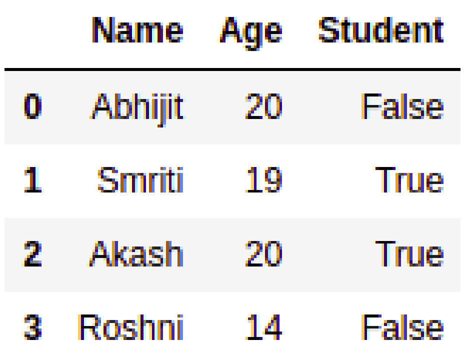
UNIT III
Data Manipulation in Python using Pandas

In Machine Learning, the model requires a dataset to operate, i.e. to train and test. But data doesn’t come fully prepared and ready to use. There are discrepancies like “Nan”/ “Null” / “NA” values in many rows and columns. Sometimes the data set also contains some of the row and columns which are not even required in the operation of our model. In such conditions, it requires proper cleaning and modification of the data set to make it an efficient input for our model. We achieve that by practicing “**Data Wrangling**” before giving data input to the model.

Ok, So let’s dive into the programming part. Our first aim is to create a Pandas dataframe in Python, as you may know, pandas is one of the most used libraries of Python.

```
# importing the pandas library
import pandas as pd



# creating a dataframe object
student_register = pd.DataFrame()

# assigning values to the
# rows and columns of the
# dataframe
student_register['Name'] = ['Abhijit',
                            'Smriti',
                            'Akash',
                            'Roshni']

student_register['Age'] = [20, 19, 20, 14]
student_register['Student'] = [False, True,
                            True, False]

Print (student_register)
```



	Name	Age	Student
0	Abhijit	20	False
1	Smriti	19	True
2	Akash	20	True
3	Roshni	14	False

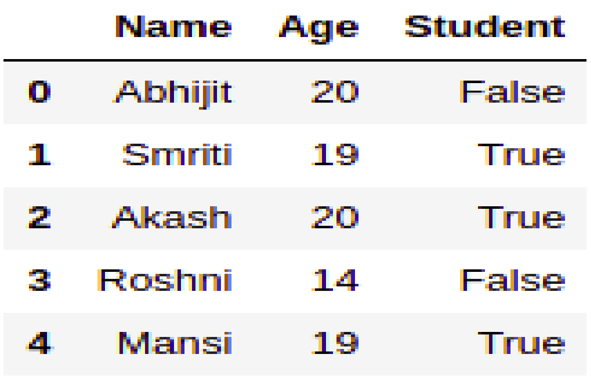
As you can see, the dataframe object has four rows [0, 1, 2, 3] and three columns[“Name”, “Age”, “Student”] respectively. The column which contains the index values i.e. [0, 1, 2, 3] is known as the **index column**, which is a default part in pandas datagram. We can change that as per our requirement too because Python is powerful.

Next, for some reason we want to add a new student in the datagram, i.e you want to add a new row to your existing data frame, that can be achieved by the following code snippet.

One important concept is that the “dataframe” object of Python, consists of rows which are “series” objects instead, stack together to form a table. Hence adding a new row means creating a new series object and appending it to the dataframe.

```
# creating a new pandas
# series object
new_person = pd.Series(['Mansi', 19, True],
                        index = ['Name', 'Age',
                                 'Student'])

# using the .append() function
# to add that row to the dataframe
student_register.append(new_person, ignore_index = True)

Print (student_register)
```

OUTPUT :



	Name	Age	Student
0	Abhijit	20	False
1	Smriti	19	True
2	Akash	20	True
3	Roshni	14	False
4	Mansi	19	True

Before processing and wrangling any data you need to get the total overview of it, which includes statistical conclusions like **standard deviation(std), mean and it's quartile distributions**. Also, you need to know the exact information of each column, i.e. what type of value it stores and how many of them are unique. There are three support functions, .shape, .info() and .describe(), which outputs the shape of the table, information on rows and columns, and statistical information of the dataframe (numerical column only) respectively.

```
# for showing the dimension
# of the dataframe
print('Shape')
print(student_register.shape)

# showing info about the data
print("\n\nInfo\n")
student_register.info()

# for showing the statistical
# info of the dataframe
print("\n\nDescribe")
student_register.describe()
```

OUTPUT:

```
Shape
(4, 3)
```

```
Info

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   Name     4 non-null      object
 1   Age      4 non-null      int64
 2   Student  4 non-null      bool
dtypes: bool(1), int64(1), object(1)
memory usage: 196.0+ bytes
```

```
Describe
```

	Age
count	4.000000
mean	18.250000
std	2.872281
min	14.000000
25%	17.750000
50%	19.500000
75%	20.000000
max	20.000000

In the above example, the .shape function gives an output (4, 3) as that is the size of the created dataframe.

The description of the output given by .info() method is as follows:

1. **"RangeIndex"** describes about the index column, i.e. [0, 1, 2, 3] in our datagram. Which is the number of rows in our dataframe.
2. As the name suggests **"Data columns"** give the total number of columns as output.
3. **"Name", "Age", "Student"** are the name of the columns in our data, "non-null " tells us that in the corresponding column, there is no NA/ Nan/ None value exists. "object", "int64″ and "bool" are the datatypes each column have.
4. **"dtype"** gives you an overview of how many data types present in the datagram, which in term simplifies the data cleaning process.
   Also, in high-end machine learning models, **"memory usage"** is an important term, we can't neglect that.

The description of the output given by .describe() method is as follows:

1. **count** is the number of rows in the dataframe.
2. **mean** is the mean value of all the entries in the "Age" column.
3. **std** is the standard deviation of the corresponding column.
4. **min** and max are the minimum and maximum entry in the column respectively.
5. 25%, 50% and 75% are the **First Quartiles**, **Second Quartile(Median)** and **Third Quartile** respectively, which gives us important info on the distribution of the dataset and makes it simpler to apply an ML model.

NumPy array in Python

Python lists are a substitute for arrays, but they fail to deliver the performance required while computing large sets of numerical data. To address this issue we use a python library called NumPy. The word NumPy stands for Numerical Python. NumPy offers an array object called ndarray. They are similar to standard python sequences but differ in certain key factors.

NumPy arrays vs inbuilt Python sequences

- Unlike lists, NumPy arrays are of fixed size, and changing the size of an array will lead to the creation of a new array while the original array will be deleted.
- All the elements in an array are of the same type.
- Numpy arrays are faster, more efficient, and require less syntax than standard python sequences.

Note: Various scientific and mathematical Python-based packages use Numpy. They might take input as an inbuilt Python sequence but they are likely to convert the data into a NumPy array in order to attain faster processing. This explains the need to understand NumPy.

Why is Numpy so fast?

Numpy arrays are written mostly in C language. Being written in C, the NumPy arrays are stored in contiguous memory locations which makes them accessible and easier to manipulate. This means that you can get the performance level of a C code with the ease of writing a python program.

Using Numpy Arrays

If you don't have NumPy installed in your system, you can do so by following these steps. After installing NumPy you can import it in your program like this

```
import numpy as np
```

Here np is a commonly used alias to NumPy.

Numpy array from a list

You can use the np alias to create ndarray of a list using the array() method.

```
li = [1,2,3,4]

numpyArr = np.array(li)
```

or

```
numpyArr = np.array([1,2,3,4])
```

The list is passed to the array() method which then returns a NumPy array with the same elements.

Example:
The following example shows how to initialize a NumPy array from a list.

- Python3

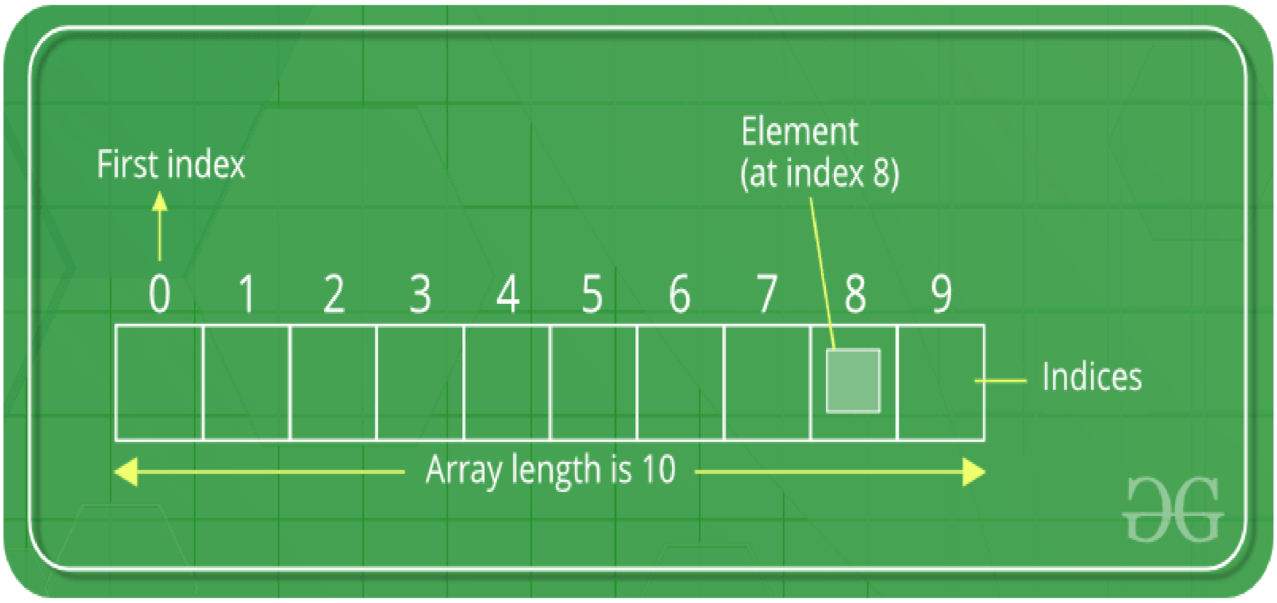
```
import numpy as np



li = [1, 2, 3, 4]

numpyArr = np.array(li)

print(numpyArr)
```

Output:

```
[1 2 3 4]
```

Python Arrays

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array). For simplicity, we can think of an array a fleet of stairs where on each step is placed a value (let’s say one of your friends). Here, you can identify the location of any of your friends by simply knowing the count of the step they are on. Array can be handled in Python by a module named **array**. They can be useful when we have to manipulate only a specific data type values. A user can treat lists as arrays. However, user cannot constraint the type of elements stored in a list. If you create arrays using the **array** module, all elements of the array must be of the same type.



Creating a Array

Array in Python can be created by importing array module. **array(*data_type, value_list*)** is used to create an array with data type and value list specified in its arguments.

- Python3

```
# Python program to demonstrate

# Creation of Array



# importing "array" for array creations

import array as arr



# creating an array with integer type

a = arr.array('i', [1, 2, 3])



# printing original array

print("The new created array is : ", end=" ")
```

```
for i in range(0, 3):

    print(a[i], end=" ")

print()



# creating an array with double type

b = arr.array('d', [2.5, 3.2, 3.3])



# printing original array

print("\nThe new created array is : ", end=" ")

for i in range(0, 3):

    print(b[i], end=" ")
```

Output

```
The new created array is :  1 2 3


The new created array is :  2.5 3.2 3.3
```

Output :

```
The new created array is :  1 2 3
The new created array is :  2.5 3.2 3.3
```

Hierarchical Indexing

Up to this point we've been focused primarily on one-dimensional and two-dimensional data, stored in Pandas `Series` and `DataFrame` objects, respectively. Often it is useful to go beyond this and store higher-dimensional data–that is, data indexed by more than one or two keys. While Pandas does provide `Panel` and `Panel4D` objects that natively handle three-dimensional and four-dimensional data (see Aside: Panel Data), a far more common pattern in practice is to make use of *hierarchical indexing* (also known as *multi-indexing*) to incorporate multiple index *levels* within a single index. In this way, higher-dimensional data can be compactly represented within the familiar one-dimensional `Series` and two-dimensional `DataFrame` objects.

In this section, we'll explore the direct creation of `MultiIndex` objects, considerations when indexing, slicing, and computing statistics across multiply indexed data, and useful routines for converting between simple and hierarchically indexed representations of your data.

We begin with the standard imports:

In [1]:

```
importpandasaspdimportnumpyasnp
```

A Multiply Indexed Series

Let's start by considering how we might represent two-dimensional data within a one-dimensional `Series`. For concreteness, we will consider a series of data where each point has a character and numerical key.

The bad way

Suppose you would like to track data about states from two different years. Using the Pandas tools we've already covered, you might be tempted to simply use Python tuples as keys:

In [2]:

```
index = [('California', 2000), ('California', 2010),
         ('New York', 2000), ('New York', 2010),
         ('Texas', 2000), ('Texas', 2010)]populations = [33871648, 37253956,
18976457, 19378102,
20851820, 25145561]pop = pd.Series(populations, index=index)pop
```

Out[2]:

```
(California, 2000)    33871648
(California, 2010)    37253956
(New York, 2000)      18976457
(New York, 2010)      19378102
(Texas, 2000)         20851820
(Texas, 2010)         25145561
dtype: int64
```

With this indexing scheme, you can straightforwardly index or slice the series based on this multiple index:

In [3]:

```
pop[('California', 2010):('Texas', 2000)]
```

Out[3]:

```
(California, 2010)    37253956
(New York, 2000)      18976457
(New York, 2010)      19378102
(Texas, 2000)         20851820
dtype: int64
```

But the convenience ends there. For example, if you need to select all values from 2010, you'll need to do some messy (and potentially slow) munging to make it happen:

In [4]:

```
pop[[i for i in pop.index if i[1] ==2010]]
```

Out[4]:

```
(California, 2010)    37253956
(New York, 2010)      19378102
(Texas, 2010)         25145561
dtype: int64
```

This produces the desired result, but is not as clean (or as efficient for large datasets) as the slicing syntax we've grown to love in Pandas.

The Better Way: Pandas MultiIndex

Fortunately, Pandas provides a better way. Our tuple-based indexing is essentially a rudimentary multi-index, and the Pandas `MultiIndex` type gives us the type of operations we wish to have. We can create a multi-index from the tuples as follows:

In [5]:

```
index = pd.MultiIndex.from_tuples(index)index
```

Out[5]:

```
MultiIndex(levels=[['California', 'New York', 'Texas'], [2000, 2010]],
           labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]])
```

Notice that the `MultiIndex` contains multiple *levels* of indexing–in this case, the state names and the years, as well as multiple *labels* for each data point which encode these levels.

If we re-index our series with this `MultiIndex`, we see the hierarchical representation of the data:

In [6]:

```
pop = pop.reindex(index)pop
```

Out[6]:

```
California   2000      33871648
             2010      37253956
New York     2000      18976457
             2010      19378102
Texas        2000      20851820
             2010      25145561
dtype: int64
```

Here the first two columns of the `Series` representation show the multiple index values, while the third column shows the data. Notice that some entries are missing in the first column: in this multi-index representation, any blank entry indicates the same value as the line above it.

Now to access all data for which the second index is 2010, we can simply use the Pandas slicing notation:

In [7]:

```
pop[:, 2010]
```

Out[7]:

```
California      37253956
New York        19378102
Texas           25145561
dtype: int64
```

The result is a singly indexed array with just the keys we're interested in. This syntax is much more convenient (and the operation is much more efficient!) than the home-spun tuple-based multi-indexing solution that we started with. We'll now further discuss this sort of indexing operation on hieararchically indexed data.