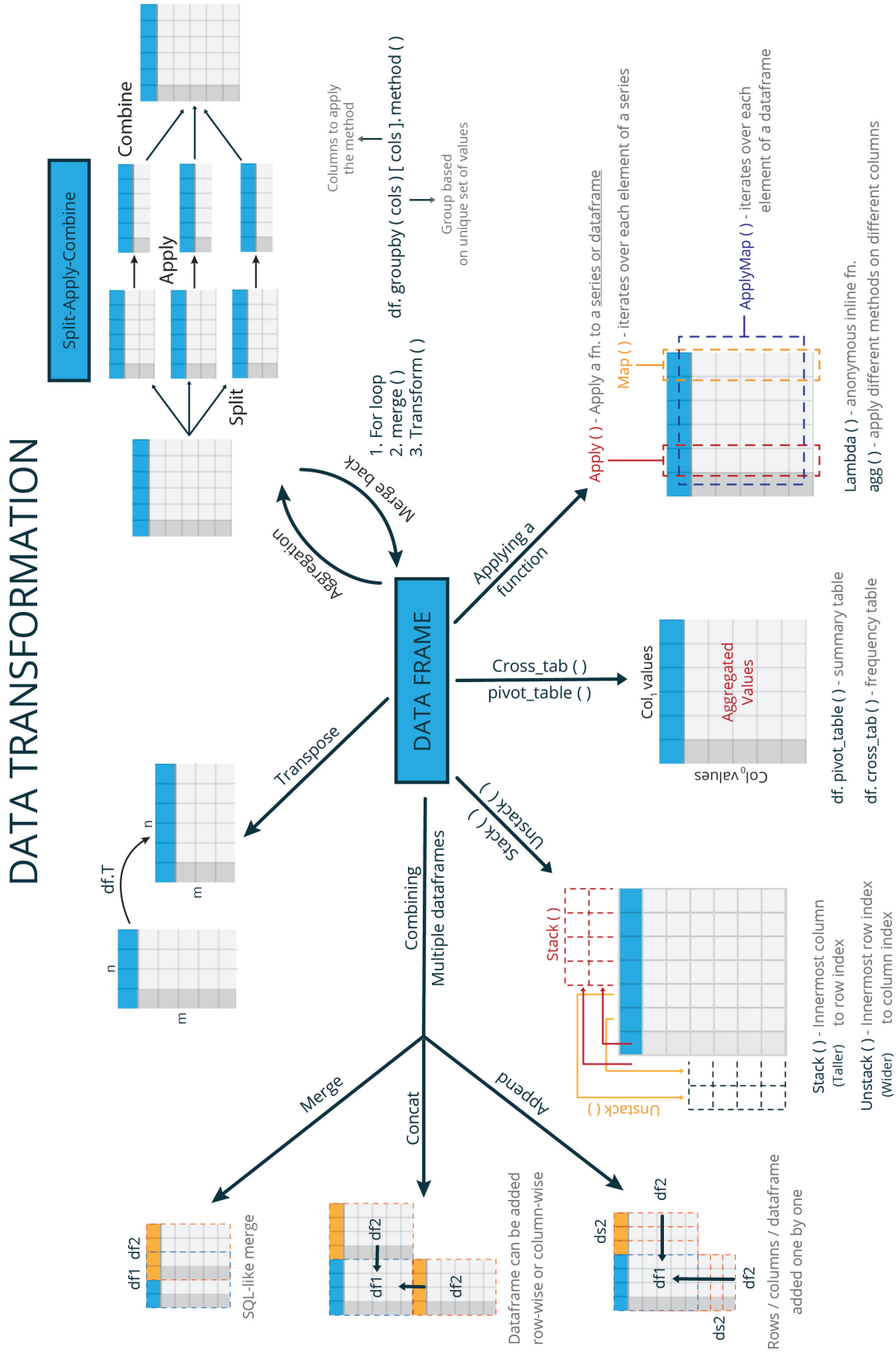
Although Method-2 is comparitively easier to implement, it is not recommended because if the columns are not in the expected order then the columns will be named inappropriately. In this case, there is chance of naming column containing age as Names and vice versa. So the Method-1 is safer to use.

4.7 Data Transformation

For data science projects, we need to do more than just modifying the values in the table. We should have the ability to aggregate values and get results for each group, apply a custom function to a feature, rotate the tables, merge two or more tables, and much more.

Various techniques for data transformation are listed below.There are only 6 types of transformations that are most commonly used to change the data to intended format.

1. Split-Apply-Combine (aggregation by grouping)
2. Applying a function to one/more feature(s)
3. Pivot tables (aggregation by values)
4. Stacking and unstacking
5. Combining 2 or more DataFrames
6. Transpose of a DataFrame



Data transformation

For this section, let's use the pima-indians diabetes dataset.

4.7.1 Split-Apply-Combine

This is one of the most useful techniques in Pandas for data transformation.

- Split - aggregate/group the data based on unique values from the column(s) specified
- Apply - apply the intended method
- Combine - combine the results back into a DataFrame/series

```
In [265]: df.head()
```

Out[265]:

	preg_count	glucose	BP	skin_thick	insulin	BMI	pedigree	age	class
0	6.0	148.0	72.0	35.0	0.0	33.6	0.627	50.0	1.0
1	1.0	85.0	66.0	29.0	0.0	26.6	0.351	31.0	0.0
2	8.0	183.0	64.0	0.0	0.0	23.3	0.672	32.0	1.0
3	1.0	89.0	66.0	NaN	94.0	28.1	0.167	21.0	0.0
4	0.0	137.0	40.0	35.0	168.0	43.1	2.288	33.0	1.0

```
In [266]: df.groupby('preg_count')['glucose'].mean()
```

```
Out[266]: preg_count
          0.0     122.838095
          1.0     114.669421
          2.0     111.659794
          3.0     123.957746
          4.0     127.426230
          5.0     118.859649
          6.0     120.617021
          7.0     135.093023
          8.0     136.558824
          9.0     131.769231
          10.0    122.260870
          11.0    125.500000
          12.0    113.555556
          13.0    125.500000
          14.0    137.500000
          15.0    136.000000
          17.0    163.000000
          Name: glucose, dtype: float64
```

The 'glucose' values are aggregated based on the values in 'preg_count' column and mean() function is applied on the values for each group. The resulting DataFrame has unique value of 'glucose' as index and the mean for each group as values.

To merge with the parent DataFrame

Method: 1

```
In [271]: mean_glucose = df.groupby('preg_count')['glucose'].mean()

In [272]: df_ = df.copy() # let's make a copy to avoid overwriting the actual data

          t0 = time.time() # gets current time
          for i in mean_glucose.index:
              df_.loc[df_['preg_count'] == i, 'mean_glucose'] = mean_glucose[i]

          print("Time taken: {:.4f}s".format(time.time() - t0))
          df_.head()

Time taken: 0.0416s


Out[272]:    preg_count  glucose    BP  skin_thick  insulin   BMI  pedigree   age  \
          0         6.0    148.0  72.0        35.0      0.0  33.6     0.627  50.0
          1         1.0     85.0  66.0        29.0      0.0  26.6     0.351  31.0
          2         8.0    183.0  64.0         0.0      0.0  23.3     0.672  32.0
          3         1.0     89.0  66.0         NaN     94.0  28.1     0.167  21.0
          4         0.0    137.0  40.0        35.0    168.0  43.1     2.288  33.0

             class  mean_glucose
          0    1.0    120.617021
          1    0.0    114.669421
          2    1.0    136.558824
          3    0.0    114.669421
          4    1.0    122.838095
```

Using 'for' loop is one of the least efficient techniques. It scales as O(n) for each level.

Method: 2

- Using merge()

```
In [273]: # To merge the results with the current DataFrame
          t0 = time.time()
          temp = mean_glucose.rename('mean_glucose').reset_index()

          # identifies the column that matches with the index name and uses that as the primary
          key for merging
          temp = df.merge(temp) # by default uses "inner" join

          print("Time taken: {:.4f}s".format(time.time() - t0))
          temp.head()
```

```
Time taken: 0.0045s
```

```
Out[273]:    preg_count  glucose    BP  skin_thick  insulin   BMI  pedigree   age  \
          0         6.0    148.0  72.0        35.0      0.0  33.6     0.627  50.0
          1         6.0     92.0  92.0         0.0      0.0  19.9     0.188  28.0
          2         6.0    144.0  72.0        27.0    228.0  33.9     0.255  40.0
          3         6.0     93.0  50.0        30.0     64.0  28.7     0.356  23.0
          4         6.0    111.0  64.0        39.0      0.0  34.2     0.260  24.0

             class  mean_glucose
          0    1.0    120.617021
          1    0.0    120.617021
          2    0.0    120.617021
          3    0.0    120.617021
          4    0.0    120.617021
```

Much faster than the previous method, approximately 10 times faster.

Method: 3

- Transform() - applies the intended method and returns a like-indexed Pandas series with transformed values

```
In [274]: t0 = time.time()
          df_["mean_glucose"] = df.groupby('preg_count')['glucose'].transform('mean')

          print("Time taken: {:.4f}s".format(time.time() - t0))
          df_.head()
```

```
Time taken: 0.0022s
```

```
Out[274]:    preg_count  glucose    BP  skin_thick  insulin   BMI  pedigree   age  \
          0         6.0    148.0  72.0        35.0      0.0  33.6     0.627  50.0
          1         1.0     85.0  66.0        29.0      0.0  26.6     0.351  31.0
          2         8.0    183.0  64.0         0.0      0.0  23.3     0.672  32.0
          3         1.0     89.0  66.0         NaN     94.0  28.1     0.167  21.0
          4         0.0    137.0  40.0        35.0    168.0  43.1     2.288  33.0

             class  mean_glucose
          0    1.0    120.617021
          1    0.0    114.669421
          2    1.0    136.558824
          3    0.0    114.669421
          4    1.0    122.838095
```

Fastest so far! Twice as fast as the previous method.

4.7.2 Apply a function

```
In [275]: # let's create two simple functions for the study
          def div_100(x):
              return x/100

          def sum_100(x):
              return sum(x)/100
```

Apply

- Applies the selected method to the columns or rows in a DataFrame

```
In [278]: # Applying to a specific column
          df['glucose'].apply(div_100).head()

Out[278]: 0    1.48
          1    0.85
          2    1.83
          3    0.89
          4    1.37
          Name: glucose, dtype: float64

In [279]: # applying to all the columns in the DataFrame (axis=0)
          df.apply(sum_100, axis=0)

Out[279]: preg_count   NaN
          glucose      NaN
          BP           NaN
          skin_thick   NaN
          insulin      NaN
          BMI          NaN
          pedigree     NaN
          age          NaN
          class        NaN
          dtype: float64
```

Why are we getting NaNs? The DataFrame contains NaNs in each column resulting in NaNs for the sum. To overcome this, we can drop the NaNs, just for computing the sum, and apply the method as follows.

```
In [280]: df.dropna().apply(sum_100, axis=0)

Out[280]: preg_count     22.57000
          glucose       691.01000
          BP            397.64000
          skin_thick    115.46000
          insulin       449.28000
```

```
BMI           183.60400
pedigree        2.69097
age           191.24000
class           1.99000
dtype: float64
```

```
In [281]: # Applying to all the rows in the DataFrame (axis=1)
          df.dropna().apply(sum_100, axis=1).head()
```

```
Out[281]: 0    3.46227
          1    2.38951
          2    3.11972
          4    4.59388
          5    2.50801
          dtype: float64
```

Map

- Iterates over each element of a series (method for Pandas series)

```
In [282]: # The function converts the input to a string and postfixes with "A"
          def str_A(x):
              return str(x) + "A"
```

```
In [283]: df['glucose'].map(str_A).head()
```

```
Out[283]: 0    148.0A
          1     85.0A
          2    183.0A
          3     89.0A
          4    137.0A
          Name: glucose, dtype: object
```

Applymap

- The function is applied to each element of a DataFrame

```
In [285]: df.applymap(str_A).head()
```

```
Out[285]:   preg_count glucose     BP skin_thick insulin    BMI              pedigree  \
          0       6.0A  148.0A  72.0A      35.0A    0.0A  33.6A                 0.627A
          1       1.0A   85.0A  66.0A      29.0A    0.0A  26.6A  0.35100000000000003A
          2       8.0A  183.0A  64.0A       0.0A    0.0A  23.3A                 0.672A
          3       1.0A   89.0A  66.0A       nanA   94.0A  28.1A  0.16699999999999998A
          4       0.0A  137.0A  40.0A      35.0A  168.0A  43.1A   2.2880000000000003A
```

```
     age class
0  50.0A  1.0A
1  31.0A  0.0A
2  32.0A  1.0A
3  21.0A  0.0A
4  33.0A  1.0A
```

lambda: When the function is simple we can easily apply the method using lambda technique without creating a function separately. It's an inline method for applying a function.

```
In [286]: df.applymap(lambda x: str(x) + "A").head()
```

```
Out[286]:   preg_count glucose     BP skin_thick insulin    BMI              pedigree  \
          0       6.0A  148.0A  72.0A      35.0A    0.0A  33.6A                 0.627A
          1       1.0A   85.0A  66.0A      29.0A    0.0A  26.6A  0.35100000000000003A
          2       8.0A  183.0A  64.0A       0.0A    0.0A  23.3A                 0.672A
          3       1.0A   89.0A  66.0A       nanA   94.0A  28.1A  0.16699999999999998A
          4       0.0A  137.0A  40.0A      35.0A  168.0A  43.1A   2.2880000000000003A

               age class
          0  50.0A  1.0A
          1  31.0A  0.0A
          2  32.0A  1.0A
          3  21.0A  0.0A
          4  33.0A  1.0A
```

agg:

- This is method is useful when we want to apply multiple functions on the same column or same function on multiple columns or both.

```
In [287]: df.groupby('preg_count').agg({"glucose": "mean", "age": "max"})
```

```
Out[287]:                glucose   age
          preg_count
          0.0         122.838095  67.0
          1.0         114.669421  62.0
          2.0         111.659794  72.0
          3.0         123.957746  63.0
          4.0         127.426230  70.0
          5.0         118.859649  69.0
          6.0         120.617021  66.0
          7.0         135.093023  61.0
          8.0         136.558824  68.0
          9.0         131.769231  81.0
          10.0        122.260870  63.0
```

```
11.0        125.500000  51.0
12.0        113.555556  62.0
13.0        125.500000  52.0
14.0        137.500000  46.0
15.0        136.000000  43.0
17.0        163.000000  47.0
```

We can also specify the target column name using nested dictionary as follows. However, this will create a multilevel index for the column names as follows.

```
In [303]: aggregators = {"glucose": {"glucose_mean": "mean",
                                     "glucose_count": "count"},
                         "age": {"age_max": "max",
                                 "age_mean": "mean"}}

          df.groupby('preg_count').agg(aggregators)
```

Out[303]:

	glucose		age	
	glucose_mean	glucose_count	age_max	age_mean
preg_count				
0.0	122.838095	105	67.0	27.490566
1.0	114.669421	121	62.0	27.280992
2.0	111.659794	97	72.0	27.302083
3.0	123.957746	71	63.0	29.138889
4.0	127.426230	61	70.0	32.769231
5.0	118.859649	57	69.0	39.320755
6.0	120.617021	47	66.0	39.562500
7.0	135.093023	43	61.0	41.113636
8.0	136.558824	34	68.0	45.638889
9.0	131.769231	26	81.0	44.400000
10.0	122.260870	23	63.0	43.043478
11.0	125.500000	10	51.0	44.800000
12.0	113.555556	9	62.0	47.444444
13.0	125.500000	10	52.0	43.750000
14.0	137.500000	2	46.0	42.000000
15.0	136.000000	1	43.0	43.000000
17.0	163.000000	1	47.0	47.000000

4.7.3 Pivot table

Pivot tables are mainly used to present results/summary in a table format. The 'index' and 'columns' should be categorical/nominal values.

```
In [304]: df.pivot_table(index="preg_count", columns="class", values="glucose", aggfunc='sum')
```

Out[304]:

class	0.0	1.0
preg_count		

```
0.0          7220.0  5005.0
1.0          9804.0  3492.0
2.0          7904.0  2472.0
3.0          4921.0  3602.0
4.0          4443.0  2992.0
5.0          3914.0  2631.0
6.0          3314.0  2118.0
7.0          2283.0  3526.0
8.0          1556.0  2787.0
9.0           981.0  2445.0
10.0         1552.0  1166.0
11.0          453.0   802.0
12.0          555.0   467.0
13.0          586.0   669.0
14.0            NaN   275.0
15.0            NaN   136.0
17.0            NaN   163.0
```

There is another variant to pivot_table() method - pivot() which is limited to DataFrames with no duplicate rows. If you are certain that there are no duplicate rows, you can simple apply pivot() method.

The pivot_table() is a generalization of pivot() method. It allows us to aggregate duplicate values by applying aggregators such as mean, median, etc.

4.7.4 Multi-Indexing (stack/unstack)

- stack (taller) - converts innermost column index to innermost row index
- unstack (wider) - converts innermost row index to innermost column index

Infact, pivot_table is a special case of stack/unstack operation

```
In [305]: df.columns

Out[305]: Index(['preg_count', 'glucose', 'BP', 'skin_thick', 'insulin', 'BMI',
                 'pedigree', 'age', 'class'],
                dtype='object')

In [306]: df_multi = df.fillna(0).groupby(['preg_count', 'age', 'class'])['glucose', 'BP',
          'skin_thick'].mean()
          df_multi.head()

Out[306]:                          glucose         BP  skin_thick
          preg_count age  class
          0.0        0.0  0.0     94.333333  71.000000   29.000000
                     21.0 0.0    112.105263  70.052632   17.789474
                          1.0    155.333333  40.000000   36.666667
                     22.0 0.0    112.866667  52.800000   21.400000
                          1.0    141.000000  70.666667   25.666667
```

Stack

```
In [307]: df_multi.stack().head(10)

Out[307]: preg_count  age   class
          0.0         0.0   0.0    glucose        94.333333
                                   BP             71.000000
                                   skin_thick     29.000000
                    21.0  0.0    glucose       112.105263
                                   BP             70.052632
                                   skin_thick     17.789474
                          1.0    glucose       155.333333
                                   BP             40.000000
                                   skin_thick     36.666667
                    22.0  0.0    glucose       112.866667
          dtype: float64
```

The innermost column index is converted to innermost row index making the DataFrame TALLER. The no. of rows has increased from 5 to 10.

Unstack

```
In [308]: df_multi.unstack().head()

Out[308]:                    glucose                    BP            skin_thick  \
          class                  0.0         1.0        0.0        1.0        0.0
          preg_count age
          0.0        0.0    94.333333         NaN  71.000000        NaN  29.000000
                     21.0  112.105263  155.333333  70.052632  40.000000  17.789474
                     22.0  112.866667  141.000000  52.800000  70.666667  21.400000
                     23.0  117.666667  133.000000  73.333333  76.000000  27.833333
                     24.0   99.400000  132.000000  64.600000  58.000000  30.800000


          class                  1.0
          preg_count age
          0.0        0.0         NaN
                     21.0  36.666667
                     22.0  25.666667
                     23.0  13.333333
                     24.0  25.200000
```

The innermost row index is converted to innermost column index making the DataFrame WIDER. Note that we have the flexibility to choose which level of index will be converted. The default is the innermost row/column index.

4.7.5 Cross-tab

- Frequency table (by default) but an aggregation function can be passed as well

A simple frequency table with count of each class for different 'preg_count'

```
In [309]: pd.crosstab(index=df['class'], columns=df['preg_count'])
```

```
Out[309]: preg_count  0.0   1.0   2.0   3.0   4.0   5.0   6.0   7.0   8.0   9.0   10.0  \
          class
          0.0           67    97    79    46    40    35    31    20    15     9    13
          1.0           35    25    18    24    22    20    16    25    20    17    10

          preg_count  11.0  12.0  13.0  14.0  15.0  17.0
          class
          0.0            4     5     5     0     0     0
          1.0            7     4     5     2     1     1
```

Instead of getting frequency, we can ask the function to use a specific column for values with an aggregation function. When an aggregation function is passed it is similar to the pivot_table (please refer below).

```
In [310]: pd.crosstab(index=df['class'], columns=df['preg_count'],
                      values=df['BMI'], aggfunc='mean').applymap(lambda x: "%.2f" %x)
```

```
Out[310]: preg_count   0.0    1.0    2.0    3.0    4.0    5.0    6.0    7.0    8.0   \
          class
          0.0         31.26  29.74  29.74  29.40  31.39  30.86  30.43  29.98  30.94
          1.0         39.41  38.42  33.45  32.86  33.73  36.92  31.77  34.68  32.79

          preg_count   9.0    10.0   11.0   12.0   13.0   14.0   15.0   17.0
          class
          0.0         29.60  30.65  37.12  30.56  33.28    nan    nan    nan
          1.0         33.33  29.80  39.39  34.58  36.72  35.10  37.10  40.90
```

```
In [311]: df.pivot_table(index='class', columns='preg_count', values='BMI').applymap(lambda x:
          "%.2f" %x)
```

```
Out[311]: preg_count   0.0    1.0    2.0    3.0    4.0    5.0    6.0    7.0    8.0   \
          class
          0.0         31.26  29.74  29.74  29.40  31.39  30.86  30.43  29.98  30.94
          1.0         39.41  38.42  33.45  32.86  33.73  36.92  31.77  34.68  32.79

          preg_count   9.0    10.0   11.0   12.0   13.0   14.0   15.0   17.0
          class
          0.0         29.60  30.65  37.12  30.56  33.28    nan    nan    nan
          1.0         33.33  29.80  39.39  34.58  36.72  35.10  37.10  40.90
```

4.7.6 Merging multiple DataFrames

Let's create 3 DataFrames from df

```
In [312]: df1 = df[['preg_count', 'glucose', 'class']].copy() # deep copy
          df2 = df[['preg_count', 'insulin', 'BP']].copy()
          df3 = df[['BMI', 'age', 'preg_count']].copy()

          print(df.shape)
```

```
(768, 9)
```

Drop few rows to introduce discontinuity in indices for each DataFrame

```
In [313]: df1 = df1[df1['preg_count']>2]
          df2 = df2[df2['BP']>50]
          df3 = df3[df3['age']>25]

          print(df1.shape)
          print(df2.shape)
          print(df3.shape)
```

```
(409, 3)
(681, 3)
(483, 3)
```

```
In [314]: display("df1.head()", "df2.head()", "df3.head()")
```

```
Out[314]: df1.head()
             preg_count  glucose  class
          0         6.0    148.0    1.0
          2         8.0    183.0    1.0
          5         5.0    116.0    0.0
          6         3.0     78.0    1.0
          7        10.0    115.0    0.0

          df2.head()
             preg_count  insulin    BP
          0         6.0      0.0  72.0
          1         1.0      0.0  66.0
          2         8.0      0.0  64.0
          3         1.0     94.0  66.0
          5         5.0      0.0  74.0

          df3.head()
```

```
    BMI   age  preg_count
0  33.6  50.0         6.0
1  26.6  31.0         1.0
2  23.3  32.0         8.0
4  43.1  33.0         0.0
5  25.6  30.0         5.0
```

Merge

- To combine DataFrames along axis=1 or columns based on the index or specified column values as the key
- Database style operation

```
In [315]: print(pd.merge(df1, df2, left_index=True, right_index=True, how='outer',
          suffixes=["_df1", "_df2"]).shape)
          pd.merge(df1, df2, left_index=True, right_index=True, how='outer', suffixes=["_df1",
          "_df2"]).head()
```

```
(720, 6)
```

```
Out[315]:    preg_count_df1  glucose  class  preg_count_df2  insulin    BP
          0             6.0    148.0    1.0             6.0      0.0  72.0
          1             NaN      NaN    NaN             1.0      0.0  66.0
          2             8.0    183.0    1.0             8.0      0.0  64.0
          3             NaN      NaN    NaN             1.0     94.0  66.0
          5             5.0    116.0    0.0             5.0      0.0  74.0
```

Merges two DataFrames based on index and fills the values for the missing columns with NaNs. The above operation is "outer" join and we can also perform "inner", "left", or "right" joins.

how - type of join operation similar to what we use for combining databases - "outer", "inner", "left", "right" suffixes - suffixes to use incase we have same columns on both the DataFrames

Please note that if left_index=True and right_index=True are not specified by default, it uses the common columns in both the DataFrames as the key for combining

```
In [316]: df12 = pd.merge(df1, df2, left_on="preg_count", right_on="preg_count", how='outer',
          suffixes=["_df1", "_df2"])
          df12.shape
```

```
Out[316]: (18249, 5)
```

```
In [317]: display("df12.head()", "df2[df2['preg_count']==6].head()")
```

```
Out[317]: df12.head()
             preg_count  glucose  class  insulin    BP
          0         6.0    148.0    1.0      0.0  72.0
```

```
1          6.0    148.0    1.0      0.0  92.0
2          6.0    148.0    1.0    228.0  72.0
3          6.0    148.0    1.0      0.0  64.0
4          6.0    148.0    1.0    156.0  74.0

df2[df2['preg_count']==6].head()
     preg_count  insulin    BP
0           6.0      0.0  72.0
33          6.0      0.0  92.0
95          6.0    228.0  72.0
121         6.0      0.0  64.0
165         6.0    156.0  74.0
```

It blows out. This is because for each combination of "preg_count"=x, "insulin", "BP" in df1, df2[df2['preg_count']==x] is inserted into the output DataFrame. Please use columns with unique values to combine unless otherwise you desire the above output.

Concat

- To combine along any axis (axis 0 or 1)

```
In [318]: # Merging columns or axis = 1
          df_concat = pd.concat([df1, df2], axis=1)
          df_concat.head()

Out[318]:    preg_count  glucose  class  preg_count  insulin    BP
          0         6.0    148.0    1.0         6.0      0.0  72.0
          1         NaN      NaN    NaN         1.0      0.0  66.0
          2         8.0    183.0    1.0         8.0      0.0  64.0
          3         NaN      NaN    NaN         1.0     94.0  66.0
          5         5.0    116.0    0.0         5.0      0.0  74.0

In [319]: df_concat.shape

Out[319]: (720, 6)
```

The DataFrame widened. The DataFrames are stacked based on the index on both the DataFrames and common columns are retained with the same name (as duplicates)

Options to know

- ignore_index=True - to reset the index after merging
- keys=["_1","_2"] - to add row/column index indicating where the data came from
- join - specific which database merge operation to perform - "outer"(default), "inner", "left", or "right"

```
In [320]: # Merging rows-wise or axis = 0
          df_concat = pd.concat([df1, df2], axis=0)
          df_concat.head()
```

```
Out[320]:    BP  class  glucose  insulin  preg_count
          0 NaN    1.0    148.0      NaN         6.0
          2 NaN    1.0    183.0      NaN         8.0
          5 NaN    0.0    116.0      NaN         5.0
          6 NaN    1.0     78.0      NaN         3.0
          7 NaN    0.0    115.0      NaN        10.0

In [321]: df_concat.shape

Out[321]: (1090, 5)
```

The DataFrame has increased in height. The df1 is simply stacked on top of df2.

Append Append operation is similar to that of lists/arrays. Using append() we can concat 2 DataFrames in few keystrokes than using the previous method

```
In [322]: df_append = df1.append(df2)
          df_append.head()

Out[322]:    BP  class  glucose  insulin  preg_count
          0 NaN    1.0    148.0      NaN         6.0
          2 NaN    1.0    183.0      NaN         8.0
          5 NaN    0.0    116.0      NaN         5.0
          6 NaN    1.0     78.0      NaN         3.0
          7 NaN    0.0    115.0      NaN        10.0

In [323]: df_append.shape

Out[323]: (1090, 5)

In [324]: df1.shape

Out[324]: (409, 3)
```

However, append() is an inefficient method for combining DataFrames. Instead of replacing df1's object it creates an new object for df1.append(df2) and they all are saved in memory buffer. So when multiple DataFrames are appended it consumes lot of memory and slows down. Hence it is recommended to use pd.concat() when combining many DataFrames.

4.7.7 Transpose

- Row index -> column index and column index -> row index

```
In [328]: df_T = df.T
          df_T.iloc[:, :15]

Out[328]:                0      1      2      3      4      5      6  \
          preg_count   6.000  1.000  8.000  1.000  0.000  5.000  3.000
```

```
glucose     148.000  85.000  183.000  89.000  137.000  116.000  78.000
BP           72.000  66.000   64.000  66.000   40.000   74.000  50.000
skin_thick   35.000  29.000    0.000     NaN   35.000    0.000  32.000
insulin       0.000   0.000    0.000  94.000  168.000    0.000  88.000
BMI          33.600  26.600   23.300  28.100   43.100   25.600  31.000
pedigree      0.627   0.351    0.672   0.167    2.288    0.201   0.248
age          50.000  31.000   32.000  21.000   33.000   30.000  26.000
class         1.000   0.000    1.000   0.000    1.000    0.000   1.000

                 7        8        9        10       11       12       13  \
preg_count   10.000    2.000    8.000    4.000   10.000   10.000    1.000
glucose     115.000  197.000  125.000  110.000  168.000  139.000  189.000
BP            0.000   70.000   96.000   92.000   74.000   80.000   60.000
skin_thick    0.000   45.000    0.000    0.000    0.000    0.000   23.000
insulin       0.000  543.000    0.000    0.000    0.000    0.000  846.000
BMI          35.300   30.500    0.000   37.600   38.000   27.100   30.100
pedigree      0.134    0.158    0.232    0.191    0.537    1.441    0.398
age          29.000   53.000   54.000   30.000      NaN   57.000   59.000
class         0.000    1.000    1.000    0.000    1.000    0.000      NaN

                14
preg_count   5.000
glucose     166.000
BP           72.000
skin_thick   19.000
insulin     175.000
BMI          25.800
pedigree      0.587
age             NaN
class         1.000
```

```
In [329]: df_T.shape
```

```
Out[329]: (9, 768)
```

4.8 Save Data

4.8.1 csv format

```
In [85]: df.to_csv( path + "df.csv", index=False)
```

index = False - to drop index before writing the data to a file

4.8.2 Excel format

Saving single sheet

```
In [82]: df.to_excel( path + "df.xlsx" )
```

Saving in multiple sheets

```
In [87]: # define an excel writer
         df_sheets = pd.ExcelWriter(path + "df_multiple.xlsx")

         # Write the DataFrames one-by-one with a different sheet name
         df.to_excel(df_sheets, sheet_name='sheet-1', index=False)
         df1.to_excel(df_sheets, sheet_name='sheet-2', index=False)
         df2.to_excel(df_sheets, sheet_name='sheet-3', index=False)
         df3.to_excel(df_sheets, sheet_name='sheet-4', index=False)

         # Save the writer
         df_sheets.save()
```

4.8.3 pickle format

```
In [106]: df.to_pickle( path + "df.pkl" )
```

5
BIBILIOGRAPHY

1. http://Pandas.pydata.org/Pandas-docs/version/0.23/tutorials.html
2. https://docs.python.org/3/tutorial/datastructures.html
3. https://docs.scipy.org/doc/numpy/user/quickstart.html
4. https://wiki.python.org/moin/TimeComplexity