

National University of Computer and Emerging Sciences



Lab Manual CL461-Artificial Intelligence Lab

Department of Computer Science
FAST-NU, Lahore, Pakistan

Table of Contents

1	Objectives	4
1	Python Iterators	4
1.1	Building Custom Iterators	4
2	Classes and Objects	5
2.1	Defining a Class in Python	5
2.2	Creating Objects	6
2.3	Constructors in Python	6
2.4	7
2.5	Inheritance in Python	7
2.5.1	7
2.5.2	Python Inheritance Syntax	7
3	Python Stacks	7
3.1	Stacks via Lists	7
3.2	Stacks via collections.deque	8
3.3	Stacks via queue.LifoQueue	8
4	Python Queues	9
5	Task Distribution	10
6	Anaconda (Python Distribution)	10
6.1	Installation Guide	10
7	Python Libraries for Data Science	11
7.1	Numpy	11
7.1.1	NumPy Installation	11
7.1.2	Arrays	12
7.1.3	Attributes of Array Object	13
7.1.4	Array Slicing and Indexing	14
7.1.5	Basic Array Operations	15
7.1.6	Datatypes	16
7.1.7	Input/Output Operations	17
7.2	Pandas	18
7.2.1	Pandas Installation	18
7.2.2	Series	18
7.2.3	Dataframe	19

7.2.4	Indexing	21
7.2.5	Input Output Operations	21
7.2.6	Other Operations	21
7.2.7	Retrieving Series/DataFrame Information	21
7.2.8	Application of Functions	22
8	Dataset Handling	22
8.1	Significance of Data	22
8.2	Types of Data Sets	23
8.2.1	Numerical Data Sets	23
8.2.2	Bivariate Data Sets	23
8.2.3	Multivariate Data Sets	23
8.2.4	Categorical Data Sets	23
8.2.5	Correlation Data Sets	24
8.3	Mean, Median, Mode and Range of Data-Sets	24
8.4	Types of Attributes	24
8.5	Properties of Dataset	25
8.6	Dataset Repositories	25
8.7	Properties of Attributes in a dataset	25
8.7.1	Missing Values	25
8.7.2	Outliers	25
8.8	Iris Flower Dataset	26
8.9	Load the Dataset	26

1 Objectives

After performing this lab, students shall be able to understand the following Python concepts and applications:

- ✓ Anaconda Introduction
- ✓ Application of NumPy
- ✓ Application of Pandas
- ✓ Dataset handling

1 Python Iterators

An iterator is an object that contains a countable number of values. It is an object that can be iterated upon, meaning that you can traverse through all the values. Technically, in Python, an iterator is an object which implements the iterator protocol, which consists of the methods `iter()` and `next()`.

Every time you ask an iterator for the **next** item, it calls its `__next__` method. If there is another value available, the iterator returns it. If not, it raises a `StopIteration` exception. More information about iterators can be found [here](#).

This behavior (only returning the next element when asked to) has two main advantages:

1. Iterators need less space in memory. They remember the last value and a rule to get to the next value instead of memorizing every single element of a (potentially very long) sequence.
2. Iterators don't check how long the sequence they produce might get. For instance, they don't need to know how many lines a file has or how many files are in a folder to iterate through them.

(One important note: don't confuse iterators with iterables. Iterables are objects that can create iterators by using their `__iter__` method)

1. Building Custom Iterators

`__iter__` and `__next__`
() the ()

Building an iterator from scratch is easy in Python. We just have to implement the methods.

`__iter__`
()

The `__iter__` method returns the iterator object itself. If required, some initialization can be performed.

`__next__`
()

The `__next__` method must return the next item in the sequence.

Return an iterator from a tuple, and print each value:

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))
)
print(next(myit))
)
print(next(myit))
)
```

To iterate the characters of a string:

```
mystr = "banana"
for x in mystr:
    print(x)
```

The **for** loop actually creates an iterator object and executes the next() method for each loop.

2 Classes and Objects

Python is an object oriented programming language. Almost everything in Python is an object, with its properties and methods. A Class is like an object constructor, or a "blueprint" for creating objects.

2.1 Defining a Class in Python

Like function definitions begin with the [def](#) keyword in Python, class definitions begin with a [class](#) keyword.

The first string inside the class is called docstring and has a brief description of the class. Although not mandatory, this is highly recommended.

Here is a simple class definition,

```
class MyNewClass:
    '''This is a docstring. I have created a new class'''
    pass
```

A class creates a new local [namespace](#) where all its attributes are defined. Attributes may be data or functions.

As soon as we define a class, a new class object is created with the same name. This class object

allows us to access the different attributes as well as to instantiate new objects of that class.

```
class Person:
    "This is a person class"
    age = 10

    def greet(self):
        print('Hello')

# Output: 10
print(Person.age)

# Output: <function Person.greet>
print(Person.greet)

# Output: "This is a person class"
print(Person.__doc__)
```

2.2 Creating Objects

The procedure to create an object is similar to a [function](#) call.

```
harry = Person()
```

2.3 Constructors in Python

Class functions that begin with double underscore `__` are called special functions as they have special meaning.

Of one particular interest is the `__init__()` function. This special function gets called whenever a new object of that class is instantiated.

This type of function is also called constructors in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

```
class ComplexNumber:
    def __init__(self, r=0, i=0):
        self.real = r
        self.imag = i

    def get_data(self):
        print(f'{self.real}+{self.imag}j')
```

```
# Create a new ComplexNumber object
num1 = ComplexNumber(2, 3)
```

We can even delete the object itself, using the del statement.

```
del c1
```

2.4

2.5 Inheritance in Python

Inheritance is a powerful feature in object oriented programming.

It refers to defining a new [class](#) with little or no modification to an existing class. The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class.

2.5.1

2.5.2 Python Inheritance Syntax

```
class BaseClass:
    Body of base class
class DerivedClass(BaseClass):
    Body of derived class
```

3 Python Stacks

A stack is a data structure that stores items in a Last-In/First-Out manner. We will look at three different implementations of stacks in Python.

- Using list data structure
- Using collections.deque module
- Using queue.LifoQueue class

3.1 Stacks via Lists

The built-in Python list object can be used as a stack. For stack **.push()** method, we can use **.append()** method of list. **.pop()** method of list can remove elements in LIFO order. Popping an empty list (stack) will raise an **IndexError**. To get the top most item (peek) in the stack, write **list[-1]**. Bigger lists (stacks) often run into speed issues as they continue to grow. **list**

may be familiar, but it should be avoided because it can potentially have memory reallocation issues.

```
myStack = []

myStack.append('a')
myStack.append('b')
myStack.append('c')

myStack      # ['a', 'b', 'c']

myStack.pop()      # 'c'
myStack.pop()      # 'b'
myStack.pop()      # 'a'
```

3.2 Stacks via collections.deque

This method solves the speed problem we face in lists. The **deque** class has been designed as such to provide O(1) time complexity for append and pop operations. The **deque** class is built on top of a doubly linked list structure which provides faster insertion and removal. Popping an empty **deque** gives the same **IndexError**. Read more about it [here](#). Also, to know more about linked lists in Python, read [this](#).

```
from collections import deque

myStack = deque()

myStack.append('a')
myStack.append('b')
myStack.append('c')
myStack      # deque(['a', 'b', 'c'])
myStack.pop()      # 'c'
myStack.pop()      # 'b'
myStack.pop()      # 'a'
```

3.3 Stacks via queue.LifoQueue

LifoQueue uses **.put()** and **.get()** to add and remove data from the stack. **LifoQueue** is designed to be fully thread-safe. But use it only if you are working with threads. Otherwise, **deque** works well. The **.get()** method by default will wait until an item is available. That means it waits forever if no item is present in the list. Instead, **get_nowait()** method would immediately raise empty stack error. Read more about it [here](#).


```

from queue import LifoQueue
myStack = LifoQueue()

myStack.put('a')
myStack.put('b')
myStack.put('c')

myStack      # <queue.LifoQueue object at 0x7f408885e2b0>

myStack.get()      # 'c'
myStack.get()      # 'b'
myStack.get()      # 'a'

```

4 Python Queues

A queue is FIFO data structure. The insert and delete operations are sometimes called **enqueue** and **dequeue**. We can use list as a queue as well. To follow FIFO, use **pop(0)** to remove the first element of the queue. But as discussed before, lists are slow. They are not ideal from a performance perspective.

We can use the **collections.deque** class again to implement Python queues. They work best for non-threaded programs. We can also use **queue.Queue** class. But it works well with synchronized programs.

If you are not looking for parallel processing, **collections.deque** is a good default choice.

```

from collections import deque
q = deque()
q.append('eat')
q.append('sleep')
q.append('code')
q      # deque(['eat', 'sleep', 'code'])
q.popleft()      # 'eat'
q.popleft()      # 'sleep'
q.popleft()      # 'code'
q.popleft()      # IndexError: "pop from an empty deque"

```

5 Task Distribution

Total Time	170 Minutes
Anaconda Introduction/Installation Guide	20 Minutes
Application of NumPy	20 Minutes
Application of Pandas	20 Minutes
Dataset handling	10 Minutes
Exercise	90 Minutes
Online Submission	10 Minutes

6 Anaconda (Python Distribution)

Anaconda is a distribution of the Python and R programming languages for scientific computing (data science, machine learning applications, large-scale data processing, predictive analytics, etc.). The distribution includes data-science packages suitable for Windows, Linux, and macOS. It is developed and maintained by Anaconda, Inc., which was founded by Peter Wang and Travis Oliphant in 2012.

Anaconda Navigator

Anaconda Navigator is a desktop graphical user interface (GUI) included in Anaconda distribution that allows users to launch applications and manage conda packages, environments and channels without using command-line commands. Navigator can search for packages on Anaconda Cloud or in a local Anaconda Repository, install them in an environment, run the packages and update them. It is available for Windows, macOS and Linux.

The following applications are available by default in Navigator:

- JupyterLab
- Jupyter Notebook
- QtConsole
- Spyder
- Glue
- Orange
- RStudio
- Visual Studio Code

6.1 Installation Guide

For installing Anaconda Inc. you need to navigate to anaconda's official website given [here](#).

1. Choose your operating system (Windows/macOS/Linux) for installation from the top left panel.

2. Click on the link of the installer given on the page.
3. Press Download
4. Open executable file
5. Follow instructions in the set-up wizard
6. Run Anaconda Navigator

7 Python Libraries for Data Science

Python is an easy-to-learn, easy-to-debug, widely used, object-oriented and open-source language. Python has been built with extraordinary libraries for data science that are used by programmers every day in solving problems. The top 10 Python libraries for data science include:

1. NumPy
2. Pandas
3. SciPy
4. TensorFlow
5. Matplotlib
6. Keras
7. SciKit-Learn
8. PyTorch
9. Scrapy
10. BeautifulSoup

7.1 Numpy

NumPy (Numerical Python) is the fundamental package for numerical computation in Python; it contains a powerful N-dimensional array object. This is the foundation on which almost all the power of Python's data science toolkit is built, and learning NumPy is the first step on any Python data scientist's journey. NumPy also addresses the slowness problem partly by providing these multidimensional arrays as well as providing functions and operators that operate efficiently on these arrays. Here are the top four benefits that NumPy can bring to your code:

1. **More speed:** NumPy uses algorithms written in C that complete in nanoseconds rather than seconds.
2. **Fewer loops:** NumPy helps you to reduce loops and keep from getting tangled up in iteration indices.
3. **Clearer code:** Without loops, your code will look more like the equations you're trying to calculate.
4. **Better quality:** There are thousands of contributors working to keep NumPy fast, friendly, and bug free.

If you are familiar with MATLAB, you can find some interesting stuff about NumPy [here](#).

7.1.1 NumPy Installation

- **Install NumPy with pip**

To install NumPy with **pip**, bring up a terminal window and type:

```
$ pip install numpy
```

This command installs NumPy in the current working Python environment.

- **Install Numpy in Anaconda**

For simple installation via Anaconda, you can follow the instructions given [here](#).

Import Convention

To use numpy in the program we need to import the module. Generally, numpy package is defined as **np** of abbreviation for convenience. But you can import it using anything you want. The recommended convention to import numpy is:

```
>>>import numpy as np
```

7.1.2 Arrays

A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the *rank* of the array; the *shape* of an array is a tuple of integers giving the size of the array along each dimension.

Creating Arrays

```
>>> a = np.array([1,2,3])
>>> b = np.array([(1.5,2,3), (4,5,6)], dtype = float)
>>> c = np.array([[(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)]],
                  dtype = float)
```

Figure 1 Array Creation

We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

```
import numpy as np

a=np.array([1,2,3]) # Create a rank 1 array
```

Numpy also provides many functions to create arrays:

```
import numpy as np

a=np.zeros((2,2)) # Create an array of all zeros
print(a) # Prints "[[ 0.  0.]
          [ 0.  0.]]"

b=np.ones((1,2)) # Create an array of all ones
```

```

print(b) # Prints "[[ 1.  1.]]"

c=np.full((2,2),7) # Create a constant array
print(c) # Prints "[[ 7.  7.]
#          [ 7.  7.]]"

d=np.eye(2) # Create a 2x2 identity matrix
print(d) # Prints "[[ 1.  0.]
#          [ 0.  1.]]"

e=np.random.random((2,2)) # Create an array filled with random
values
print(e) # Might print "[[ 0.91940167 0.08143941]
#0.68744134  0.87236687]]"

```

Initial Placeholders	
>>> np.zeros((3,4))	Create an array of zeros
>>> np.ones((2,3,4),dtype=np.int16)	Create an array of ones
>>> d = np.arange(10,25,5)	Create an array of evenly spaced values (step value)
>>> np.linspace(0,2,9)	Create an array of evenly spaced values (number of samples)
>>> e = np.full((2,2),7)	Create a constant array
>>> f = np.eye(2)	Create a 2X2 identity matrix
>>> np.random.random((2,2))	Create an array with random values
>>> np.empty((3,2))	Create an empty array

Figure 2 Numpy Functions

For learning more ways of array creation check [this](#) out.

7.1.3 Attributes of Array Object

`ndarray.shape`: The dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with n rows and m columns, shape will be `(n, m)`.

`ndarray.ndim`: The number of axes (dimensions) of the array.

`ndarray.dtype`: Retrieves the data type of array.

`ndarray.itemsize`: The size in bytes of each element of the array.

`ndarray.size`: The total number of elements of the array. This is equal to the product of the elements of shape.

```

import numpy as np

a=np.array([1,2,3])# Create a rank 1 array
print(type(a))# Prints "<class 'numpy.ndarray'>"
print(a.shape)# Prints "(3,)"
print(a[0],a[1],a[2])# Prints "1 2 3"
a[0]=5# Change an element of the array
print(a)# Prints "[5, 2, 3]"
print(a.ndim)          #Prints 1
print(a.size)          #Prints 3

b=np.array([[1,2,3],[4,5,6]])# Create a rank 2 array
print(b.shape)# Prints "(2, 3)"
print(b[0,0],b[0,1],b[1,0])# Prints "1 2 4"

```

7.1.4 Array Slicing and Indexing

Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

```

import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2
rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b=a[:2,1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0,1])# Prints "2"
b[0,0]=77# b[0, 0] is the same piece of data as a[0, 1]
print(a[0,1])# Prints "77"

```

```

a = np.arange(4)**3 # create array a
a[2]                # member of a array in 2nd position
a[::-1]             # reversed a
a[0:4,1]            # each row in the second column of b
a[1,...]             # same as a[1,:,:] or a[1]
a[a>5]              # a with values greater than 5
x = a[0:4]           # assign x with 4 values of a
x[:]=99              # change the values of x to 99 which will change
the 4 values of a also.

```

Figure 3 Indexing and Slicing Techniques

```

>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 55],
       [54, 55]])

>>> a[:, 2]
a([2, 12, 22, 32, 42, 52])

>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])

```

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

Figure 4 Indexing and slicing illustration using NumPy Array

7.1.5 Basic Array Operations

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

```

Import numpy as np

x=np.array([[1,2],[3,4]],dtype=np.float64)
y=np.array([[5,6],[7,8]],dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x+y)
print(np.add(x,y))

# Elementwise difference; both produce the array
# [[-4.0 -4.0]

```

```

# [-4.0 -4.0]]
print(x-y)
print(np.subtract(x,y))

# Elementwise product; both produce the array
# [[ 5.0 12.0]
# [21.0 32.0]]
print(x*y)
print(np.multiply(x,y))

# Elementwise division; both produce the array
# [[ 0.2          0.33333333]
# [ 0.42857143  0.5          ]]
print(x/y)
print(np.divide(x,y))

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
# [ 1.73205081  2.          ]]
print(np.sqrt(x))

```

```

A = np.array([[1,1],[0,1]])
B = np.array([[2,0],[3,4]])
A+B          #addition of two array
np.add(A,B)  #addition of two array
A * B        # elementwise product
A @ B        # matrix product
A.dot(B)     # another matrix product
B.T          #Transpose of B array
A.flatten()  #form 1-d array
B < 3        #Boolean of Matrix B. True for elements less than 3
A.sum()      # sum of all elements of A
A.sum(axis=0) # sum of each column
A.sum(axis=1) # sum of each row
A.cumsum(axis=1) # cumulative sum along each row
A.min()      # min value of all elements
A.max()      # max value of all elements
np.exp(B)    # exponential
np.sqrt(B)   # square root
A.argmin()   #position of min value of elements
A.argmax()   #position of max value of elements
A[1,1]       #member of a array in (1,1) position

```

Figure 5 Array operations

7.1.6 Datatypes

Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric

datatypes that you can use to construct arrays. Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype.

If you want to know the data type of an array, you can query the attributes of *dtype*. An object describing the type of the elements in the array. One can create or specify *dtype* using standard Python types.

Additionally, *numpy* provides types of its own. *numpy.int32*, *numpy.int16*, and *numpy.float64* are some examples.

Data Types	
>>> np.int64	Signed 64-bit integer types
>>> np.float32	Standard double-precision floating point
>>> np.complex	Complex numbers represented by 128 floats
>>> np.bool	Boolean type storing TRUE and FALSE values
>>> np.object	Python object type
>>> np.string_	Fixed-length string type
>>> np.unicode_	Fixed-length unicode type

Figure 6 Data Types

Here is an example:

```
import numpy as np

x=np.array([1,2]) # Let numpy choose the datatype
print(x.dtype) # Prints "int64"

x=np.array([1.0,2.0]) # Let numpy choose the datatype
print(x.dtype) # Prints "float64"

x=np.array([1,2],dtype=np.int64) # Force a particular datatype
print(x.dtype) # Prints "int64"
```

7.1.7 Input/Output Operations

I/O	
Saving & Loading On Disk	
<pre>>>> np.save('my_array', a) >>> np.savez('array.npz', a, b) >>> np.load('my_array.npy')</pre>	
Saving & Loading Text Files	
<pre>>>> np.loadtxt("myfile.txt") >>> np.genfromtxt("my_file.csv", delimiter=',') >>> np.savetxt("myarray.txt", a, delimiter=" ")</pre>	

Figure 7 I/O operations

7.2 Pandas

The Pandas library is one of the most preferred tools for data scientists to do data manipulation and analysis. It can work with data from a wide variety of sources. Pandas is suited for many different kinds of data: tabular data, time-series data, arbitrary matrix data with row and column labels, and any other form of observational/statistical data sets.

Features:

1. Enables you to create your own function and run it across a series of data
2. Contains high-level data structures and manipulation tools

7.2.1 Pandas Installation

- **Installation via Terminal**

To install pandas in your system you can use this command `pip install pandas` OR `conda install pandas`.

- **Installation in Anaconda**

Pandas or any other packages can be installed in Anaconda environment by following the tutorial [here](#).

Import Convention

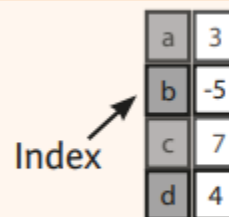
```
>>>importpandasaspd
```

7.2.2 Series

Pandas series works the same way both in list and numpy array as well as dictionary. To make series in pandas we need to use `pd.Series(data, index)` format where `data` are input data and `index` are selected index for data. A one-dimensional labeled array capable of holding any data type:

Series

A one-dimensional labeled array capable of holding any data type



a	3
b	-5
c	7
d	4

```
>>> s = pd.Series([3, -5, 7, 4], index=['a', 'b', 'c', 'd'])
```

```
>>> s = pd.Series([3, -5, 7, 4], index=['a', 'b', 'c', 'd'])
```

A	3
---	---

B	5
C	7
D	4

Pandas series alongwith numPy array:

```
import numpy as np          #importing numpy
import pandas as pd        #importing pandas
arr=np.array([1,3,5,7,9])  #create arr array
s2=pd.Series(arr)          #create pandas series s2
print(s2)                  #print s2
print(type(s2))            #print type of s2
```

Output:

```
0    1
1    3
2    5
3    7
4    9
dtype: int64
<class 'pandas.core.series.Series'>
```

7.2.3 Dataframe

Pandas DataFrame is a way to store data in rectangular grids that can easily be overviewed. It's like a tabular data structure with labeled axes (rows and columns). The default format of a DataFrame would be `pd.DataFrame(data, index, column)`. You need to mention the data, index and columns value to generate a DataFrame. Data should be at least *two-dimensional*, *index* will be the row name and *columns* values for the columns. In general, you could say that the Pandas DataFrame consists of three main components: the data, the index, and the columns.

The DataFrame can contain data that is:

1. Pandas DataFrame
2. Pandas Series: An example of a Series object is one column from a DataFrame.
3. a NumPy ndarray, which can be a record or structured
4. a two-dimensional ndarray
5. lists, dictionaries or Series.

DataFrame

Columns →

	Country	Capital	Population
0	Belgium	Brussels	11190846
1	India	New Delhi	1303171035
2	Brazil	Brasília	207847528

Index →

A two-dimensional labeled data structure with columns of potentially different types

```
>>> data = {'Country': ['Belgium', 'India', 'Brazil'],
'Capital': ['Brussels', 'New Delhi', 'Brasilia'],
'Population': [11190846, 1303171035, 207847528]}
>>> df = pd.DataFrame(data, columns=['Country', 'Capital', 'Population'])
```

To get subset of a DataFrame:

```
>>> df[1:]
Country    Capital    Population
1  India      New Delhi  1303171035
2  Brazil    Brasilia   207847528
```

```
import pandas as pd
s2 = pd.Series([10, 20, 30])
print(s2)
print(type(s2))
s3=pd.DataFrame([[1,2],[3,4]],columns=['A','B'], index = ['C', 'D'])
print(s3)
print(type(s3))
```

```
0    10
1    20
2    30
dtype: int64
<class 'pandas.core.series.Series'>
   A  B
C  1  2
D  3  4
<class 'pandas.core.frame.DataFrame'>
```

Figure 8 Example of Pandas Series and DataFrame

7.2.4 Indexing

- `.loc[]` works on labels of your index. This means that if you give in `loc[2]`, you look for the values of your DataFrame that have an index labeled 2.
- `.iloc[]` works on the positions in your index. This means that if you give in `iloc[2]`, you look for the values of your DataFrame that are at index '2'.

7.2.5 Input Output Operations

Read and Write to CSV

```
#Read
pd.read_csv('file.csv', header=None, nrows=5)
```

```
#Write
df.to_csv('myDataFrame.csv')

#Write and download CSV file
from google.colab import files
df.to_csv('myDataFrame.csv', index = False, header=True)
files.download('myDataFrame.csv')
```

7.2.6 Other Operations

- Set index of Series s to 6

```
>>> s['a'] = 6
```

- Sort by the values along an axis

```
>>> df.sort_values(by='Country')
```

- Assign ranks to entries

```
>>> df.rank()
```

7.2.7 Retrieving Series/DataFrame Information

- Basic Information (rows, columns)

```
>>> df.shape
```

- Describe index

```
>>> df.index
```

- Describe DataFrame columns

```
>>> df.columns
```

- Info on DataFrame

```
>>> df.info()
```

- Number of non-NA values

```
>>> df.count()
```

- Sum of values

```
>>> df.sum()
```

- Cumulative sum of values

```
>>> df.cumsum()
```

- Minimum/maximum values

```
>>> df.min()/df.max()
```

- Minimum/Maximum index value

```
>>> df.idxmin()/df.idxmax()
```

7.2.8 Application of Functions

```
>>> f = lambda x: x*2
```

Apply function

```
>>> df.apply(f)
```

8 Dataset Handling

8.1 Significance of Data

A data set is a set or collection of data. This set is normally presented in a tabular pattern. Every column describes a particular variable. And each row corresponds to a given member of the data set, as per the given question. The data are essentially organized to a certain model that helps to process the needed information. This set of data is any permanently saved collection of information that usually contains either case-level, gathered data, or statistical guidance level data.

In Machine Learning projects, it is impossible for an “AI” to learn without data. During an AI development, we always rely on data. From training, tuning, model selection to testing, we use three different data sets: the training set, the validation set and the testing set.

- **Training data set:** The training data set is the one used to train an algorithm to understand how to apply concepts such as neural networks, to learn and produce results. It is the actual **data set** used to train the model for performing various actions. It includes both input data and the expected output.
- **Test data set:** The test data set is used to evaluate how well your algorithm was trained with the training data set. In AI projects, we can’t use the training data set in the testing

stage because the algorithm will already know in advance the expected output which is not our goal.

8.2 Types of Data Sets

In Statistics, we have different types of data sets available for different types of information. They are:

- Numerical data sets
- Bivariate data sets
- Multivariate data sets
- Categorical data sets
- Correlation data sets

Let us discuss all these data sets with examples.

8.2.1 Numerical Data Sets

A set of all numerical data. It deals only with numbers. Some of the examples are;

- Weight and height of a person
- The count of RBC in a medical report
- Number of pages present in a book

8.2.2 Bivariate Data Sets

A data set that has two variables is called a Bi-variate data set. It deals with the relationship between the two variables.

Example: To find the percentage score and age of the students in a class. Score and age can be considered as two variables.

8.2.3 Multivariate Data Sets

A data set with multiple variables.

Example: If we have to measure the length, width, height, volume of a rectangular box, we have to use multiple variables to distinguish between those entities.

8.2.4 Categorical Data Sets

Categorical data sets represent features or characteristics of a person or an object.

Example:

- A person's gender (male or female)
- Marital status (married/unmarried)

8.2.5 Correlation Data Sets

The set of values that demonstrate some relationship with each other indicates correlation data sets. Here the values are found to be dependent on each other.

Example: A tall person is considered to be heavier than a short person. So here the weight and height variables are dependent on each other.

8.3 Mean, Median, Mode and Range of Data-Sets

The mean, median and mode along with range are the major topics in Statistics. Let us get through with respect to data-sets here.

Mean of a data-set is the average of all the observations present in the table. It is the ratio of the sum of observations to the total number of elements present in the data-set. The formula of mean is given by;

$$\text{Mean} = \text{Sum of Observations} / \text{Total Number of Elements in Data Set}$$

Median of a data-set is the middle value of the collection of data when arranged in ascending order and descending order.

Mode of a data-set is the variable or number or value which is repeated maximum number of times in the set.

Range of a data set is the difference between the maximum value and minimum value.

$$\text{Range} = \text{Maximum Value} - \text{Minimum Value}$$

Example:

Find the mean, mode, median and range of the given data set.

{2, 4, 6, 8, 2, 10, 12}

Solution:

Given, {2, 4, 6, 8, 2, 10, 12} is a set of data.

$$\text{Mean} = 2+4+6+8+2+10+12/7 = 44/7$$

To find median we have to first arrange the given data in ascending or descending order

So, {2,2,4,6,8,10,12}. Thus,

$$\text{Median} = 6$$

$$\text{Mode} = 2$$

$$\text{Range} = 12-2 = 10$$

8.4 Types of Attributes

1. Nominal

Examples: ID numbers, eye color, zip codes

2. Ordinal

Examples: rankings (e.g., taste of potato chips on a scale from 1-10), grades, height in

{tall, medium, short}

3. **Interval**

Examples: calendar dates, temperatures in Celsius or Fahrenheit.

4. **Ratio**

Examples: temperature in Kelvin, length, time, counts

8.5 Properties of Dataset

Before performing any statistical analysis, it is essential to understand the nature of the data. We can use different Exploratory Data Analysis (EDA techniques), which helps to identify the properties of data, so that the appropriate statistical methods can be applied on the data. With the help of EDA techniques, we can check the following properties of the dataset.

- Centre of data
- Skewness of data
- Presence of outliers
- Correlation among the data

8.6 Dataset Repositories

Datasets for analysis can be downloaded from common repositories like:

1. Kaggle
2. UCI

Check out [this](#) link for a list of various data repositories.

8.7 Properties of Attributes in a dataset

8.7.1 Missing Values

Some entries can be missing because of the following reasons:

1. **Data Extraction:** It is possible that there are problems with extraction process. In such cases, we should double-check for correct data with data guardians. Errors at data extraction stage are typically easy to find and can be corrected easily as well.
2. **Data collection:** These errors occur at time of data collection and are harder to correct.

8.7.2 Outliers

Outlier is an observation that appears far away and diverges from an overall pattern in a sample. Let's take an example, we do customer profiling and find out that the average annual income of customers is \$1 lakh. But, there are two customers having annual income of \$4 and \$4.2 million. These two customers annual income is much higher than rest of the population. These two observations will be seen as Outliers.

Outliers can be due to the following reasons:

1. **Data Entry Errors:** Human errors such as errors caused during data collection, recording, or entry can cause outliers in data. For example: Annual income of a customer

is \$100,000. Accidentally, the data entry operator puts an additional zero in the figure. Now the income becomes \$1,000,000 which is 10 times higher. Evidently, this will be the outlier value when compared with rest of the population.

2. **Measurement Error:** It is the most common source of outliers. This is caused when the measurement instrument used turns out to be faulty. For example: There are 10 weighing machines. 9 of them are correct, 1 is faulty. Weight measured by people on the faulty machine will be higher / lower than the rest of people in the group. The weights measured on faulty machine can lead to outliers.
3. **Experimental Error:** Another cause of outliers is experimental error. For example: In a 100m sprint of 7 runners, one runner missed out on concentrating on the 'Go' call which caused him to start late. Hence, this caused the runner's run time to be more than other runners. His total run time can be an outlier.
4. **Data Processing Error:** Whenever we perform data mining, we extract data from multiple sources. It is possible that some manipulation or extraction errors may lead to outliers in the dataset.
5. **Sampling error:** For instance, we have to measure the height of athletes. By mistake, we include a few basketball players in the sample. This inclusion is likely to cause outliers in the dataset.

8.8 Iris Flower Dataset

The Iris flower data set or Fisher's Iris data set is a multivariate data set introduced by the British statistician. The dataset contains 150 observations of iris flowers. There are four columns of measurements of the flowers in centimeters. The fifth column is the species of the flower observed. All observed flowers belong to one of three species. The data set consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor).

To gain more insights about this dataset check [this link](#).

You can download this dataset from [here](#).

8.9 Load the Dataset

In order to load the dataset, run Anaconda Navigator.

1. Launch Jupyter Notebook
2. Click on "Upload" button as shown in fig. 10
3. Select the dataset file from your directory
4. Create a new Python3 notebook by selecting "New" on the Jupyter host page
5. Follow the instructions given in the code snippet provided next



Figure 9 Load Dataset in Anaconda

Open the newly created notebook and implement the code given below:

```
import pandas as pd
```

```
iris_dataset= pd.read_csv("iris.csv")
iris_dataset
```

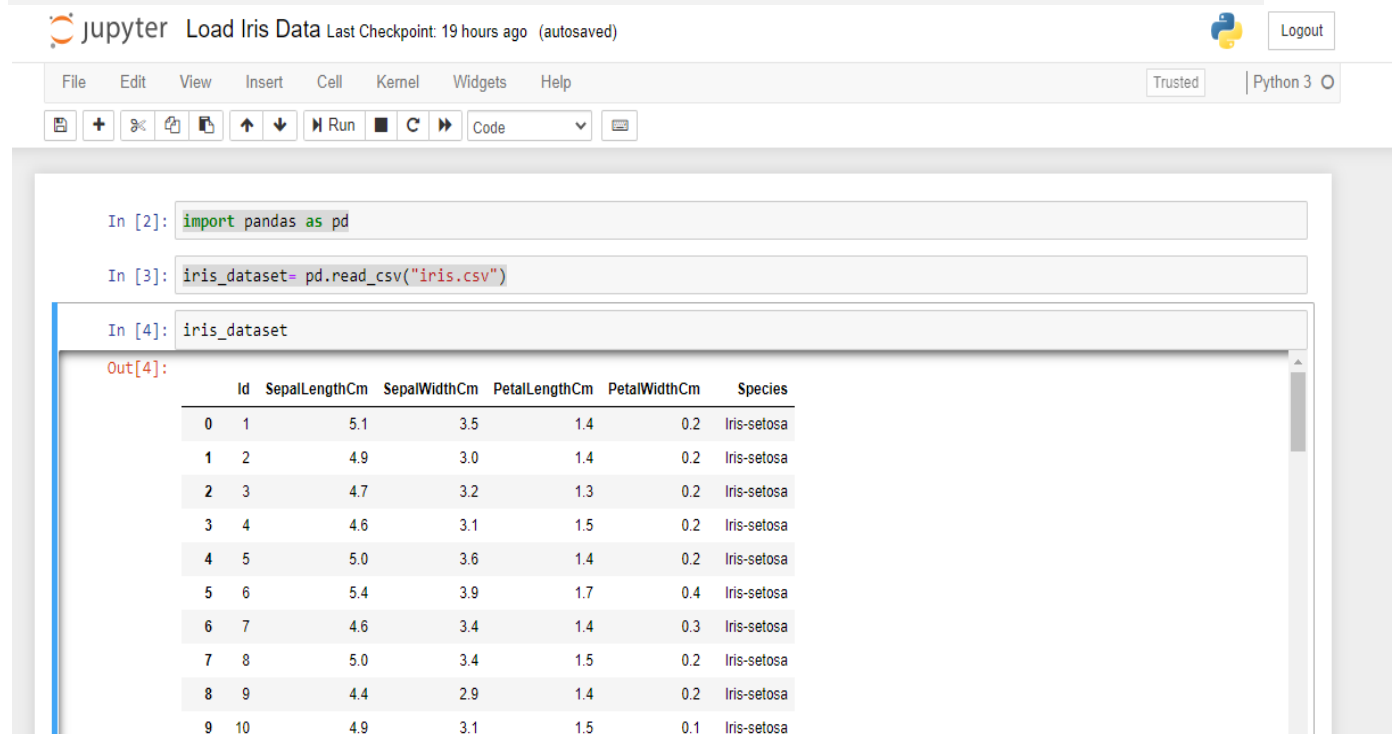


Figure 10 Code instructions

For more information about loading files, you can visit this [site](#).