PYTHON PROGRAMMING

BCA IV SEM(NEP)

NOTES

Prepared By

Mr. Prabhu Kichadi, BE, MTECH

9880437187

UNIT – V

GU Interface: The tkinter Module; Window and Widgets; Layout Management- pack, grid and place.

Python SQLite: The SQLite3 module; SQLite Methods- connect, cursor, execute, close; Connect to Database; Create Table; Operations on Tables Insert, Select, Update. Delete and Drop Records.

Data Analysis: NumPy- Introduction to **NumPy,** Array Creation using NumPy, Operations on Arrays; Pandas.

Introduction to Pandas, Series and DataFrames, Creating DataFrames from Excel Sheet and .csv file, Dictionary and Tuples. Operations on DataFrames.

Data Visualisation: Introduction to Data Visualisation; **Matplotlib Library;** Different Types of Charts using Pyplot- Line chart, Bar chart and Histogram and Pie chart.

GU Interface:

GUI stands for Graphical User Interface. In Python, GUI is a way to create visual interfaces for users to interact with the program. It allows the user to interact with the program by using graphical elements such as buttons, menus, text boxes, etc. instead of typing commands in the terminal.

GUI (Graphical User Interface) is a way to provide a visual interface for users to interact with the program. In Python, there are several libraries available to create GUIs. One of the most popular is Tkinter, which comes pre-installed with Python.

The tkinter Module:

Tkinter is a built-in module in Python that provides a way to create GUI (Graphical User Interface) applications. It is the standard Python interface to the Tk GUI toolkit, which is a cross-platform toolkit that is widely used for building desktop applications.

Features of the Tkinter module:

- It provides a set of widgets, such as buttons, labels, text boxes, and frames, to create GUIs.
- It has a powerful layout manager that allows you to place widgets in the window or frame.
- It provides event handling mechanism, where you can bind an event, such as a button click, to a function.
- It supports standard dialogs such as file open dialog, message boxes, etc.
- It has a simple and easy-to-learn API.

Window and Widgets: "**window"** typically refers to the top-level container for a graphical user interface (GUI) application.

"widget" refers to any graphical element inside the window, such as buttons, text boxes, labels, etc.

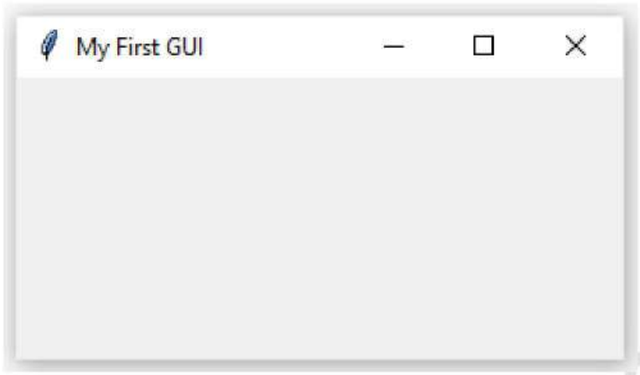
Window(Top level Containers): "**window"** typically refers to the top-level container for a graphical user interface (GUI) application.

Example:

```
import tkinter as tk
```

```
# create a new window
window = tk.Tk()
window.title("My First GUI");

# run the main loop
window.mainloop()
```

Output:



Some of the window and widgets of Tkinter

Top Level Containers	Usage
Tk	This is the main window or root window of the application. It is the first widget that should be created in any Tkinter program.
Toplevel	This is a secondary window or dialog box that can be created using the Tkinter Toplevel widget. It can be used to display additional information or to prompt the user for input.
Frame	This is a container widget that can be used to group other widgets together. It can be used to organize the layout of the application.
PanedWindow:	This is a container widget that can be used to create a resizable pane with multiple sub-panes. It can be used to create a split view or a resizable layout.

Widgets(Low level components): In Tkinter, widgets are the low-level components used to build the graphical user interface (GUI). Here are some of the most commonly used widgets in Tkinter:

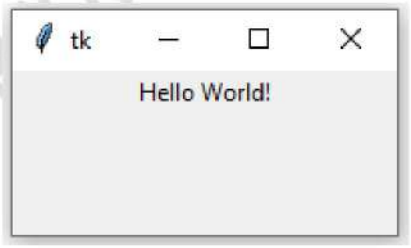
1. **Label:** This widget is used to display text or images.
2. **Button:** This widget is used to trigger an action when clicked.
3. **Entry:** This widget is used to get user input through a single line text box.
4. **Text:** This widget is used to display or edit multi-line text.
5. **Checkbutton:** This widget is used to toggle a boolean value.
6. **Radiobutton:** This widget is used to select a single option from a group of options.
7. **Listbox:** This widget is used to display a list of items from which the user can select one or more.
8. **Scale:** This widget is used to select a value from a range of values by dragging a **slider.**
9. **Canvas:** This widget is used to draw graphics, shapes, and images.
10. **Menu**: This widget is used to create menus and sub-menus.

1. Label Widget: The Label widget in Tkinter is used to display text or images on the GUI. It is a simple widget that can be created using the Label class from the tkinter module.

Ex:

```
import tkinter as tk
root = tk.Tk()
label = tk.Label(root, text="Hello World!")
label.pack()
root.mainloop()
```

Output:



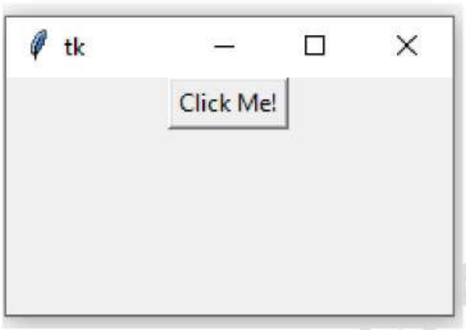
Hello World!

2. Button Widget: The Button widget in Tkinter is used to create a clickable button on the GUI. It is a simple widget that can be created using the Button class from the tkinter module.

Ex:

```
import tkinter as tk

def button_click():
   print("Button clicked")

root = tk.Tk()
button = tk.Button(root, text="Click Me!", command=button_click)
button.pack()
root.mainloop()
```

output:

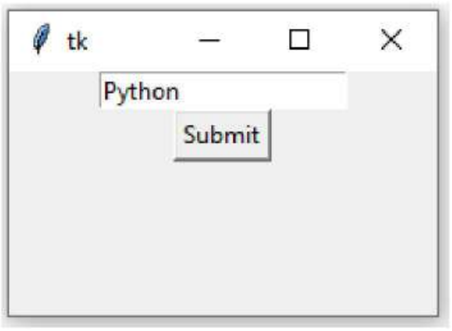


3. Entry Widget: The Entry widget in Tkinter is used to create a single-line text box where the user can enter text or values. It is a simple widget that can be created using the Entry class from the tkinter module.

Ex:

```
import tkinter as tk

def submit():
   value = entry.get()
   print("User entered:", value)

root = tk.Tk()
entry = tk.Entry(root)
entry.pack()

submit_button = tk.Button(root, text="Submit", command=submit)
submit_button.pack()

root.mainloop()
```

Output:

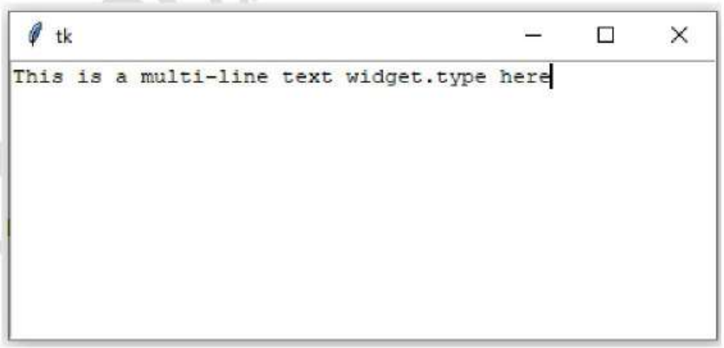


4. Text widget: The Text widget in Tkinter is used to create a multi-line text box where the user can enter or display multiple lines of text. It is a versatile widget that can be created using the Text class from the tkinter module.

Ex:

```
import tkinter as tk

root = tk.Tk()

text = tk.Text(root)
text.pack()

text.insert(tk.END, "This is a multi-line text widget.")

root.mainloop()
```

Output:



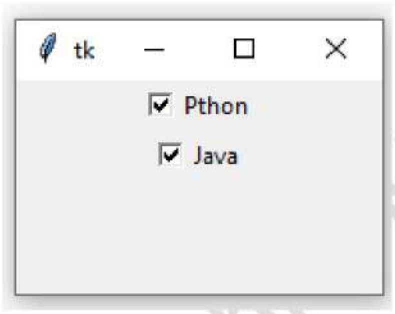
```
This is a multi-line text widget.type here
```

5. Checkbutton Widget: The Checkbutton widget in Tkinter is used to create a checkbox that can be selected or deselected by the user. It is a simple widget that can be created using the Checkbutton class from the tkinter module.

Ex:

```
import tkinter as tk

root = tk.Tk()

var1 = tk.BooleanVar()
var2 = tk.BooleanVar()
ckb1 = tk.Checkbutton(root, text="Pthon", variable=var1)
ckb2 = tk.Checkbutton(root, text="Java", variable=var2)
ckb1.pack()
ckb2.pack()

root.mainloop()
```

Output:



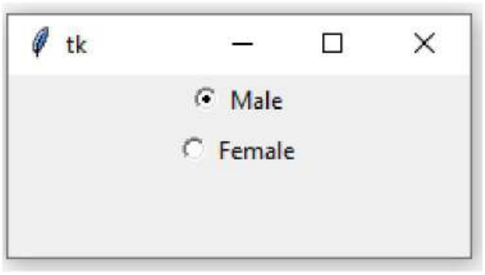
6. Radiobutton Widget: The Radiobutton widget in Tkinter is used to create a group of radio buttons where only one option can be selected at a time. It is a simple widget that can be created using the Radiobutton class from the tkinter module.

Ex:

```
import tkinter as tk

root = tk.Tk()

var = tk.StringVar(value="Option 1")
```

```
r1 = tk.Radiobutton(root, text="Male", variable=var, value="Male")
r1.pack()

r2 = tk.Radiobutton(root, text="Female", variable=var, value="Female")
r2.pack()

root.mainloop()
```

Output:

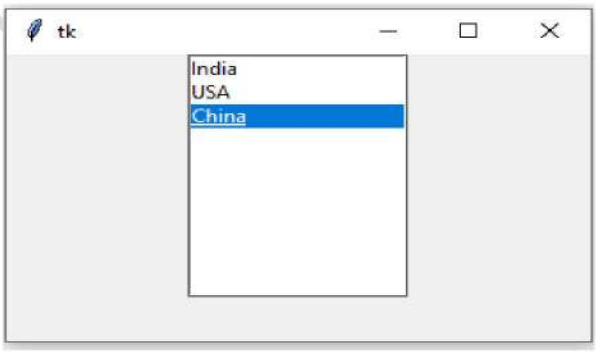


7. Listbox Widget: The Listbox widget in Tkinter is used to display a list of items from which the user can select one or more items. It is a versatile widget that can be created using the Listbox class from the tkinter module.

Ex:

```
import tkinter as tk
root = tk.Tk()
lb = tk.Listbox(root)
lb.pack()

lb.insert(1, "India")
lb.insert(2, "USA")
lb.insert(3, "China")

root.mainloop()
```

Output:

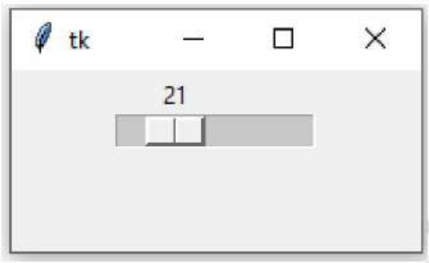


8. Scale Widget: The Scale widget in Tkinter is used to create a slider that allows the user to select a value from a range of values. It is a simple widget that can be created using the Scale class from the tkinter module.

Ex:

```
import tkinter as tk

root = tk.Tk()

scale = tk.Scale(root, from_=0, to=100, orient=tk.HORIZONTAL)
scale.pack()

root.mainloop()
```

Output:

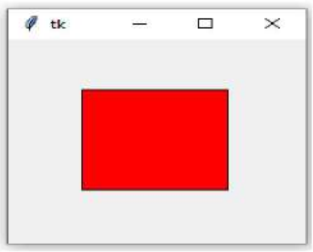


9. Canvas Widget: The Canvas widget in Tkinter is a versatile widget that allows you to create and manipulate 2D graphics. It provides a virtual canvas on which you can draw various shapes and images, as well as add text and other widgets.

Ex:

```
import tkinter as tk
root = tk.Tk()
canvas = tk.Canvas(root, width=200, height=200)
canvas.pack()
canvas.create_rectangle(50, 50, 150, 150, fill="red")
root.mainloop()
```

Output:

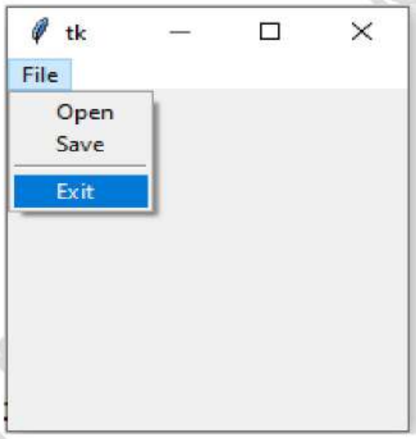


10. Menu Widget: The Menu widget in Tkinter provides a way to create menus in your GUI. It allows you to create a hierarchical structure of menu items, with each item either triggering a command or displaying a submenu.

Ex:

```
import tkinter as tk

root = tk.Tk()

menu_bar = tk.Menu(root)

file_menu = tk.Menu(menu_bar, tearoff=0)
file_menu.add_command(label="Open")
file_menu.add_command(label="Save")
file_menu.add_separator()
file_menu.add_command(label="Exit", command=root.quit)

menu_bar.add_cascade(label="File", menu=file_menu)

root.config(menu=menu_bar)

root.mainloop()
```

Output:



Layout Management- pack, grid and place.

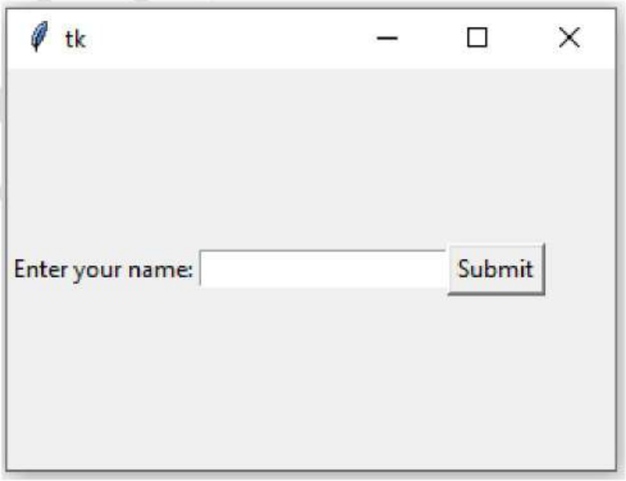
In Tkinter, there are three layout managers that can be used to arrange the widgets on the screen: pack, grid, and place.

1. pack: The pack layout manager places widgets in the available space in a horizontal or vertical direction. It is the simplest of the three managers and is often used for simple interfaces. The pack method takes several options, such as side to specify the side of the parent widget to pack the child widget and fill to specify how the widget should expand to fill available space.

Ex:

```
import tkinter as tk

root = tk.Tk()
root.geometry("200x200")

# Using pack layout manager
label1 = tk.Label(root, text="Enter your name:")
label1.pack(side="left")

entry1 = tk.Entry(root)
entry1.pack(side="left")

button1 = tk.Button(root, text="Submit")
button1.pack(side="left")

root.mainloop()
```

Output:

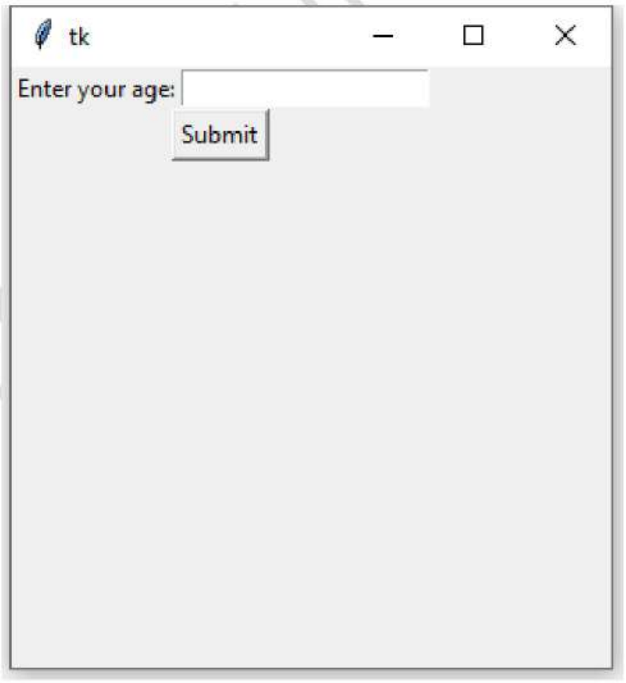


2. grid: The grid layout manager arranges widgets in a grid pattern with rows and columns. You can specify the number of rows and columns and then place the widgets in the cells using the grid method. The grid method takes several options, such as row and column to specify the cell to place the widget and sticky to specify how the widget should expand to fill the cell.

Ex:

```
import tkinter as tk

root = tk.Tk()
root.geometry("300x300")

# Using grid layout manager
label2 = tk.Label(root, text="Enter your age:")
label2.grid(row=0, column=0)

entry2 = tk.Entry(root)
entry2.grid(row=0, column=1)

button2 = tk.Button(root, text="Submit")
button2.grid(row=1, column=0, columnspan=2)

root.mainloop()
```

Output:

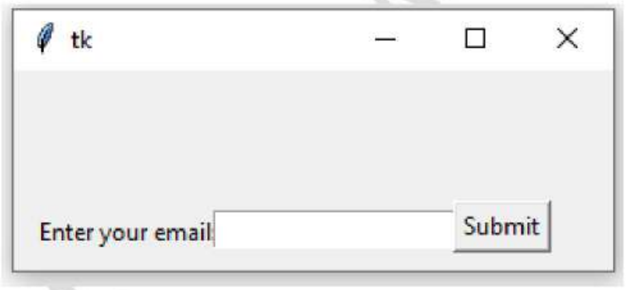


3. place: The place layout manager allows you to specify the exact location of a widget using x and y coordinates. You can also use relative positioning and offsets to position the widget. The place method takes several options, such as x and y to specify the coordinates of the widget and relx and rely to specify the relative position of the widget.

Ex:

```
import tkinter as tk

root = tk.Tk()
root.geometry("300x100")

# Using place layout manager
label3 = tk.Label(root, text="Enter your email:")
label3.place(x=10, y=70)

entry3 = tk.Entry(root)
entry3.place(x=100, y=70)

button3 = tk.Button(root, text="Submit")
button3.place(x=220, y=65)

root.mainloop()
```

Output:



Python SQLite:

SQLite is a lightweight, serverless, and self-contained SQL database engine that is widely used with Python.

SQLite is a software library that provides a relational database management system (RDBMS) for embedded and small-scale applications. It is a self-contained, serverless, zero-configuration, and transactional SQL database engine that is widely used in various applications such as mobile devices, desktop applications, and web browsers.

SQLite is designed to be fast, lightweight, and easy to use, with a small memory footprint and low CPU usage. It stores data in a single file, making it easy to distribute and manage, and supports many standard SQL features, such as transactions, indexes, triggers, and views.

SQLite is a popular choice for developers who need a local database solution that is easy to set up and use, and does not require a dedicated server or complex administration. It is also used as a testing and development database, and as a data storage format for many applications and operating systems.

Light-weight RDBMS software to manage data in file format.

Download link: https://sqlitebrowser.org/dl/

The SQLite3 module: The sqlite3 module is a built-in module in Python that provides a simple and easy-to-use interface for working with SQLite databases. It allows you to create, connect to, and manipulate SQLite databases using SQL commands.

To use the sqlite3 module in your Python code, you first need to import it by adding the following line at the beginning of your code:

```
import sqlite3
```

SQLite Methods - connect, cursor, execute, close;

The sqlite3 module in Python provides several methods for working with SQLite databases:

Connect to Database:

1. connect: connect() method: This method is used to create a connection to an SQLite database. It takes a single argument - the name of the database file - and returns a connection object that can be used to execute SQL commands.

If the database file does not exist, it will be created automatically.

Here's an example:

```
import sqlite3
# Create a connection to the database
conn = sqlite3.connect('example.db')
```

Cursor Object and its methods: A cursor object is used to interact with a database. A cursor is created by a database connection object, and it is used to execute SQL statements and retrieve data from the database.

2. cursor() method: This method is used to create a cursor object that can be used to execute SQL commands. It is called on a connection object and returns a cursor object. Here's an example:

```
# Create a cursor object
c = conn.cursor()
```

3. execute() method: This method is used to execute an SQL command. It takes a single argument - the SQL command - and returns a cursor object that can be used to fetch the results. Here's an example:

```
# Execute a query
c.execute('CREATE TABLE person (id integer, name text)')
```

4. fetchone() method: This method is used to fetch the next row of a query result set. It returns a tuple of values or None if there are no more rows to fetch. Here's an example:

```
# Fetch the next row
row = c.fetchone()
```

5. fetchall() method: This method is used to fetch all the rows of a query result set. It returns a list of tuples of values. Here's an example:

```
# Fetch all the rows
rows = c.fetchall()
```

6. commit() method: This method is used to save the changes made to the database. It is called on a connection object and commits the current transaction. Here's an example:

```
# Commit the changes
conn.commit()
```

7. close() method: This method is used to close the cursor and the connection. It is called on both the cursor object and the connection object. Here's an example:

```
# Close the cursor and the connection
c.close()
conn.close()
```

Operatins on Tables Create, Insert, Select, Update. Delete and Drop Records.

```
import sqlite3

# create a connection to a SQLite database
conn = sqlite3.connect('test.db')

# create a cursor object
cursor = conn.cursor()

# create a table
cursor.execute("CREATE TABLE employee (id INTEGER,name TEXT)")

# insert a record into the table
cursor.execute("INSERT INTO employee(id,name) VALUES (101, 'Ravi')")
cursor.execute("INSERT INTO employee(id,name) VALUES (102, 'Ravishankar')")
cursor.execute("INSERT INTO employee(id,name) VALUES (103, 'Ramesh')")

# select records from the table
cursor.execute("SELECT * FROM employee")
rows = cursor.fetchall()
for row in rows:
   print(row)

# update a record in the table
cursor.execute("update employee set name = 'Rakesh' where id = 101")

# delete a record from the table
cursor.execute("delete from employee where id = 103")

# drop the table
cursor.execute("DROP TABLE employee")

# commit the transaction
conn.commit()

# close the cursor and the connection
cursor.close()
conn.close()
```

Data Analysis: Python provides many powerful libraries for data analysis, including NumPy, Pandas, Matplotlib, and more.

NumPy- Introduction to NumPy: NumPy is a powerful Python library for numerical computing that is widely used in data analysis. NumPy is a popular Python library for numerical computing. It provides efficient implementations of many mathematical functions and operations on large arrays and matrices. Here are some of the key features of NumPy:

Multi-Dimensional Arrays: NumPy provides a powerful ndarray class for representing multi-dimensional arrays. These arrays can have any number of dimensions, and can be indexed and sliced in various ways.

Mathematical Operations: NumPy provides many built-in functions for performing mathematical operations on arrays. These functions are optimized for efficiency and can handle large arrays with ease. Some examples include np.add(), np.subtract(), np.multiply(), np.divide(), np.power(), np.sqrt(), and more.

Linear Algebra: NumPy provides a range of functions for linear algebra operations, including matrix multiplication, determinant calculation, eigenvalue and eigenvector calculation, and more. These functions are optimized for performance and can handle large matrices efficiently.

Random Number Generation: NumPy provides functions for generating random numbers from various probability distributions. These functions are useful for many statistical and simulation tasks.

Broadcasting: NumPy provides a powerful broadcasting feature that allows arrays with different shapes to be used in arithmetic operations. This can simplify many calculations and make them more efficient.

Array Creation using NumPy: NumPy provides several ways to create arrays in Python. Here are some of the most common ways to create arrays using NumPy:

1. np.array(): This function creates an array from a Python list or tuple.
Ex:
```
import numpy as np
a = np.array([1, 2, 3, 4, 5])
print(a)    # [1 2 3 4 5]
b = np.array((1, 2, 3))
print(b)    # [1 2 3]
```

2. np.zeros(): This function creates an array filled with zeros.

Ex:
```
import numpy as np
a = np.zeros(5)
print(a)    # [0. 0. 0. 0. 0.]
b = np.zeros((2, 3))
print(b)    # [[0. 0. 0.]
        #  [0. 0. 0.]]
```

3. np.arange(): This function creates an array with a range of values.

Ex:

```
import numpy as np
a = np.arange(1, 6)
print(a)    # [1 2 3 4 5]
b = np.arange(0, 1, 0.2)
print(b)    # [0.  0.2 0.4 0.6 0.8]
```

Operations on Arrays: NumPy is a Python library that provides support for large, multi-dimensional arrays and matrices, along with a range of mathematical functions to operate on them efficiently. Here are some of the commonly used operations on arrays using NumPy:

1. add() operation & addition: to add two arrays element-wise, you can use the numpy.add() function

Ex:
```
import numpy as np
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
c = np.add(a, b)
print(c) # Output: [5 7 9]
```

2. array indexing: You can access elements of an array using indexing. For example, to access the element at index 2 of an array arr, you can use arr[2].

3. array slicing: Array slicing is a technique used to extract a subset of elements from an array in Python. It is a way to create a new array by selecting a part of an existing array.

Ex:

```
import numpy as np

# Create a 1-dimensional array
a = np.array([1, 2, 3, 4, 5])
```

```
# Slice the array from index 1 to index 3 (exclusive)
b = a[1:3]

print(b)  # Output: [2 3]
```

4. reshape(): You can reshape an array using the numpy.reshape() function. For example, to reshape a 1-dimensional array to a 2-dimensional array with 2 rows and 3 columns, you can use:

Ex:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6])
new_arr = arr.reshape((2, 3))
print(new_arr) # Output: [[1 2 3]
                 #  [4 5 6]]
```

5. concatenate(): You can concatenate two or more arrays using the numpy.concatenate() function. For example, to concatenate two arrays a and b vertically, you can use:

Ex:

```
import numpy as np
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
c = np.concatenate((a, b), axis=0)
print(c) # Output: [[1 2]
      #     [3 4]
      #     [5 6]]
```

6. transpose(): You can transpose an array using the numpy.transpose() function. For example, to transpose a 2-dimensional array arr, you can use:

Ex:

```
import numpy as np
arr = np.array([[1, 2], [3, 4], [5, 6]])
new_arr = np.transpose(arr)
print(new_arr) # Output: [[1 3 5]
                 #      [2 4 6]]
```

7. mean(): You can perform statistical operations on arrays using NumPy. For example, to calculate the mean of an array arr,

Ex:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
mean = np.mean(arr)
print(mean) # Output: 3.0
```

8. sort(): You can sort the elements of an array using the numpy.sort() function. For example, to sort the elements of an array arr in ascending order, you can use:
Ex:

```
import numpy as np
arr = np.array([3, 1, 4, 2, 5])
sorted_arr = np.sort(arr)
print(sorted_arr) # Output: [1 2 3 4 5]
```

9. broadcasting: NumPy provides broadcasting, which is a powerful mechanism that allows you to perform arithmetic operations between arrays with different shapes. For example, to add a scalar value to an array arr, you can use:
Ex:

```
import numpy as np
arr = np.array([1, 2, 3])
new_arr = arr + 1
print(new_arr) # Output: [2 3 4]
```

Pandas- Introduction to Pandas: Pandas is a popular open-source data analysis and manipulation library in Python. It provides easy-to-use data structures and data analysis tools for handling and manipulating numerical tables and time-series data. Pandas is built on top of NumPy, which makes it fast and efficient for working with large datasets.

Series and DataFrames: The two primary data structures used in Pandas are Series and DataFrame.

Series: A one-dimensional array-like object that can hold data of any type. It consists of a sequence of values and an associated array of labels, which is called the index. The index can be of any type, including integers, strings, or dates.

DataFrame: A two-dimensional table-like data structure that consists of rows and columns. It is similar to a spreadsheet or a SQL table. Each column in a DataFrame can have a different data type, such as integer, float, or string. A DataFrame can also have row labels and column labels.

Pandas provides many tools for data analysis and manipulation, including:

- Data filtering, selection, and manipulation
- Data aggregation and grouping
- Handling missing data
- Merging and joining datasets
- Reshaping and pivoting data
- Time-series analysis and manipulation

Ex:

```
import pandas as pd

# Create a DataFra
me from a dictionary
data = {'name': ['Alice', 'Bob', 'Charlie', 'Dave'],
    'age': [25, 32, 18, 47],
    'city': ['New York', 'Paris', 'London', 'San Francisco']}
df = pd.DataFrame(data)

print(df)
```

Output:

```
     name  age           city
0    Alice   25       New York
1      Bob   32          Paris
2  Charlie   18         London
3     Dave   47  San Francisco
```

Creating DataFrames from Excel Sheet and .csv file, Dictionary and Tuples.

1. From excel data:

```
import pandas as pd

# Read data from multiple sheets in an Excel file
data = pd.read_excel('person_emp.xlsx', sheet_name=['person'])

# Create DataFrames from the data
df1 = data['person']

# Print the DataFrames
print(df1)
```

Output:

```
   NAME   AGE
0  Ravi    25
1   Raju   22
2 Rakesh   21
```

2. From dictionary: You can create a Pandas DataFrame from a dictionary in Python by using the pd.DataFrame() constructor. The dictionary keys become the column names, and the dictionary values become the column values. Here's an example:

Ex:

```
import pandas as pd

# Create a dictionary of data
data = {'regno': [1,2,3],'name': ['ramesh', 'suresh', 'rajesh']}


# Create a DataFrame from the dictionary
df = pd.DataFrame(data)

# Print the DataFrame
print(df)
```

Output:

```
   regno    name
0      1  ramesh
1      2  suresh
2      3  rajesh
```

2. From tuple: You can create a Pandas DataFrame from a list of tuples in Python by using the pd.DataFrame() constructor. Each tuple in the list represents a row in the DataFrame. Here's an example:

```
import pandas as pd

# Create a list of tuples
data = [('Rajesh', 25, 'New York'),
      ('Ramessh', 32, 'Paris')]

# Create a DataFrame from the list of tuples
df = pd.DataFrame(data, columns=['name', 'age', 'city'])

# Print the DataFrame
print(df)
```

Output:

```
     name  age      city
0   Rajesh   25  New York
1  Ramessh   32     Paris
```

Operations on DataFrames:

Selecting Columns: You can select one or more columns from a DataFrame using indexing. For example, df['column_name'] selects a single column, and df[['column_1', 'column_2']] selects multiple columns.

Filtering Rows: You can filter rows of a DataFrame using boolean indexing. For example, df[df['column_name'] > 10] filters rows where the value in column_name is greater than 10.

Sorting: You can sort a DataFrame by one or more columns using the sort_values() method. For example, df.sort_values(by='column_name') sorts the DataFrame by column_name.

Aggregation: You can compute summary statistics of a DataFrame using the agg() method. For example, df.agg({'column_name': 'mean'}) computes the mean of column_name.

Grouping: You can group a DataFrame by one or more columns using the groupby() method. For example, df.groupby('column_name').agg({'column_2': 'mean'}) groups the DataFrame by column_name and computes the mean of column_2 for each group.

Joining: You can join two or more DataFrames using the merge() method. For example, pd.merge(df1, df2, on='column_name') joins df1 and df2 on column_name.

Reshaping: You can reshape a DataFrame using the pivot() method. For example, df.pivot(index='column_1', columns='column_2', values='column_3') pivots the DataFrame so that column_1 becomes the index, column_2 becomes the columns, and column_3 becomes the values.

Data Visualization: Data visualization is the process of representing data in a visual format, such as charts, graphs, and maps, to help identify patterns, trends, and relationships within the data. In Python, there are several libraries available for data visualization, such as Matplotlib, Seaborn, Plotly, etc.

Introduction to Data Visualization: Data visualization can be used for exploratory data analysis, to better understand the data and identify interesting patterns and relationships.

- It can also be used for data communication, to effectively convey insights and findings to stakeholders.
- Python's data visualization libraries provide a wide range of customizable and interactive plots and charts that can be used to visualize different types of data, such as numerical, categorical, and time-series data. These libraries provide a high level of flexibility and customization, allowing users to create publication-quality visualizations.
- Some common types of plots that can be created in Python include line plots, scatter plots, bar plots, histograms, heatmaps, and more.

Matplotlib Library: Matplotlib is a plotting library in Python that is widely used for creating static, interactive, and publication-quality visualizations. It provides a wide range of plots, including line plots, scatter plots, bar plots, histograms, and more.

Different Types of Charts using Pyplot- Line chart, Bar chart and Histogram and Pie chart.

Matplotlib's **pyplot** module provides a wide range of charts and plots that can be used for data visualization. Here are some of the most commonly used types of charts:

Line Chart: A line chart is used to plot a series of data points connected by straight lines. It is useful for visualizing trends and changes over time.

Bar Chart: A bar chart is used to compare values between different categories. It is useful for visualizing categorical data, such as the number of students in different grades.

Pie Chart: A pie chart is used to show the proportion of different categories in a dataset. It is useful for visualizing percentages and proportions.

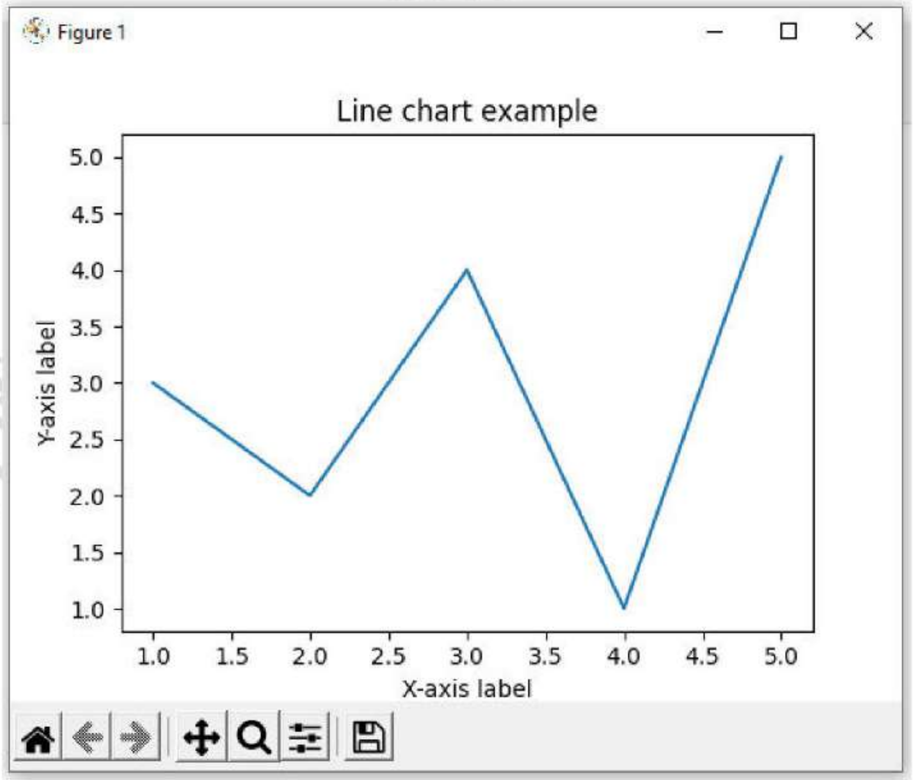
Histogram: A histogram is used to show the distribution of a dataset. It is useful for visualizing the frequency of data values.

1. Line chart: In this example, we first import the pyplot module from the matplotlib library. We then define the x and y data arrays, which represent the x and y values of the data points in the line chart.

Ex:

```
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y = [3, 2, 4, 1, 5]

# Create line chart
plt.plot(x, y)

# Add labels and title
plt.xlabel('X-axis label')
plt.ylabel('Y-axis label')
plt.title('Line chart example')

# Show the plot
plt.show()
```

Output:



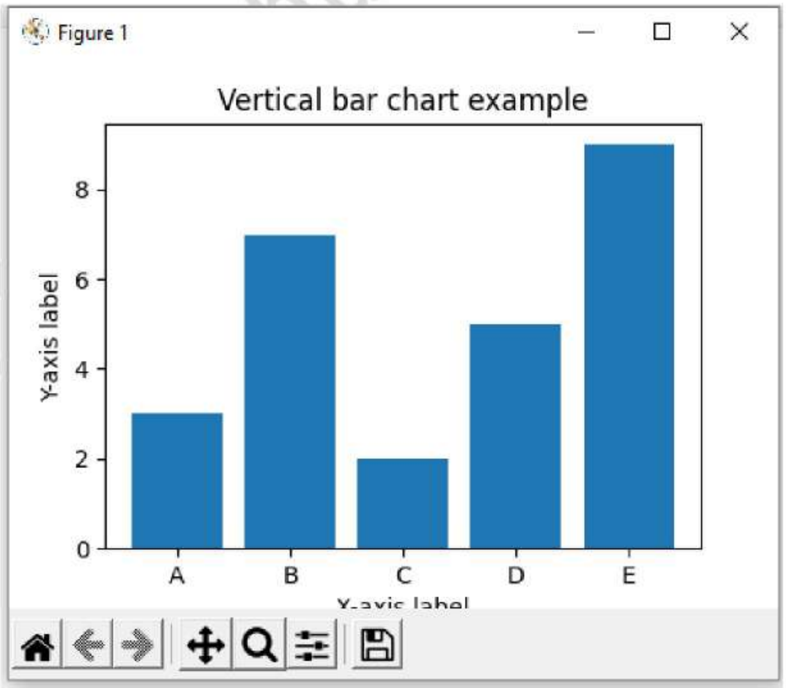
2. Bar chart: In this example, we first import the pyplot module from the matplotlib library. We then define the x and y data arrays, which represent the categories and values for the bar chart.

Next, we use the plt.bar() function to create a vertical bar chart. The plt.bar() function takes the x and y data arrays as arguments and creates a bar chart with the categories on the x-axis and the values on the y-axis.

Ex:

```
import matplotlib.pyplot as plt

# Sample data
x = ['A', 'B', 'C', 'D', 'E']
y = [3, 7, 2, 5, 9]

# Create vertical bar chart
plt.bar(x, y)

# Add labels and title
plt.xlabel('X-axis label')
plt.ylabel('Y-axis label')
plt.title('Vertical bar chart example')

# Show the plot
plt.show()
```

Output:



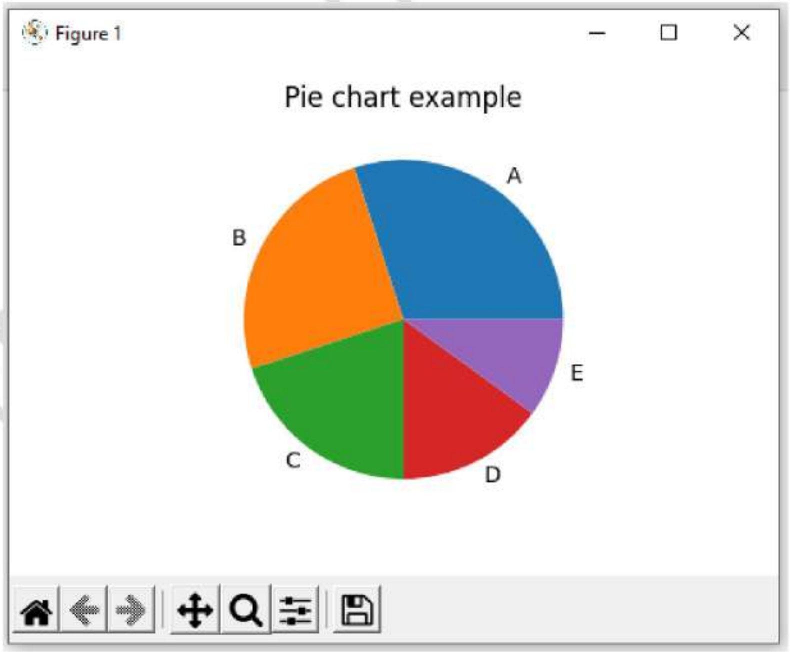
3. Pie chart: In this example, we first import the pyplot module from the matplotlib library. We then define the sizes and labels lists, which represent the sizes and labels of the pie chart slices.

Next, we use the plt.pie() function to create a pie chart. The plt.pie() function takes the sizes and labels lists as arguments and creates a pie chart with the slices labeled with the labels.

Ex:

```
import matplotlib.pyplot as plt

# Sample data
sizes = [30, 25, 20, 15, 10]
labels = ['A', 'B', 'C', 'D', 'E']

# Create pie chart
plt.pie(sizes, labels=labels)

# Add title
plt.title('Pie chart example')

# Show the plot
plt.show()
```

Output:



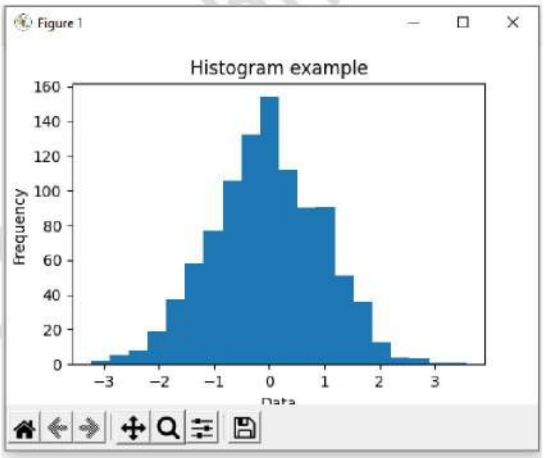
4. Histogram: In this example, we first import the pyplot module from the matplotlib library and the numpy library for generating random sample data. We then use the np.random.normal() function to generate a sample of 1000 data points from a normal distribution with mean 0 and standard deviation 1.

Next, we use the plt.hist() function to create a histogram chart. The plt.hist() function takes the data array as an argument and creates a histogram chart with 20 bins by default.

Ex:

```
import matplotlib.pyplot as plt
import numpy as np

# Generate sample data
np.random.seed(123)
data = np.random.normal(0, 1, size=1000)

# Create histogram
plt.hist(data, bins=20)

# Add labels and title
plt.xlabel('Data')
plt.ylabel('Frequency')
plt.title('Histogram example')

# Show the plot
plt.show()
```

Output:



Note: Refer all python Lab journal programs on this unit.