

1Q.

Create, Read, Update and Delete.

2Q.

int, float and object are the data types used in telco_churn

3Q.

Because, if you are deleting the duplicates without any criteria or factors, so times we may lost some valuable information.

4Q.

Obviously, it effects the organization in an every corner of developments areas. As a data scientist If you give more access to data to me, I will show you the ways of development in many areas, and also improve the performance of organizations by going through large sets of data.

5Q.

Statsmodels, Requests, Scipy, and Sqlite 3 are the four libraries are not discussed in the class

One of the Python libraries is statsmodels. For many forms of analysis, such as statistical models based on time series, linear regression, and also hypothesis testing, we employed this stats models library.

SciPy: SciPy is a fascinating library that has a large number of collections for mathematical and numerical functions to carry out a variety of tasks, including interpolation, image processing, and single-digit processing.

And I like SciPy because I did a study on signal processing, which I found to be extremely interesting, and because utilizing it allows us to do jobs in multiple ways.

6Q.

Numpy, Scipy and Matplotlib are used to design the SciKit-Learn

These three libraries are the most powerful, thus using them allows us to carry out multiple tasks at once. tasks including mathematics, science, and machine learning algorithms are also included.

7Q.

Basically, the removing feature is an introduce bias for an data, the author believes the same and he also thinks that missing data as to be kept in separate feature and keeps the feature for the missing data as Not answered.

8Q.

Measure the relationship between two variable quantities.

#Vijay Raj Pathani 6/25/2023

Assignment 3 Hands-on

Complete the following three sections: 

- Python Data Structures: Series
- Python Data Structures: DataFrames
- Python Data Structures: NumPy Arrays

Python Data Structures: Series

Programming for Data Science with Python

Overview

Series is a **one-dimensional labeled NumPy array**

- capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.)
 - The axis labels are collectively referred to as the index.
 - All of the values have the same data type, similar to Numpy ndarrays

The basic method to create a series is to call: `**s = pandas.Series(data, index=index)`

Here "data" can be a ndarray, or a dictionary, or a scalar value, etc.

1. Create Series

1.1 Series Constructor

A pandas Series can be created with the following constructor: **pandas.Series(data, Index, dtype, copy)**

data: constants, ndarray, list, dictionary, etc **index:** Index values must be unique and hashable, same length as data.

- index: passed parameter is a list of axis label
- Default: `np.arange(n)` if no index is passed.

dtype: dtype is for data type. If None, data type will be inferred.

- copy: Copy data. Default: False

1.2 Create empty series

Run the following code block:

```
In [1]: #filter warnings  
import warnings  
warnings.filterwarnings("ignore")  
  
# Create an empty series  
import pandas as pd  
s=pd.Series()  
print(s)
```

```
Series([], dtype: float64)
```

1.3 Create a series from an ndarray

If data is an ndarray, then index passed must be of the same length.

- If no index is passed, then by default index will be range(n) where n is array length, i.e., [0, 1, 2, 3 ... range (len(array))-1].

Run the following 2 code blocks:

```
In [2]: # Example 1: Create a series from an ndarray  
import pandas as pd  
import numpy as np  
# Array is created from a List  
data = np.array(['a', 'b', 'c', 'd'])  
# A series is created from the array with the default index  
s = pd.Series(data)  
print(s)
```

```
0    a  
1    b  
2    c  
3    d  
dtype: object
```

```
In [3]: #Example 2: Create a series from an ndarray  
import pandas as pd  
import numpy as np  
#Array is created from a List  
data = np.array(['a', 'b', 'c', 'd'])  
# A series is created from the array with specific indices  
s = pd.Series(data, index=[100,101,102,103])  
print(s)
```

```
100    a  
101    b  
102    c  
103    d  
dtype: object
```

PRACTICE

- Change the index to 10,11,12,13

```
In [4]: #To Do add your code here  
#Example 2: Create a series from an ndarray  
import pandas as pd  
import numpy as np  
#Array is created from a List  
data = np.array(['a', 'b', 'c', 'd'])  
# A series is created from the array with specific indices  
s = pd.Series(data, index=[10,11,12,13])  
print(s)
```

```
10     a  
11     b  
12     c  
13     d  
dtype: object
```

1.4 Create a series from a dictionary

A dict can be passed as input.

- If no index is specified, then the dictionary keys are taken in a sorted order to construct the index.
- If index is passed, the values in data corresponding to the labels in the index will be pulled out.

Run the following 2 code blocks:

```
In [5]: ▶ # Create a series from a dictionary
import pandas as pd
import numpy as np

# Declare a dictionary with keys: 'a', 'b', 'c'
aDict = {'a': 0., 'b' : 1., 'c' :2.}

#Create a series from this dictionary
s = pd.Series(aDict)
print(s)
```

```
a    0.0
b    1.0
c    2.0
dtype: float64
```

```
In [6]: ▶ # Create a series from a dictionary

import pandas as pd
import numpy as np

# Declare a dictionary with keys: 'a', 'b', 'c'
data = {'a': 0., 'b' : 1., 'c' :2.}

# Create a series from this dictionary with specific indices
# The dictionary has only three items
s = pd.Series(data, index=['b','c','d','a'])
print(s)
```

```
b    1.0
c    2.0
d    NaN
a    0.0
dtype: float64
```

1.5 Create a series from scalar values

If data is a **scalar** value, an **index must be provided**.

- The value will be repeated to match the length of index.

Run the following code block:

```
In [7]: # Create a series from scalar values  
import pandas as pd  
import numpy as np  
# Create a series  
s = pd.Series(5, index=[0, 1, 2, 3])  
print(s)
```

```
0    5  
1    5  
2    5  
3    5  
dtype: int64
```

1.6 Accessing Data from Series with Position

Data in the series can be accessed similar to that ndarray.

Run the following code block:

```
In [8]: import pandas as pd  
s = pd.Series([1,2,3,4,5], index = ['a','b','c','d','e'])  
  
#retrieve the first element  
print(s[0])
```

```
1
```

```
In [9]: import pandas as pd  
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])  
  
# Retrieve the first 3 elements: from 0 - 3, not including 3  
# i.e., retrieve 0,1,2  
print(s[:3])
```

```
a    1  
b    2  
c    3  
dtype: int64
```

```
In [10]: ▶ import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

# Retrieve the last 3 elements:
print(s[-3:])
```

```
c    3
d    4
e    5
dtype: int64
```

```
In [11]: ▶ import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

# Retrieve a single element at a specific index
print(s['a'])
```

```
1
```

```
In [12]: ▶ import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

# Retrieve multiple elements using a list of index label values
print(s[['a','c','d']])
```

```
a    1
c    3
d    4
dtype: int64
```

Python Pandas: Dataframes

Programming for Data Science with Python

Overview

What is NDFrame?

N-dimensional analogue of DataFrame. Store multi-dimensional in a size-mutable, labeled data structure.

What's a DataFrame?

`class DataFrame(NDFrame)`: **Two-dimensional** size-mutable, potentially heterogeneous tabular data structure with labeled axes (**rows and columns**). Arithmetic operations align on both row and column labels. Can be thought of as a **dict-like** container for **Series** objects.

- DataFrame is a subclass (i.e., special case) of NDFrame.
- In Pandas programs generally, DataFrame is used a lot and NDFrame is used rarely.
- In fact, Pandas has Series for 1D, DataFrame for 2D, and for most people that's the end (even though half of Pandas' name is for Panel which Pandas also has, but most people do not use).

FUN FACT: There is/was even a 4D thing in Pandas, (but truly no one uses it (this being the internet, someone will now appear to say they do!).

For higher dimensions than two or maybe three, some people have shifted their efforts to **xarray**.

That's probably where it's at if your ambitions cannot be contained in 2D.

1.1 Definition

A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.

	Column Index (df.columns)					
Row of Index (df.index)	Series of data	Series of data	Series of data	Series of data	Series of data	Series of data

1.2 pandas.DataFrame: Attributes

`T` → Transpose index and columns

`at` → Fast label-based scalar accessor

`axes` → Return a list with the row axis labels and column axis labels as the only members.

`blocks` → Internal property, property synonym for `as_blocks()`

`dtypes` → Return the dtypes in this object.

`empty` → True if NDFrame is entirely empty [no items], meaning any of the axes are of length 0.

`ftypes` → Return the ftypes (indication of sparse/dense and dtype) in this object.

iat → Fast integer location scalar accessor.

iloc → Purely integer-location based indexing for selection by position.

ix → A primarily label-location based indexer, with integer position fallback.

loc → Purely label-location based indexer for selection by label.

ndim → Number of axes/array dimensions.

shape → Return a tuple representing the dimensionality of the DataFrame.

size → number of elements in the NDFrame

style → Property returning a Styler object containing methods for building a styled HTML representation for the DataFrame.

values → Numpy representation of NDFrame.

2. Create Dataframes

2.1 DataFrame Constructor

pandas.DataFrame(data, index, columns, dtype, copy)

Data:

- can be ndarray, series, map, lists, dict, constants, and another DataFrame.

Index:

- For the row labels, the index to be used for the resulting frame is Optional Default np.arange(n), if no index is passed.

Columns:

- For column labels, the Optional Default syntax is - np.arange(n). This is only true if no index is passed.

dtype:

- Data type of each column

Copy:

- This command (or whatever it is) is used for copying of the data.

Default: False

In [13]:  *# Create an empty dataframe*

```
import pandas as pd

df = pd.DataFrame()
print(df)
```

```
Empty DataFrame
Columns: []
Index: []
```

2.3 Create a Dataframe from lists Run the following 2 code blocks:

In [14]:  *# Create a dataframe from a List*

```
import pandas as pd

# Declare a List
alist = [1,2,3,4,5]

# Create a dataframe from the List
df = pd.DataFrame(alist)

print(df)
```

```
0
0  1
1  2
2  3
3  4
4  5
```

In [15]:  *# Create a dataframe from a List of Lists*

```
import pandas as pd

# Declare a List of Lists - each List element has two elements [string, nu
alistOfLists = [['Alex',10],['Bob',12],['Clarke',13]]

# Create a dataframe from this List, naming the columns as 'Name' and 'Age
df = pd.DataFrame(alistOfLists, columns=['Name', 'Age'])

print(df)
```

```
   Name  Age
0  Alex   10
1   Bob   12
2 Clarke   13
```

```
In [16]: ▶ # Create a dataframe from a List of Lists and set the data type
import pandas as pd

# Declare a List of Lists - each List element has two elements [string, number]
aListOfLists = [['Alex',10], ['Bob',12], ['Clarke',13]]

# Create a dataframe from this List, naming the columns as 'Name' and 'Age'
df = pd.DataFrame(aListOfLists, columns=['Name', 'Age'], dtype=float)

print(df)
```

	Name	Age
0	Alex	10.0
1	Bob	12.0
2	Clarke	13.0

PRACTICE

- Change the data type to complex and the names to Cos, Sin, and Tangent

```
In [17]: ▶ import pandas as pd

aListOfLists = [['Cos',10], ['Sin',12], ['Tan',13]]

df = pd.DataFrame(aListOfLists, columns=['Name', 'angle'], dtype=complex)

print(df)
```

	Name	angle
0	Cos	10.0+0.0j
1	Sin	12.0+0.0j
2	Tan	13.0+0.0j

2.4 Create dataframes from dictionaries of ndarray/lists

- All the ndarrays must be of same length.
- If index is passed, then the length of the index should equal to the length of the arrays.
- If no index is passed, then by default, index will be range(n), where n is the array length.

Run the following 3 code blocks:

In [1]: ▶ *# Create a dataframe from a dictionary without specified indices*

```
import pandas as pd

# Declare a dictionary that has two key-value pairs
# One key is "Name" that has its value = a list of strings
# Another key is "Age" that has its value = a list of integers

aDict = {'Name': ['Tom', 'Jack', 'Steve', 'Ricky'], 'Age': [28, 34, 29, 42]}

# Create the dataframe from the dictionary
# VIP NOTES: Automatically adding the indices for the rows

df = pd.DataFrame(aDict)
print(df)
```

	Name	Age
0	Tom	28
1	Jack	34
2	Steve	29
3	Ricky	42

In [2]: ▶ *# Create a dataframe from a dictionary with specified indices*

```
import pandas as pd

# Declare a dictionary that has two key-value pairs
# One key is "Name" that has its value = a List of strings
# Another key is "Age" that has its value = a list of integers

aDict = { ' Name' : ['Tom', 'Jack', 'Steve', 'Ricky'], 'Age' : [28, 34, 29, 42]}

# Create the dataframe from the dictionary
# VIP NOTES: Specifying the indices for the rows

df = pd.DataFrame(aDict, index=['rank1', 'rank2', 'rank3', 'rank4'])
print(df)
```

	Name	Age
rank1	Tom	28
rank2	Jack	34
rank3	Steve	29
rank4	Ricky	42

```
In [3]: # Create a dataframe from a dictionary of series

import pandas as pd

# Declare a dictionary of 2 series named 'one' and 'two'

aDictOfSeries = {'one': pd.Series([1, 2, 3], index=['a', 'b', 'c']),
                  'two': pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

# Create a dataframe from this dictionary
# VIP NOTES: Each column of a dataframe is a series

df = pd.DataFrame(aDictOfSeries)
print(df)
```

	one	two
a	1.0	1
b	2.0	2
c	3.0	3
d	NaN	4

2.5 Access Dataframe Columns

Run the following code block:

```
In [23]: # Access a dataframe columns

import pandas as pd

# Declare a dictionary of 2 series named 'one' and 'two'
aDictOfSeries = {'one': pd.Series([1, 2, 3], index= ['a', 'b', 'c']),
                  'two': pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

# Create a dataframe from this dictionary
# VIP NOTES: Each column of a dataframe is a series

df = pd.DataFrame(aDictOfSeries)

# Access the column 'one' and print it out
# HOW TO access a column: Using its label as a column index


print(df['one'])
```

a	1.0
b	2.0
c	3.0
d	NaN

Name: one, dtype: float64

2.6 Add Columns into a Dataframe

Run the following 2 code blocks:

In [29]:  *# Add columns into a dataframe*

```
import pandas as pd

# Declare a dictionary of 2 series named 'one' and 'two'
aDictOfSeries = {'one': pd.Series([1, 2, 3], index=['a', 'b', 'c']), 'two': pd.Series([1, 2, 3], index=['a', 'b', 'c'])}


# Create a dataframe from this dictionary
# VIP NOTES: Each column of a dataframe is a series

df = pd.DataFrame(aDictOfSeries)

# Adding a new column to an existing DataFrame object with column label by adding a new series into the dataframe as a new column: 'three'
# First, creating a new series
# Then, assign the new series into the new column

df['three'] = pd.Series([10, 20, 30], index=['a', 'b', 'c'])
print(df)
```

	one	two	three
a	1.0	1	10.0
b	2.0	2	20.0
c	3.0	3	30.0
d	NaN	4	NaN

In [30]:  *# Adding a new column using the existing columns in Data Frame*

```
df['four'] = df['one'] + df['three']
print('\n')
print(df)
```

	one	two	three	four
a	1.0	1	10.0	11.0
b	2.0	2	20.0	22.0
c	3.0	3	30.0	33.0
d	NaN	4	NaN	NaN

PRACTICE

- Add another column using the existing columns that is a copy of column two

In [4]:



```
df['five']=df['two']  
print(df)
```

	one	two	five
a	1.0	1	1
b	2.0	2	2
c	3.0	3	3
d	NaN	4	4

2.7 Delete/Pop/Remove a Column from a Dataframe

Run the following code block:

```
In [5]: # Delete a column using del function
import pandas as pd
# Declare a dictionary of 2 series named 'one' and 'two'
aDictOfSeries = {'one': pd.Series ([1, 2, 3],
index = ['a', 'b', 'c']), 'two': pd.Series([1, 2, 3, 4],
index = ['a', 'b', 'c', 'd']), 'three': pd.Series([10,20,30],
index = ['a', 'b', 'c'])}
# Create a dataframe from this dictionary
# VIP NOTE: Each column of a dataframe is a series
df = pd.DataFrame(aDictOfSeries)
print(df)
print('\n')
# using del function to delete/remove the first column
del(df['one'])
print(df)
print('\n')
# using pop function to delete another column " 'two'
# Deleting another column using PDP function
df.pop('two')
print(df)
```

	one	two	three
a	1.0	1	10.0
b	2.0	2	20.0
c	3.0	3	30.0
d	NaN	4	NaN

	two	three
a	1	10.0
b	2	20.0
c	3	30.0
d	4	NaN

	three
a	10.0
b	20.0
c	30.0
d	NaN

2.8 Access Rows of a Dataframe → loc & iloc

Run the following 3 code blocks:

```
In [6]: # Access rows of a dataframe using Loc function
# Loc is a row index

import pandas as pd

aDictOfSeries = {'one': pd.Series ([1, 2, 3], index=['a', 'b', 'c']),
                  'two':pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(aDictOfSeries)

# Access the row with index='b' and print the row
print(df.loc['b'])
```

one 2.0
two 2.0
Name: b, dtype: float64

```
In [7]: # Access rows of a dataframe using iLoc (integer Location/row index) function

import pandas as pd

aDictOfSeries = {'one': pd.Series ([1, 2, 3], index= ['a', 'b', 'c']),
                  'two': pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(aDictOfSeries)

# Access the row with index= '2' and print the row
print(df.iloc[2])
```

one 3.0
two 3.0
Name: c, dtype: float64

```
In [8]: # Access a group of rows using the ':' operator

import pandas as pd

aDictOfSeries = {'one': pd.Series ([1, 2, 3], index= ['a', 'b', 'c']),
                  'two': pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(aDictOfSeries)

# Access all the rows with indices 2, 3 and print them
print(df[2:4])
```

	one	two
c	3.0	3
d	NaN	4

2.9 Delete/Remove Rows from a Dataframe

Run the following code block:

In [11]:  *# Remove rows from a dataframe using the drop() function*

```
import pandas as pd

df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a', 'b'])
df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a', 'b'])

df = df.append(df2)

# Drop rows with Label 0
df = df.drop(0)

print(df)
```

```
   a  b
1  3  4
1  7  8
```

```
/tmp/ipykernel_713/3515169852.py:8: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  df = df.append(df2)
```

3. Load Data into DataFrame

To read data into a dataframe, we use this command:

pd.read_file_type(file_name)

Where file_type can be csv, excel, etc.

For example, for CSV files, the command to read a csv file: pd.read_csv()

Example:

You will have your datasets already loaded but I want you to have an example in case you challenge yourself and work on your own.

```
import pandas as pd
```

Reads the flights data set and create the dataframe flights

```
df_flights = pd.read_csv  
( 'C:/DATA/DROPBOX/Dropbox/DATA_APPLS/DATASETS/flights_2  
.csv' )
```

The command below would print out the 1st 5 rows of the dataframe that was created.

```
df_flights.head(5)
```

NumPy Arrays

Programming for Data Science with Python

1. Introduction

NumPy is the fundamental package for **scientific computing** in Python.

It is a **Python library** that provides:

- a multidimensional array object
- various derived objects (such as masked arrays and matrices)
- an assortment of routines for fast operations on arrays (including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.)

At the **core** of the **NumPy** package, is the ndarray object.

- This encapsulates **n-dimensional arrays** of homogeneous data types - with many operations being performed in compiled code for performance.

There are several **important differences** between **NumPy arrays** and the **standard Python sequences**:

- **NumPy arrays** have a **fixed size** at creation, unlike Python lists (which can grow dynamically).

- Changing the size of a ndarray will create a new array and delete the original.

The **elements** in a **NumPy array** are all required to be of the same data type, thus will be the same size in memory.

The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.

NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.

A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays.

In other words, in order to efficiently use much (perhaps even most) of today's scientific/mathematical Python-based software, just knowing how to use Python's built-in sequence types is insufficient - one also needs to know how to use NumPy arrays.

2. NumPy: Class ndarray

2.1 Overview

In NumPy, an array object represents a multidimensional, homogeneous array of fixed-size items.

An associated data-type object describes the format of each element in the array:

- its byte-order
- how many bytes it occupies in memory,
- whether it is an integer, a floating point number, or something else.

2.2 Fundamental Concepts

2.2.1 NumPy Arrays: Dimension & Axis

NumPy's main object is the ***homogeneous multidimensional array**.

It is a table of elements (usually numbers), all of the same type, indexed by a **tuple of positive integers**.

In NumPy, **dimensions** are called **axes**. The number of axes is rank.

EXAMPLE:

The coordinates of a point in 3D space [1, 2, 1] is an array of rank 1, because it has one axis.

- This axis has a length of 3.

This array, `[[1., 0., 0.], [0., 1., 2.]]` has rank 2, i.e., 2 dimensions (it is 2-dimensional).

- Let's see this array as a table of **2 rows and 3 columns**.
 - The first dimension (axis 0) has a length of 2 (2 rows), the second dimension (axis 1) has a length of 3 (3 columns).

A 2-dimensional array has two corresponding axes:

- the first running vertically downwards across rows (axis 0)
- the second running horizontally across columns (axis 1)

Many operation can take place along one of these axes. For example, we can sum each row of an array, in which case we operate along columns, or axis 1.

```
In [37]: import numpy as np
x = np.arange(12).reshape((3,4))
x
```

```
Out[37]: array([[ 0,  1,  2,  3],
               [ 4,  5,  6,  7],
               [ 8,  9, 10, 11]])
```

```
In [38]: # Sum up all the elements
x.sum()
```

```
Out[38]: 66
```

```
In [39]: # Sum up elements along the horizontal axis
x.sum(axis = 1)
```

```
Out[39]: array([ 6, 22, 38])
```

```
In [40]: # Sum up elements along the vertical axis
x.sum(axis = 0)
```

```
Out[40]: array([12, 15, 18, 21])
```

IMPORTANT NOTES: **axis= 0:** Actions occur vertically, i.e., moving along vertically actions on the rows, e.g., drop rows, add rows

IMPORTANT NOTES: **axis = 1:** Actions occur horizontally, i.e., moving along horizontally actions on the columns, e.g., add columns, drop columns

3. Numpy Arrays: Creation

3.1 Overview

NumPy arrays can be created in different ways, with default initial values or manually specified ones.

3.2 NumPy Arrays: Creation: Empty, Ones, Zeros, Full Arrays

empty(shape[, dtype, order]) Returns a new array of given shape and type, without initializing entries.

empty_like(a[, dtype, order, subok]) Returns a new array with the same shape and type as a given array.

eye(N[, M, k, dtype]) Returns a 2-D array with ones on the diagonal and zeros elsewhere.

identity(n[, dtype]) Returns the identity array.

ones(shape[, dtype, order]) Returns a new array of given shape and type, filled with ones.

ones_like(a[, dtype, order, subok]) Returns an array of ones with the same shape and type as a given array.

zeros(shape[, dtype, order]) Returns a new array of given shape and type, filled with zeros.

zeros_like(a[, dtype, order, subok]) Returns an array of zeros with the same shape and type as a given array.

full(shape, fill_value[, dtype, order]) Returns a new array of given shape and type, filled with fill_value.

full_like(a, fill_value[, dtype, order, subok]) Returns a full array with the same shape and type as a given array.

3.2.1 empty(shape[, dtype, order])

Returns a new array of given shape and type, without initializing entries.

Parameters:

shape: int or tuple of int → Shape of the empty array

dtype: data-type, optional → Desired output data-type.

order: {'C', 'F'}, optional

Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

Returns:

out: ndarray

In [41]: `import numpy as np`

```
# An empty 1D array: shape = 8 (1D, 8 elements)
np.empty(8)
```

Out[41]: array([4.9e-324, 1.5e-323, 2.5e-323, 3.5e-323, 9.9e-324, 2.0e-323, 3.0e-323, 4.0e-323])

In [42]: `import numpy as np`

```
# An empty 1D array of integers: shape = 8
np.empty(8, dtype=int)
```

Out[42]: array([0, 0, 1, 0, 0, 0, 1, 0])

In [43]: `import numpy as np`

```
# Store values in rows as "C" style
np.empty(8, dtype=int, order='C')
```

Out[43]: array([0, 1073741824, 0, 0, 0, 0, 1074790400, 0])

In [44]: `import numpy as np`

```
# An empty 1D array of integers: shape (a tuple)
np.empty([2, 3], dtype = int)
```

Out[44]: array([[1, 0, 2],
 [0, 3, 0]])

In [45]: `import numpy as np`

```
# an 1D array of strings
np.empty(8, dtype=str)
```

Out[45]: array(['', '', '', '', '', '', '', ''], dtype='<U1')

3.2.2 empty_like(a[, dtype, order, subok])

Returns a new array with the same shape and type as a given array.

Parameters:

a: array like → The shape and data-type of a define these same attributes of the returned array.

dtype: data-type, optional; → Overrides the data type of the result.

...subok: bool, optional

- If True, then the newly created array will use the sub-class type of 'a'
 - Otherwise it will be a base-class array. Defaults to True.

Returns:

out: ndarray

Array of uninitialized (arbitrary) data with the same shape and type as 'a'.

Run the following 2 code blocks:

```
In [46]:  ▶ import numpy as np

a= ([1,2,3], [4,5,6]) # a is array-like

np.empty_like(a)
```

```
Out[46]: array([[ -788977888,      32764, -788977856],
               [      32764, -788977824,      32764]])
```

```
In [47]:  ▶ import numpy as np

a= np .array([[1., 2., 3.],[4.,5.,6.]])

np.empty_like(a)
```

```
Out[47]: array([[10.,  0., 12.],
               [ 0., 13.,  0.]])
```

3.2.3 identity(n[, dtype])

Returns the identity array.

Run the following code block:

```
In [48]:  ▶ import numpy as np

np.identity(8, dtype=int)
```

```
Out[48]: array([[1, 0, 0, 0, 0, 0, 0, 0],
               [0, 1, 0, 0, 0, 0, 0, 0],
               [0, 0, 1, 0, 0, 0, 0, 0],
               [0, 0, 0, 1, 0, 0, 0, 0],
               [0, 0, 0, 0, 1, 0, 0, 0],
               [0, 0, 0, 0, 0, 1, 0, 0],
               [0, 0, 0, 0, 0, 0, 1, 0],
               [0, 0, 0, 0, 0, 0, 0, 1]])
```

3.2.4 eye(N[, M, k, dtype])

Returns a 2-D array with ones on the diagonal and zeros elsewhere.

Parameters:

N: int → Number of rows in the output.

M: int, optional → Number of columns in the output. If None, defaults to N.

k: int, optional → Index of the diagonal

0 (the default): → to the main diagonal; a positive value refers to an upper diagonal; and a negative value to a lower diagonal.

dtype: data-type, optional → Data-type of the returned array.

Returns:

I: ndarray of shape (N,M)

An array where all elements are equal to zero, except for the k-th diagonal, whose values are equal to one.

Run the following 2 code blocks:

```
In [49]: ▶ import numpy as np
          np.eye (8, dtype = int)
```

```
Out[49]: array([[1, 0, 0, 0, 0, 0, 0, 0],
                [0, 1, 0, 0, 0, 0, 0, 0],
                [0, 0, 1, 0, 0, 0, 0, 0],
                [0, 0, 0, 1, 0, 0, 0, 0],
                [0, 0, 0, 0, 1, 0, 0, 0],
                [0, 0, 0, 0, 0, 1, 0, 0],
                [0, 0, 0, 0, 0, 0, 1, 0],
                [0, 0, 0, 0, 0, 0, 0, 1]])
```

```
In [50]: ▶ import numpy as np
          np.eye (8, k=2)
```

```
Out[50]: array([[0., 0., 1., 0., 0., 0., 0., 0.],
                [0., 0., 0., 1., 0., 0., 0., 0.],
                [0., 0., 0., 0., 1., 0., 0., 0.],
                [0., 0., 0., 0., 0., 1., 0., 0.],
                [0., 0., 0., 0., 0., 0., 1., 0.],
                [0., 0., 0., 0., 0., 0., 0., 1.],
                [0., 0., 0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0., 0., 0.]])
```

3.2.5 ones(shape[, dtype, order])

Returns a new array of given shape and type, filled with ones.

Parameters:

shape: int or sequence of ints → Shape of the new array, e.g., (2, 3) or 2.

dtype: data-type, optional → The desired data-type for the array, e.g., numpy.int8. Default is numpy.float64.

order: {'C', 'F'}, optional → ...

Returns:

out: ndarray

Array of ones with the given shape, dtype, and order.

Run the following 2 code blocks:

```
In [51]:  ▶ import numpy as np
          np.ones(8)
```

```
Out[51]: array([1., 1., 1., 1., 1., 1., 1., 1.])
```

```
In [52]:  ▶ import numpy as np
          np.ones(8, dtype=int)
```

```
Out[52]: array([1, 1, 1, 1, 1, 1, 1, 1])
```

3.2.6 zeros(shape[, dtype, order])

Returns a new array of given shape and type, filled with zeros.

Run the following code block:

```
In [53]:  ▶ import numpy as np
          np.zeros(8)
```

```
Out[53]: array([0., 0., 0., 0., 0., 0., 0., 0.])
```

3.2.7 full(shape, fill_value[, dtype, order])

Returns a new array of given shape and type, filled with fill_value.

Run the following 2 code blocks:

```
In [54]:  ▶ import numpy as np
          np.full(8, 2)
```

```
Out[54]: array([2, 2, 2, 2, 2, 2, 2, 2])
```

```
In [55]:  import numpy as np  
          np.full(8, "Hello")
```

```
Out[55]: array(['Hello', 'Hello', 'Hello', 'Hello', 'Hello', 'Hello', 'Hello',  
               'Hello'], dtype='<U5')
```

3.2.8 arange([start,]stop, [step,]dtype=None)

Returns evenly spaced values within a given interval.

Values are generated within the half-open interval [start, stop) (in other words, the interval including start but excluding stop). For integer arguments the function is equivalent to the Python built-in range function, but returns an ndarray rather than a list. When using a non-integer step, such as 0.1, the results will often not be consistent. It is better to use linspace for these cases.

Parameters:

start: number, optional → Start of interval. The interval includes this value. The default start value is 0.

stop: number → End of interval. The interval does **not include** this value (except in some cases where step is not an integer and floating point round-off affects the length of out).

step: number, optional → Spacing between values.

- For any output out, this is the distance between two adjacent values, out[i+1] - out[i].
- The default step size is 1. If step is specified, start must also be given.

dtype: dtype → The type of the output array.

- If dtype is not given, infer the data type from the other input arguments.

Returns:

arange: ndarray → Array of evenly spaced values.

- For floating point arguments, the length of the result is ceil((stop - start)/step).
- Because of floating point overflow, this rule may result in the last element of out being greater than stop.

Run the following 3 code blocks:

```
In [56]:  import numpy as np  
          np.arange(8)
```

```
Out[56]: array([0, 1, 2, 3, 4, 5, 6, 7])
```

```
In [57]: In ► import numpy as np  
np.arange(3, 8)
```

```
Out[57]: array([3, 4, 5, 6, 7])
```

```
In [58]: In ► import numpy as np  
np.arange(3, 19, 2)
```

```
Out[58]: array([ 3,  5,  7,  9, 11, 13, 15, 17])
```

3.2.9 linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)

Returns **evenly spaced** numbers over a specified interval.

Returns **num evenly spaced samples**, calculated over the interval [start, stop].

The endpoint of the interval can optionally be excluded.

Parameters:

start: scalar → The starting value of the sequence.

stop: scalar → The end value of the sequence, unless the endpoint is set to False.

- In that case, the sequence consists of all but the last of num + 1 evenly spaced samples, so that stop is excluded. **Note:** the step size changes when endpoint is False.

num: int, optional → Number of samples to generate. Default is 50. Must be non-negative.

endpoint: bool, optional

- If True, stop is the last sample.
 - Otherwise, it is not included. Default is True.

Retstep: bool, optional

- If True, return (samples, step), where step is the spacing between samples.

dtype: dtype, optional → The type of the output array.

- If dtype is not given, infers the data type from the other input arguments.

Returns: samples : ndarray

- There are num equally spaced samples in the closed interval [start, stop] or the half-open interval [start, stop) (depending on whether endpoint is True or False)

step: float, optional

- Only returned if retstep is True
- Size of spacing between samples.

Run the following 4 code blocks:

```
In [59]: In import numpy as np
np.linspace(2.0, 3.0, num=5)
```

```
Out[59]: array([2. , 2.25, 2.5 , 2.75, 3.  ])
```

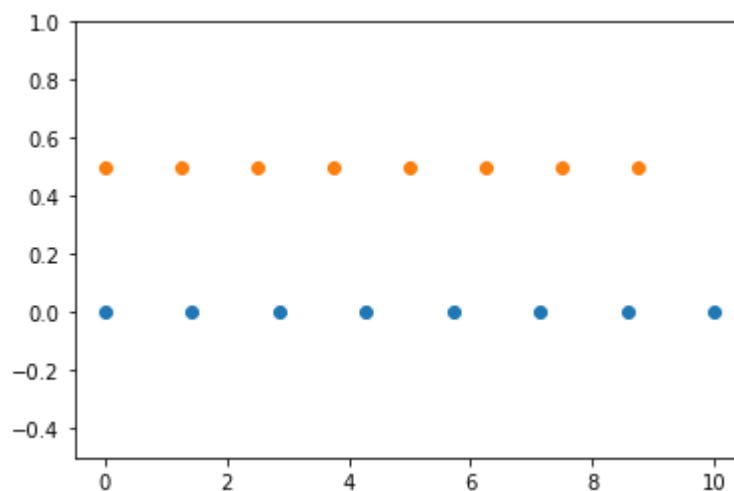
```
In [60]: In import numpy as np
np.linspace(2.0, 3.0, num=5 , endpoint=False)
```

```
Out[60]: array([2. , 2.2, 2.4, 2.6, 2.8])
```

```
In [61]: In import numpy as np
np.linspace(2.0, 3.0, num=5, retstep=True)
```

```
Out[61]: (array([2. , 2.25, 2.5 , 2.75, 3.  ]), 0.25)
```

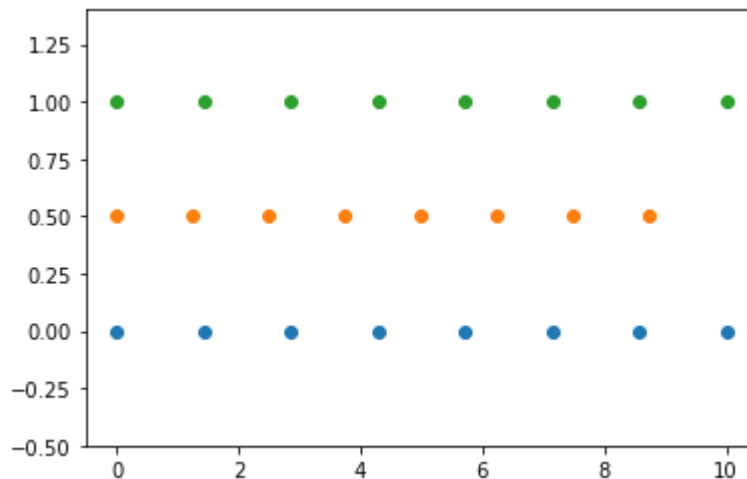
```
In [69]: In import numpy as np
import matplotlib.pyplot as plt
N = 8
y = np.zeros(N)
x1 = np.linspace(0, 10, N, endpoint=True)
x2 = np.linspace(0, 10, N, endpoint=False)
plt.plot(x1, y, 'o') # [<matplotlib.lines.Line2D object at 0x... >]
plt.plot(x2, y + 0.5, 'o') # [<matplotlib.lines.Line2D object at 0x... >]
plt.ylim([-0.5, 1])
(-0.5, 1)
plt.show()
```



PRACTICE

- Add another linspace to the graph that is $y + 1.0$
- Make sure shows on the graph

```
In [12]: #To Do add your code here
import numpy as np
import matplotlib.pyplot as plt
N = 8
y = np.zeros(N)
x1 = np.linspace(0, 10, N, endpoint=True)
x2 = np.linspace(0, 10, N, endpoint=False)
x3 = np.linspace(0, 10, N, endpoint=True)
plt.plot(x1, y, 'o') # [<matplotlib.lines.Line2D object at 0x... >]
plt.plot(x2, y + 0.5, 'o') # [<matplotlib.lines.Line2D object at 0x... >]
plt.plot(x3, y + 1.0, 'o') # [<matplotlib.lines.Line2D object at 0x... >]
plt.ylim([-0.5, 1.4])
(-0.5, 1)
plt.show()
```



3.3 NumPy Arrays: Creation: from Existing Data

3.3.1 array(object, dtype=None, copy=True, order='K', subok=False, ndmin=0)

Create an array from existing data

Parameters:

object: array_like → An array, any object exposing the array interface, an object whose array method returns an array, or any (nested) sequence.

dtype: data-type, optional → The desired data-type for the array.

- If not given, then the type will be determined as the minimum type required to hold the objects in the sequence.
- This argument can only be used to 'upcast' the array. For downcasting, use the `.astype(t)` method.

copy: bool, optional

- If true (default), then the object is copied.

- Otherwise, a copy will only be made if **array** returns a copy, if obj is a nested sequence, or if a copy is needed to satisfy any of the other requirements (dtype, order, etc.).

order: {'K', 'A', 'C', 'F'}, optional

- Specify the memory layout of the array. If object is not an array, the newly created array will be in C order (row major) unless 'F' is specified, in which case it will be in Fortran order (column major).
- If object is an array the following holds.

Order: no copy copy = True

K' unchanged F & C order preserved, otherwise most similar order

A' unchanged F order if input is F and not C, otherwise C order

C' C order C order

F' F order F order

When copy=False and a copy is made for other reasons, the result is the same as if copy=True, with some exceptions for A, see the Notes section.

The default order is 'K'.

subok: bool, optional

- If True, then sub-classes will be passed-through,
 - otherwise the returned array will be forced to be a base-class array (default).

ndmin: int, optional

- Specifies the minimum number of dimensions that the resulting array should have.
- Ones will be pre-pended to the shape as needed to meet this requirement.

Returns:

out: ndarray

An array object satisfying the specified requirements.

```
In [74]:  ▶ import numpy as np
          np.array([1, 2, 3])
```

```
Out[74]: array([1, 2, 3])
```

```
In [75]:  ▶ import numpy as np
          np.array([[1, 2], [3, 4]])
```

```
Out[75]: array([[1, 2],
                [3, 4]])
```

```
In [76]: ▶ np.array([1, 2, 3, 4, 5], ndmin=2)
```

```
Out[76]: array([[1, 2, 3, 4, 5]])
```

3.3.2 asarray(a, dtype=None, order=None)

Convert the input a into an array

Parameters:

a: array_like → Input data, in any form that can be converted to an array.

- (This includes lists, lists of tuples, tuples, tuples of tuples, tuples of lists and ndarrays.)

dtype: data-type, optional → By default, the data-type is inferred from the input data.

order: {'C', 'F'}, optional

- Whether to use row-major (C-style) or column-major (Fortran-style) memory representation. Defaults to 'C'.

Returns:

out: ndarray

Array interpretation of a. No copy is performed if the input is already an ndarray with matching dtype and order. If a is a subclass of ndarray, a base class ndarray is returned.

Run the following code block:

```
In [77]: ▶ import numpy as np
a = [1, 2, 3, 4, 5]
b = np.asarray([1, 2, 3, 4, 5])
print (b)
c = np.asarray(a)
print (c)
```

```
[1 2 3 4 5]
[1 2 3 4 5]
```

3.3.3 fromstring(string, dtype=float, count=-1, sep="")

A new 1-D array initialized from raw binary or text data in a string.

Parameters:

string: str → A string containing the data.

dtype: data-type, optional → The data type of the array; default: float.

- For binary input data, the data must be in exactly this format.

count: int, optional → Read this number of dtype elements from the data.

- If this is negative (the default), the count will be determined from the length of the data.)

sep: str, optional

- If not provided or, equivalently, the empty string, the data will be interpreted as binary data.
- Otherwise, as ASCII text with decimal numbers.
- Also in this latter case, this argument is interpreted as the string separating numbers in the data; extra whitespace between elements is also ignored.

Returns: **arr:** ndarray

Run the following 2 code blocks:

```
In [78]: ▶ import numpy as np
aStr = "This is a sentence."
np.fromstring(aStr, dtype=np.uint8)
```

```
Out[78]: array([ 84, 104, 105, 115, 32, 105, 115, 32, 97, 32, 115, 101, 110,
116, 101, 110, 99, 101, 46], dtype=uint8)
```

```
In [79]: ▶ import numpy as np
aStr = "This is a sentence."
anArray = np.fromstring(aStr, dtype=np.uint8)
print(anArray)
```

```
[ 84 104 105 115 32 105 115 32 97 32 115 101 110 116 101 110 99 101
46]
```

3.3.4 diag(v, k=0)

Extract a diagonal or construct a diagonal array.

See the more detailed documentation for `numpy.diagonal` if you use this function to extract a diagonal and wish to write to the resulting array; whether it returns a copy or a view depends on what version of numpy you are using.

Parameters:

v: array_like

- If v is a 2-D array, returns a copy of its k-th diagonal.
- If v is a 1-D array, return a 2-D array with v on the k-th diagonal.

k: int, optional

- Diagonal in question. The default is 0.
- Use $k > 0$ for diagonals above the main diagonal.
- $k < 0$ for diagonals below the main diagonal.

Returns:**out:** ndarray

The extracted diagonal or constructed diagonal array.

Run the following 4 code blocks:

```
In [80]: ▶ import numpy as np
x = np.arange(9).reshape((3, 3))
print(x)

[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

```
In [81]: ▶ import numpy as np
x = np.arange(9).reshape((3,3))
x
```

```
Out[81]: array([[0, 1, 2],
               [3, 4, 5],
               [6, 7, 8]])
```

```
In [82]: ▶ import numpy as np
x = np.arange(9).reshape((3,3))
np.diag(x)
```

```
Out[82]: array([0, 4, 8])
```

```
In [83]: ▶ import numpy as np
x = np.arange(9).reshape((3,3))
np.diag(x, k = 1)
```

```
Out[83]: array([1, 5])
```



4. Creation of NumPy Arrays: Simple Methods

4.1. 1-D Arrays

4.1.1 Using ndarray.arange()

Run the following code block:

```
In [84]: ▶ import numpy as np  
         np.arange(8)
```

```
Out[84]: array([0, 1, 2, 3, 4, 5, 6, 7])
```

4.2 2-D Arrays

4.2.1 Using ndarray.arange().reshape(a, b)

Run the following 4 code blocks:

```
In [85]: ▶ x = np.arange(12).reshape((3,4))  
         x
```

```
Out[85]: array([[ 0,  1,  2,  3],  
                [ 4,  5,  6,  7],  
                [ 8,  9, 10, 11]])
```

5. Numpy: Built-In Functions: Array Manipulation

5.1 Overview

reshape(a, newshape, order='C') Gives a new shape to an array without changing its data.

flat() A 1-D iterator over the array.

flatten(order='C') Return a copy of the array collapsed into one dimension.

transpose(a, axes=None) Permute the dimensions of an array.

concatenate((a1, a2, ...), axis=0) Join a sequence of arrays along an existing axis.

split(ary, indices_or_sections, axis=0) Split an array into multiple sub-arrays.

delete(arr, obj, axis=None) Returns a new array with sub-arrays along an axis deleted. For a one dimensional array, this returns those entries not returned by arr[obj].

insert(arr, obj, values, axis=None) Insert values along the given axis before the given indices.

append(arr, values, axis=None) Append values to the end of an array.

resize (a, new_shape) Returns a new array with the specified shape. If the new array is larger than the original array,

- then the new array is filled with repeated copies of a. **NOTE:** this behavior is different from a.resize(new_shape),
- which fills with zeros instead of repeated copies of a.

trim_zeros(filt, trim='fb') Trims the leading and/or trailing zeros from a 1-D array or sequence.

unique(ar, return_index=False, return_inverse=False, return_counts=False, axis=None) Finds the unique elements of an array. Returns the sorted unique elements of an array. There are three optional outputs in addition to the unique elements:

- the indices of the input array that give the unique values,
- the indices of the unique array that reconstruct the input array,
- the number of times each unique value comes up in the input array.

flip(m, axis) Reverse the order of elements in an array along the given axis. The shape of the array is preserved, but the elements are reordered .

fliplr(m) Flips array in the left/right direction. Flips the entries in each row in the left/right direction. Columns are preserved but appear in a different order than before.

flipud (m) Flips array in the up/down direction. Flips the entries in each column in the up/down direction. Rows are preserved, but appear in a different order than before.

tile(A, reps) Construct an array by repeating A the number of times given by reps. If reps has length d, the result will have dimension of max(d, A.ndim).

repeat(a, repeats, axis=None)

Repeats elements of an array.

You are done. Great job!

In []: ▶

#Vijay Raj Pathani, 6/25/2023

Assignment 3 Hands-on

EDA: Python Data Visualization with Matplotlib, Pandas, and NumPy

Import Libraries and Load Dataset

In [1]:  *# Importing the libraries which are needed to run the code.*

```
import pandas as pd
import numpy as np

# filter warnings
import warnings
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt

from pandas.plotting import scatter_matrix
from pandas import DataFrame, read_csv
```

In [2]:  *# Import os module which is a miscellaneous operating system interface*
import os

```
#Use a command in the os module to show your current working directory for
cwd = os.getcwd()
cwd
```

Out[2]: `'/home/ab3c60f8-eb8c-437b-8ff6-5de59aebbf91'`

```
In [3]: # Load the data set 'iris.csv' into a pandas dataframe
# Read the iris data set and create the dataframe df

df = ('iris.csv')
df = pd.read_csv ('iris.csv')
df.head(5)
```

```
Out[3]:
```

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa

```
In [4]: #print the information about the dataset

print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 6 columns):
#   Column          Non-Null Count  Dtype
---  -
0   Id               150 non-null   int64
1   SepalLengthCm    150 non-null   float64
2   SepalWidthCm     150 non-null   float64
3   PetalLengthCm    150 non-null   float64
4   PetalWidthCm     150 non-null   float64
5   Species          150 non-null   object
dtypes: float64(4), int64(1), object(1)
memory usage: 7.2+ KB
None
```

Univariate Data Visualization

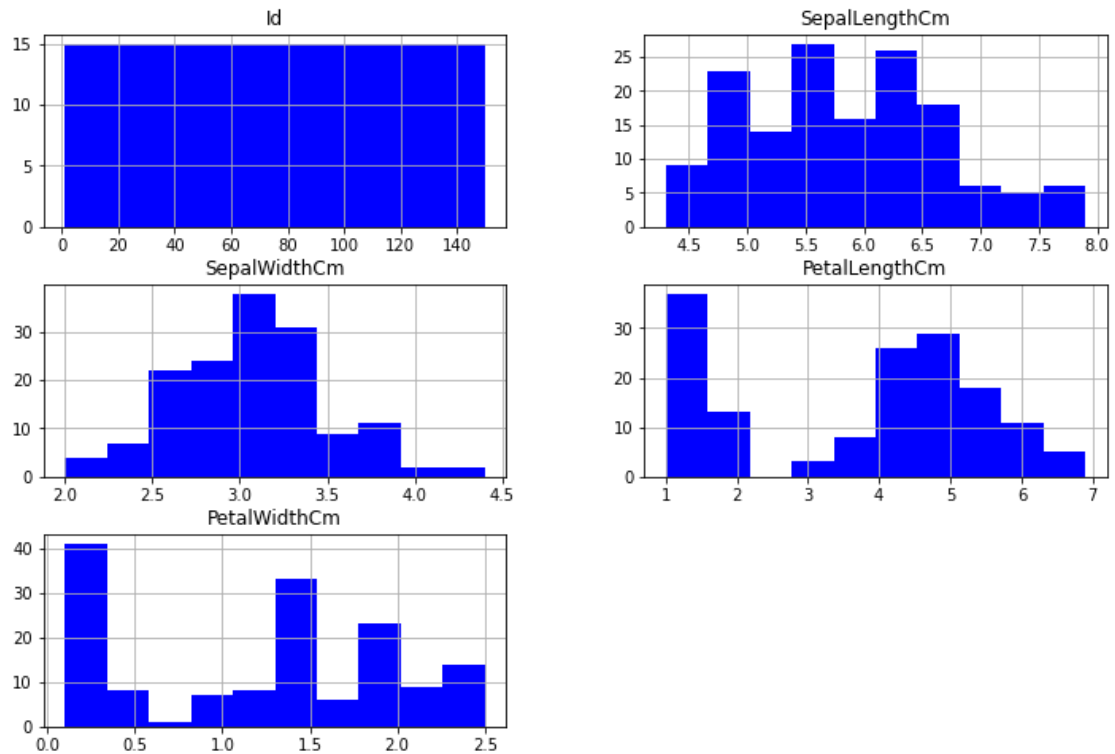
Histograms

- Histograms are great when we would like to show the distribution of the data we are working with.

```
In [5]: ▶ # Create a histogram
# SepalWidth - normal dist, PetalLength - bimodal
# The normal distribution is so important easier for mathematical statistics
# Many kinds of statistical tests can be derived from normal distributions

df.hist(figsize=(12,8), color='blue')
plt.show
```

Out[5]: <function matplotlib.pyplot.show(close=None, block=None)>



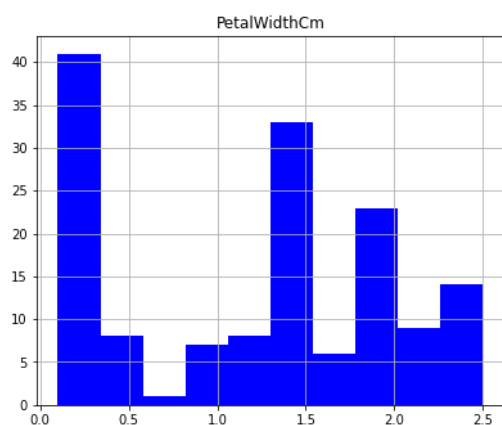
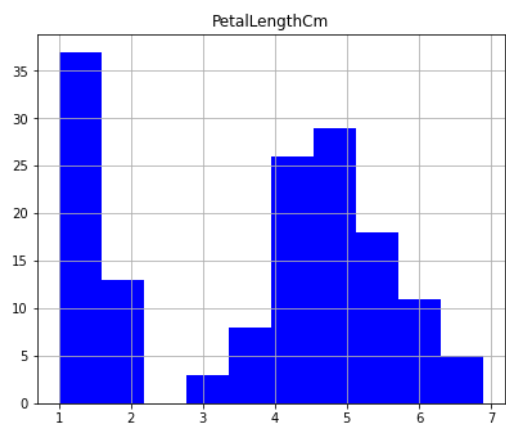
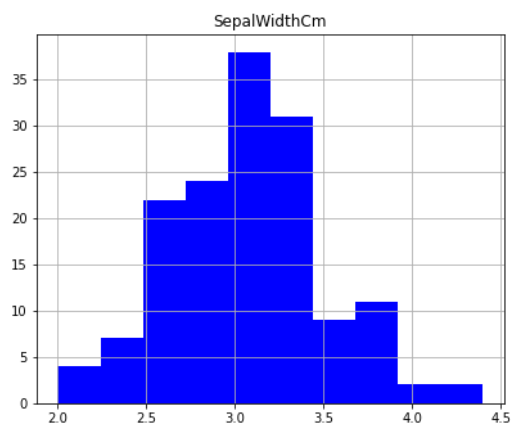
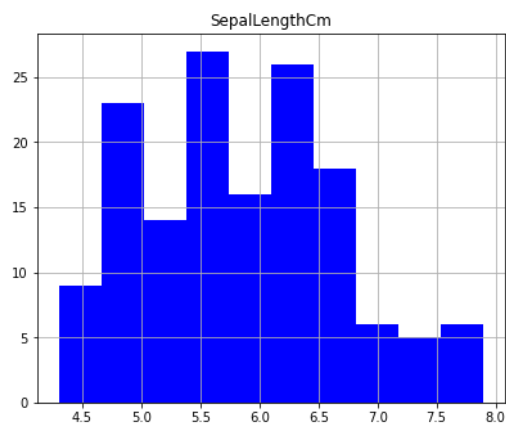
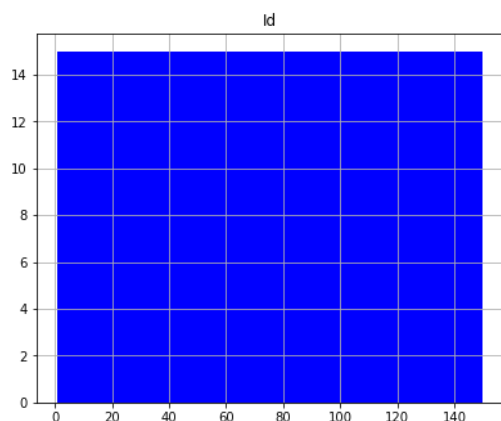
```
In [6]: ▶ # Here we want to see the different Species

print(df.groupby('Species').size())
```

```
Species
Iris-setosa      50
Iris-versicolor  50
Iris-virginica   50
dtype: int64
```

```
In [7]: # Histograms are great when we would like to show the distribution of the  
df.hist(figsize=(15,19), color='blue')  
plt.show
```

```
Out[7]: <function matplotlib.pyplot.show(close=None, block=None)>
```

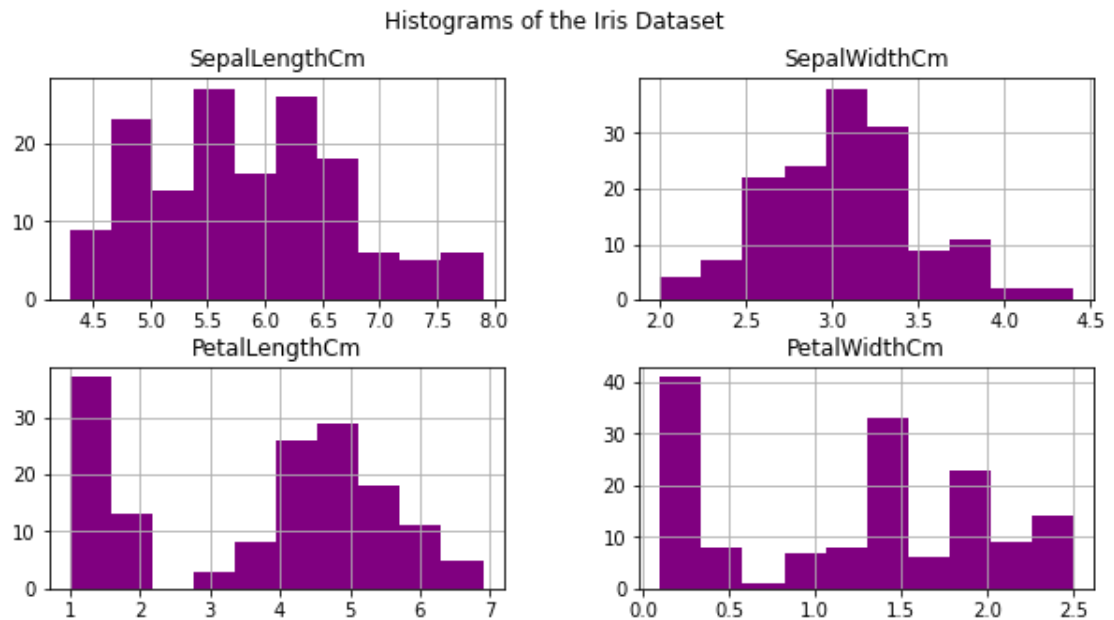



```
In [8]: # In the above code, we see the variable "Id" is included in the analysis.  
# In order to get rid of this variable we use the following code.  
# It doesn't provide any output.
```

```
df.__delitem__('Id')
```

```
In [9]: df.hist(figsize=(10,5), color ="purple")  
plt.suptitle("Histograms of the Iris Dataset")  
plt.show  
  
# After this run, we see that "Id" is gone. you can also see we changed th
```

```
Out[9]: <function matplotlib.pyplot.show(close=None, block=None)>
```

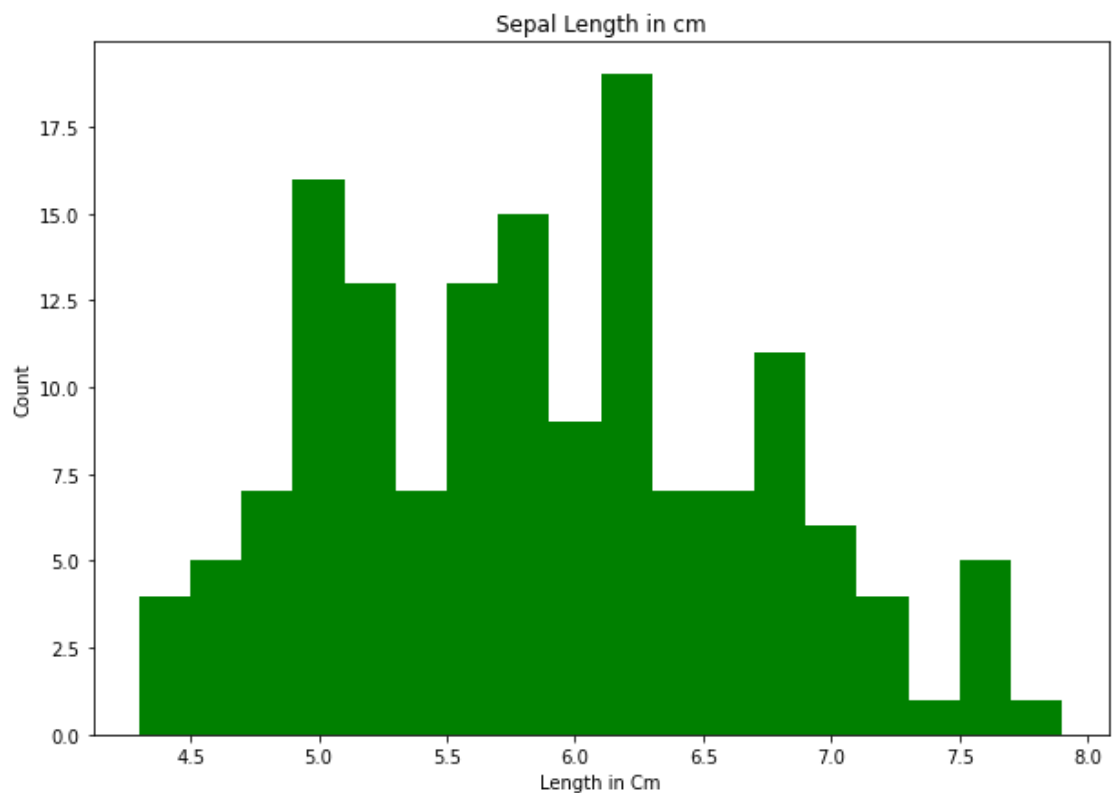


Let's say we want to make changes to the histograms.

We will look at only the PetalLengthCm variable.

```
In [10]: ▶ # A histogram with just one variable - Sepal Length.  
# We need to isolate the one variable using square brackets [].  
  
plt.figure(figsize = (10, 7))  
x = df["SepalLengthCm"]  
  
# bins is an integer, it defines the number of equal-width bins in the range  
plt.hist(x, bins = 18, color = "green")  
plt.title("Sepal Length in cm")  
plt.xlabel("Length in Cm")  
plt.ylabel("Count")
```

Out[10]: Text(0, 0.5, 'Count')

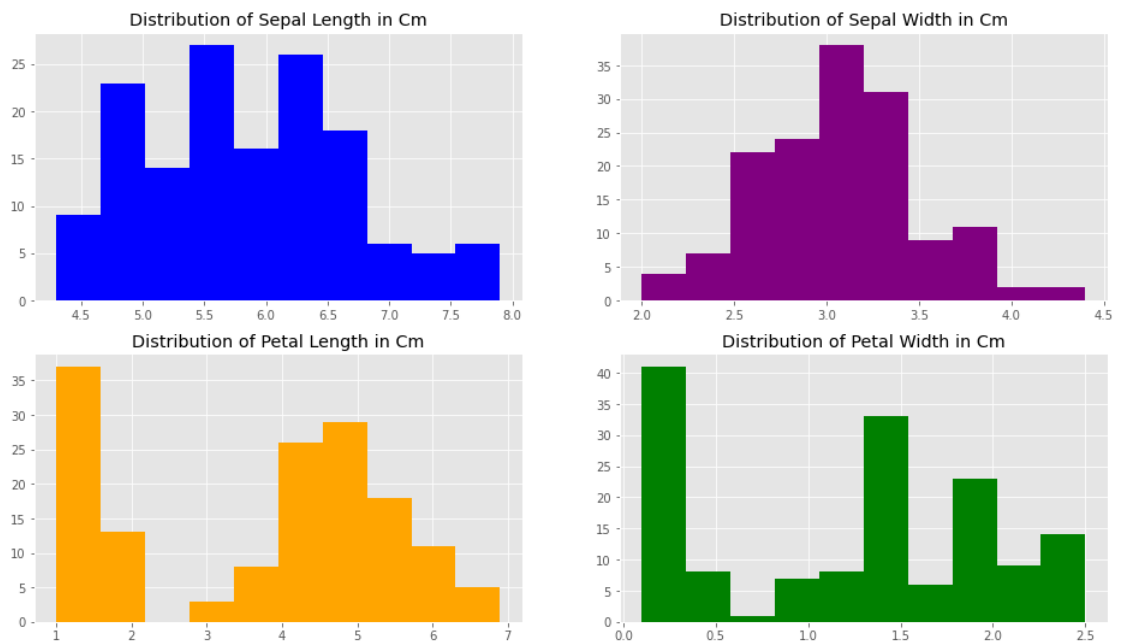


```
In [11]: # Here we are isolating the variable and giving each one a color and a spe  
# grammar of graphics plot
```

```
plt.style.use("ggplot")
```

```
fig, axes = plt.subplots(2, 2, figsize=(16,9))
```

```
axes[0,0].set_title("Distribution of Sepal Length in Cm")  
axes[0,0].hist(df['SepalLengthCm'], bins=10, color='blue');  
axes[0,1].set_title("Distribution of Sepal Width in Cm")  
axes[0,1].hist(df['SepalWidthCm'], bins=10, color='purple');  
axes[1,0].set_title("Distribution of Petal Length in Cm")  
axes[1,0].hist(df['PetalLengthCm'], bins=10, color='orange');  
axes[1,1].set_title("Distribution of Petal Width in Cm")  
axes[1,1].hist(df['PetalWidthCm'], bins=10, color='green');
```

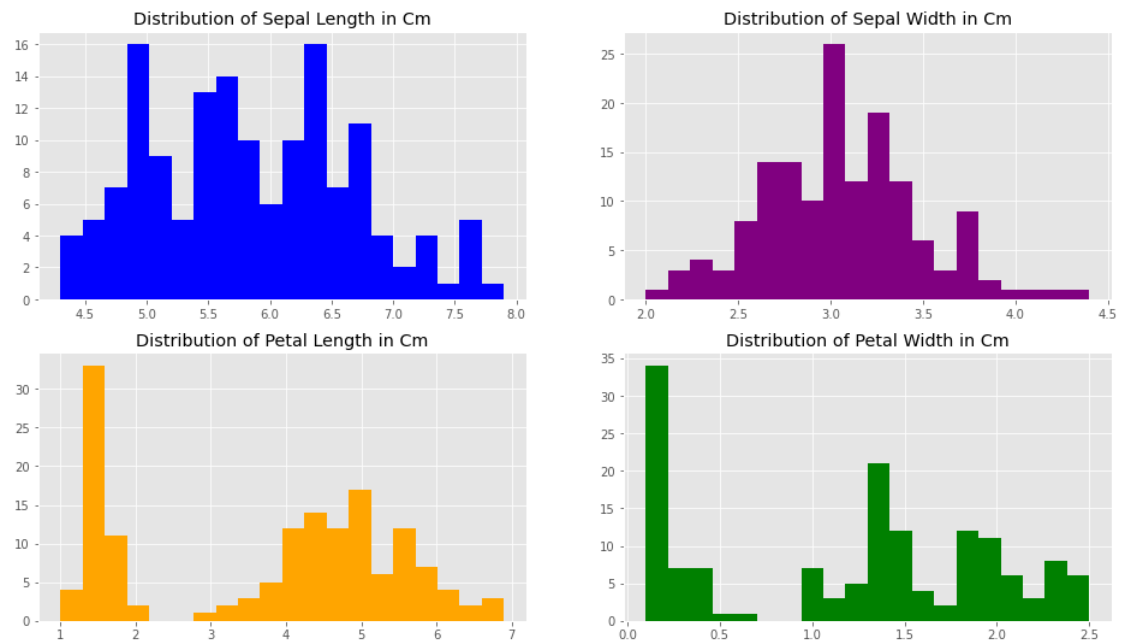


In [12]: `# Lastly, let's change the bin size`

```
plt.style.use("ggplot")

fig, axes = plt.subplots(2, 2, figsize=(16,9))

axes[0,0].set_title("Distribution of Sepal Length in Cm")
axes[0,0].hist(df['SepalLengthCm'], bins=20, color='blue');
axes[0,1].set_title("Distribution of Sepal Width in Cm")
axes[0,1].hist(df['SepalWidthCm'], bins=20, color='purple');
axes[1,0].set_title("Distribution of Petal Length in Cm")
axes[1,0].hist(df['PetalLengthCm'], bins=20, color='orange');
axes[1,1].set_title("Distribution of Petal Width in Cm")
axes[1,1].hist(df['PetalWidthCm'], bins=20, color='green');
```

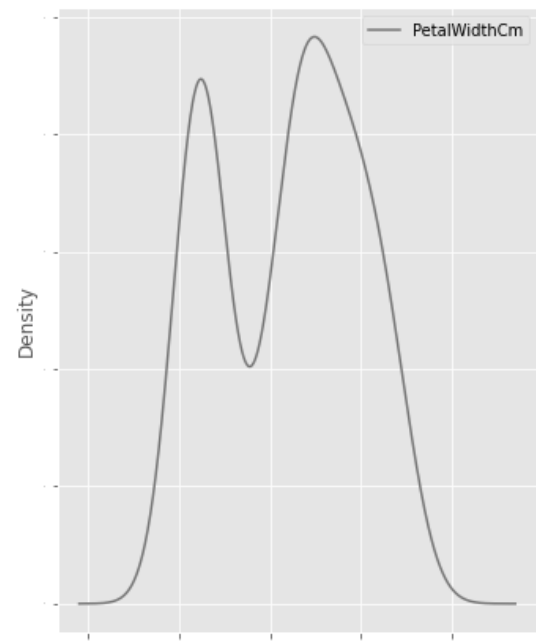
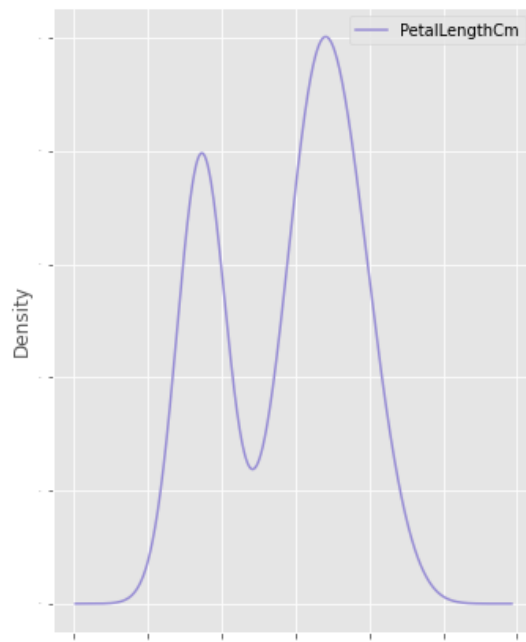
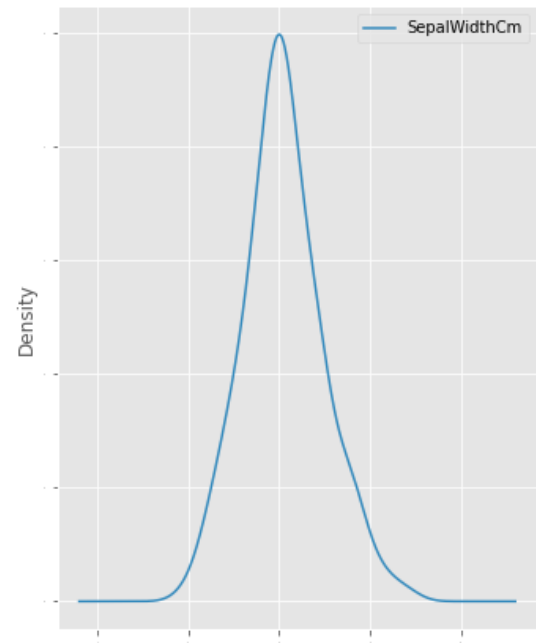
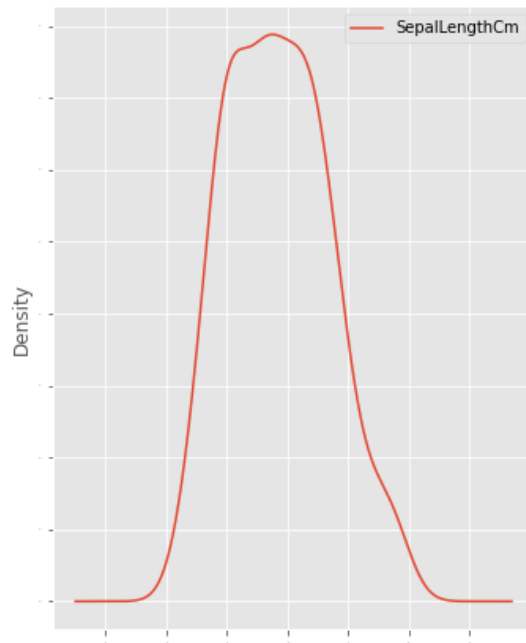


Density Plots

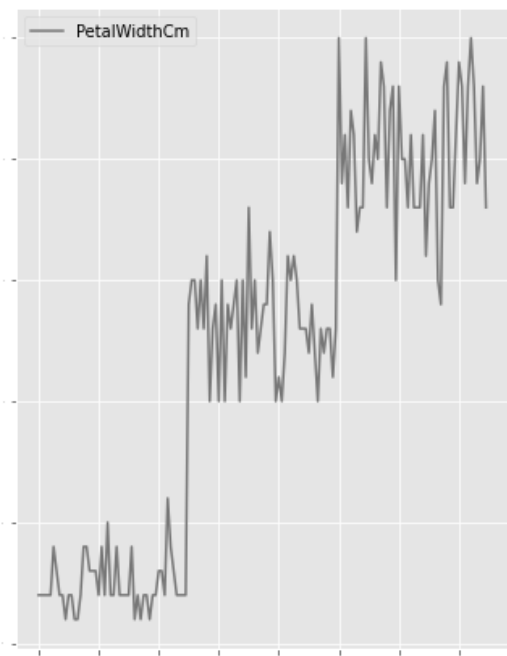
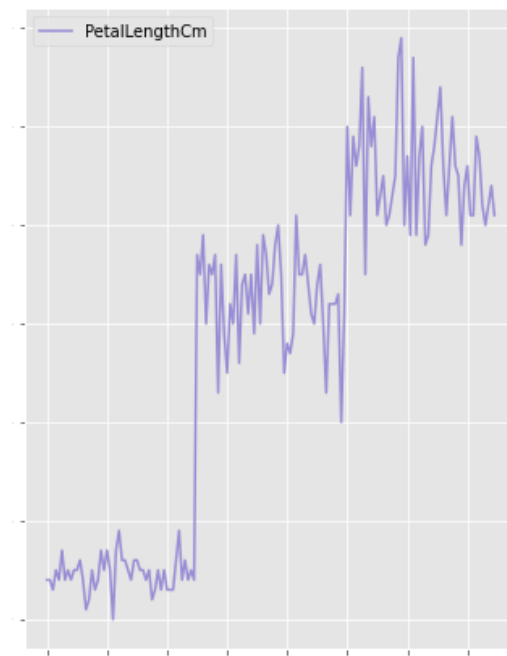
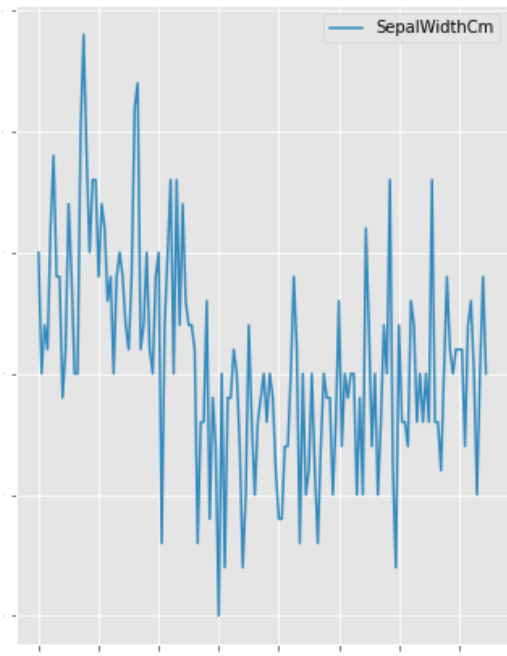
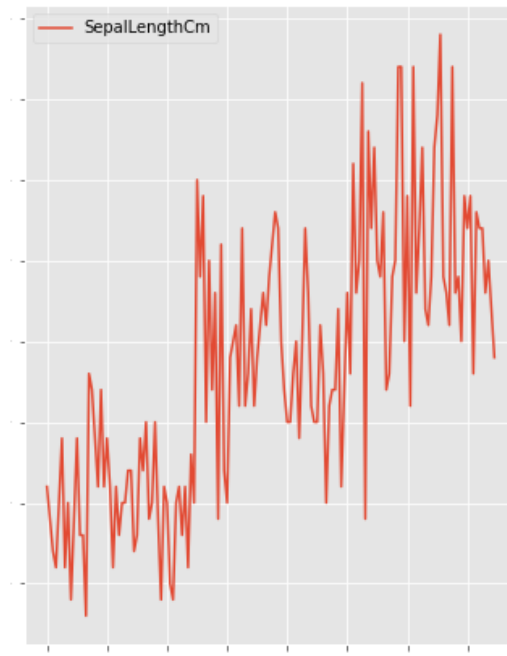
- A Density Plot visualizes the distribution of data over a time period or a continuous interval. This chart is a variation of a Histogram but it smooths out the noise made by binning.
- Density Plots have a slight advantage over Histograms since they're better at determining the distribution shape and, as mentioned above, they are not affected by the number of bins used (each bar used in a typical histogram). As we saw above a Histogram with only 10 bins wouldn't produce a distinguishable enough shape of distribution as a 20-bin Histogram would. With Density Plots, this isn't an issue.

```
In [13]: # Create the density plot
# If subplots=True is specified, plots for each column are drawn as subplots

df.plot(kind='density', subplots=True, layout=(2,2), sharex=False, legend=True,
plt.show())
```



```
In [14]: # Create a line graph  
df.plot(kind='line', subplots=True, layout=(2,2), sharex=False, legend=True,  
plt.show())
```

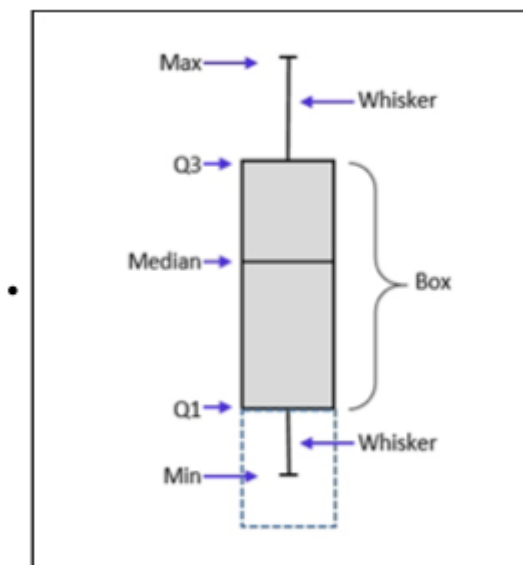


Boxplots

- A box plot is a very good plot to understand the spread, median, and outliers of data

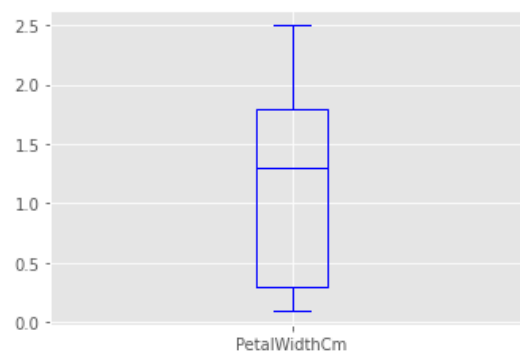
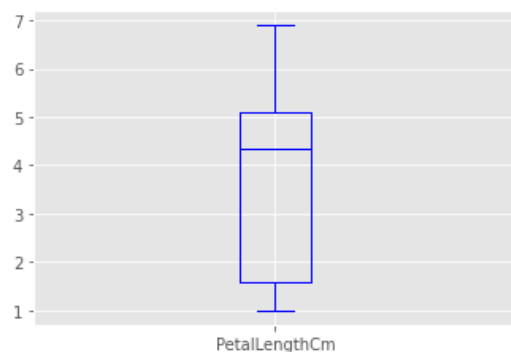
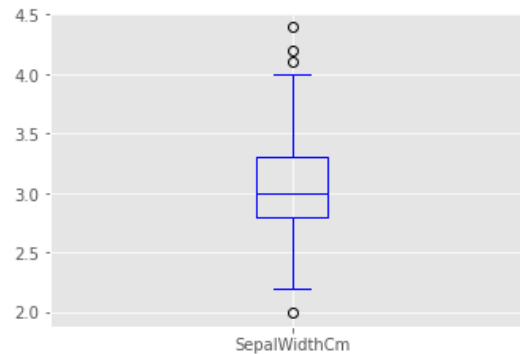
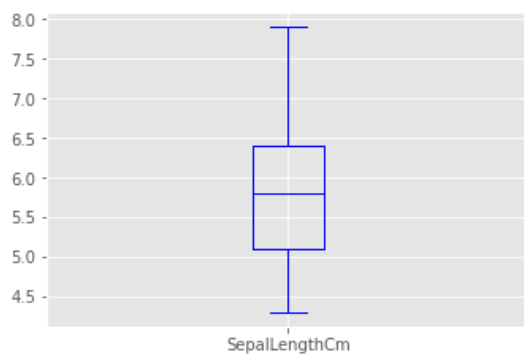
1. Q3: This is the 75th percentile value of the data. It's also called the upper hinge.
2. Q1: This is the 25th percentile value of the data. It's also called the lower hinge.

3. Box: This is also called a step. It's the difference between the upper hinge and the lower hinge.
4. Median: This is the midpoint of the data.
5. Max: This is the upper inner fence. It is 1.5 times the step above Q3.
6. Min: This is the lower inner fence. It is 1.5 times the step below Q1.

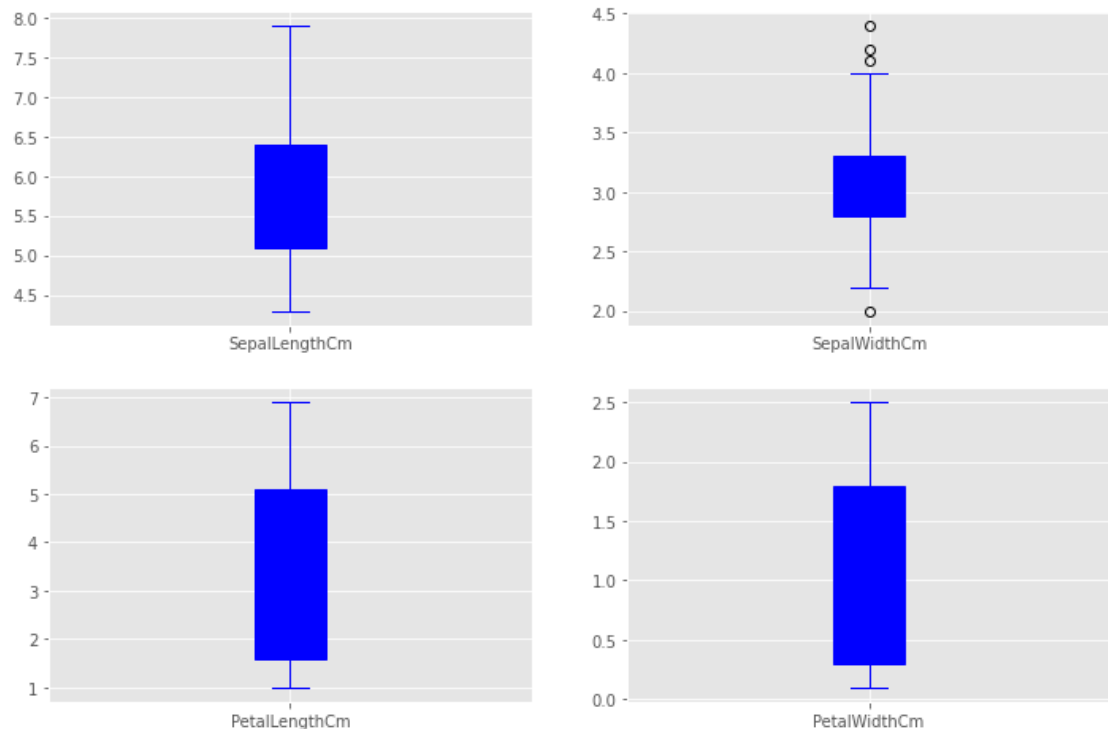


In [15]: `# Create a box plot`

```
df.plot(kind='box', subplots=True, layout=(2,2), sharex=False, sharey=False,
plt.show())
```



```
In [16]: # Fill the boxes with color, using patch_artist  
df.plot(kind='box', subplots=True, layout=(2,2), sharex=False, sharey=False,  
plt.show())
```

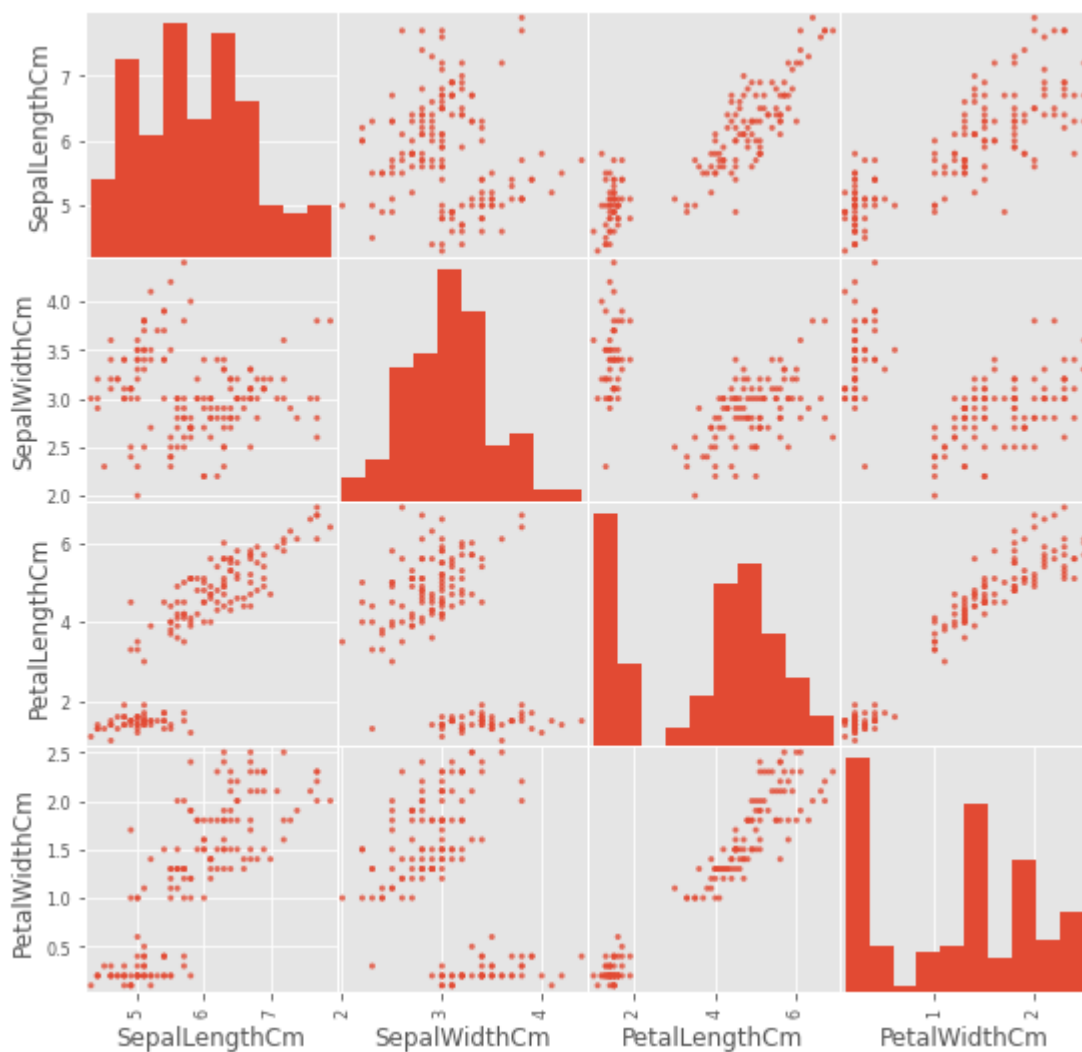


Multivariate Data Visualization

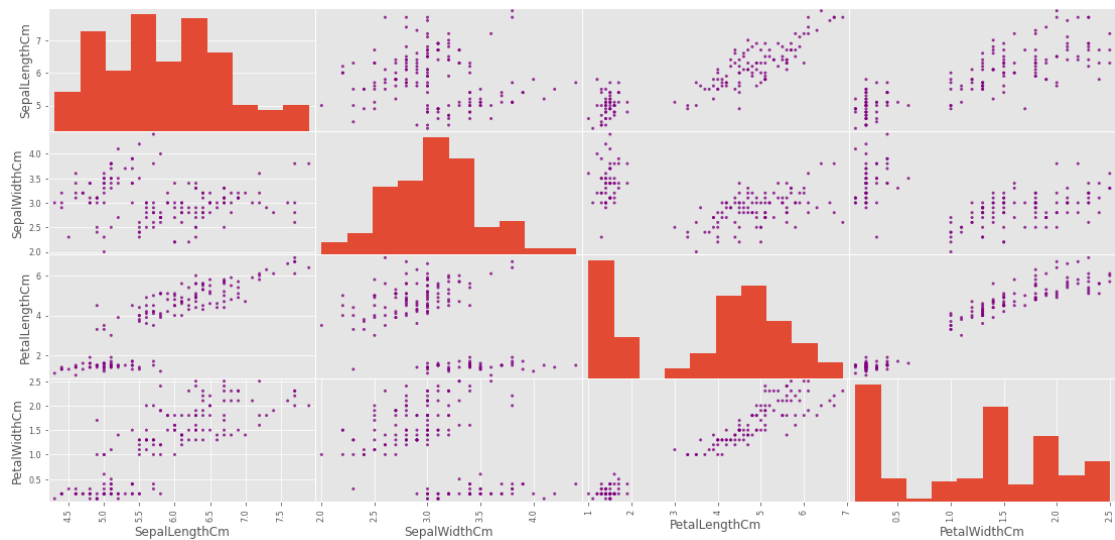
Scatter Matrix Plot

- A scatter plot matrix is a grid (or matrix) of scatter plots. This type of graph is used to visualize bivariate relationships between different combinations of variables. Each scatter plot in the matrix visualizes the relationship between a pair of variables, allowing many relationships to be explored in one chart. For example, the first row shows the relationship between SepalLength and the other 3 variables.


```
In [17]: ▶ # Create a scatter matrix plot  
# alpha = Amount of transparency applied  
  
scatter_matrix(df, alpha=0.8, figsize=(9,9))  
plt.show()
```



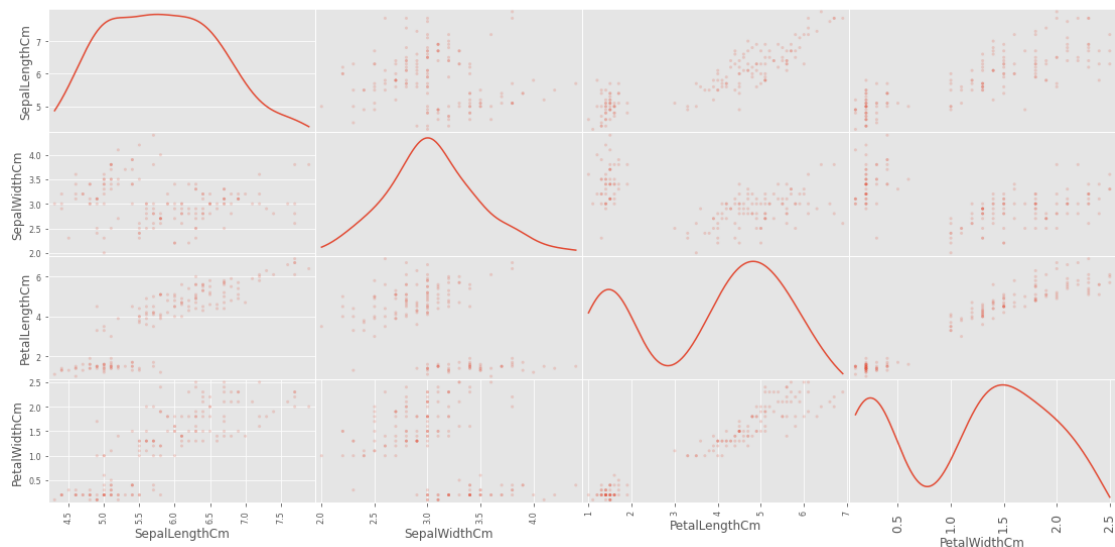
```
In [18]: # change the color to purple. Notice only one part of the plot changes.
scatter_matrix(df, alpha=0.8, figsize=(19,9), color = 'purple')
plt.show()
```



```
In [19]: # Here you are adding a subtitle.
# Pick between 'kde' and 'hist' for either Kernel Density Estimation or Histogram
# KDE = a density estimator is an algorithm which seeks to model the probability density function of a continuous random variable
```

```
scatter_matrix(df, alpha=0.2, diagonal = 'kde', figsize=(19,9))
plt.suptitle('Scatter-matrix for each input variable')
plt.tick_params(labelsize=12, pad=6)
```

Scatter-matrix for each input variable

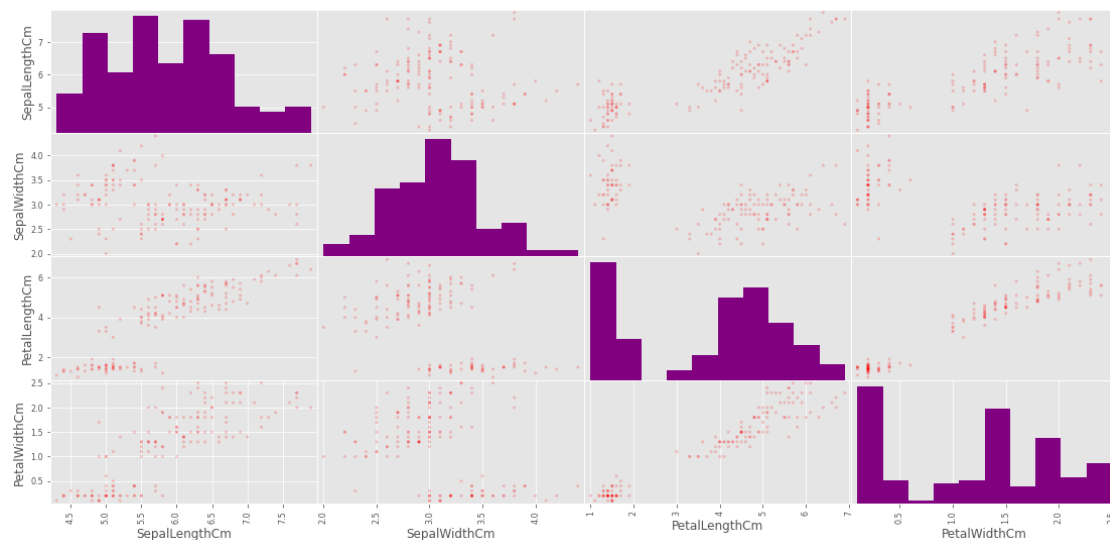


In [20]: `# Lastly, you are changing the color to both parts of the plot.`

```
scatter_matrix(df, figsize= (19,9), alpha=0.2,  
c='red', hist_kws={'color':['purple']})  
plt.suptitle('Scatter-matrix for each input variable', fontsize=28)
```

Out[20]: Text(0.5, 0.98, 'Scatter-matrix for each input variable')

Scatter-matrix for each input variable



You are done. Great job!

In []: