PYTHON LIBRARIES

In Python, a library is a collection of pre-written code that allows developers to perform common tasks more easily and efficiently. Libraries in Python are used to provide additional functionality to the base language.

Python has a vast ecosystem of libraries and frameworks that cover almost every domain of software development. Some of the most popular libraries in Python include:

1. support for building and training deep neural networks.
2. Scikit-learn - provides support for various machine learning algorithms, including classification, regression, and clustering.
3. Django - provides support for building web applications and APIs.
4. Flask - provides a lightweight NumPy - provides support for large, multi-dimensional arrays and matrices, as well as a large collection of mathematical functions.
5. Pandas - provides support for data analysis and manipulation, including reading and writing data from various sources.
6. Matplotlib - provides support for creating static, animated, and interactive visualizations of data.
7. TensorFlow - provides web framework for building web applications and APIs.
8. Pygame - provides support for creating games and multimedia applications.

These libraries can be installed using a package manager like pip and can be imported into a Python project to provide additional functionality. Let's explain them further below;

1.NumPY

NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

Here is a simple example that demonstrates some of the functionality of NumPy:

```
import numpy as np

# create a 1-dimensional NumPy array

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

```
# create a 2-dimensional NumPy array

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


# mean of the array

print("Mean of a: ", a.mean())

print("Mean of b: ", b.mean())


# sum of the elements in the array

print("Sum of a: ", a.sum())

print("Sum of b: ", b.sum())


# standard deviation of the array

print("Standard deviation of a: ", a.std())

print("Standard deviation of b: ", b.std())
```

here is the output;

```
Mean of a:  3.0

Mean of b:  3.5

Sum of a:  15

Sum of b:  21

Standard deviation of a:  1.5811388300841898

Standard deviation of b:  1.707825127659933
```

Here's a simple example showing how to create and manipulate arrays in NumPy:

```
import numpy as np
```

```
# create a 1-dimensional array

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


# create a 2-dimensional array

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


# access elements

print("a[0]:", a[0])

print("b[0, 0]:", b[0, 0])


# shape of an array

print("a shape:", a.shape)

print("b shape:", b.shape)


# reshape an array

c = b.reshape(4,)

print("c shape:", c.shape)
```

In this example, we first import NumPy under the **np** alias. We then create two arrays **a** and **b**, which are both 1-dimensional and 2-dimensional arrays respectively. The **shape** property of an array returns its dimensions. Finally, we use the **reshape** method to change the shape of **b** into a 1-dimensional array and assign it to **c**.

2. PANDAS

Pandas is a popular Python library for data manipulation and analysis. It provides data structures for efficiently storing large datasets and tools for working with them.

Here's a simple example that demonstrates how to create and manipulate a Pandas DataFrame:

```
import pandas as pd


# create a DataFrame from a dictionary
data = {'Name': ['John', 'Jane', 'Jim', 'Joan'],
        'Age': [32, 28, 40, 38],
        'City': ['New York', 'London', 'Paris', 'Berlin']}
df = pd.DataFrame(data)
# display the first 5 rows of the DataFrame
print(df.head())
# select a column
print(df['Name'])
# select multiple columns
print(df[['Name', 'City']])
# filter rows based on a condition
print(df[df['Age'] > 35])


# apply a function to a column
df['Age_Plus_One'] = df['Age'].
```

In this example, we first import Pandas under the **pd** alias. We then create a DataFrame **df** from a dictionary **data**, where the keys of the dictionary correspond to column names and the values correspond to the data in each column. The **head** method displays the first 5 rows of the DataFrame, while selecting a single or multiple columns can be done by indexing the DataFrame with the column names in square brackets. We can also filter rows based on a condition, and apply a function to a column using the **apply** method. Finally, we create a new column **Age_Plus_One** by adding 1 to each value in the **Age** column.

3. Matplotlib

Matplotlib is a plotting library for the Python programming language. It allows users to create a wide range of static, animated, and interactive visualizations in Python.

Here is a simple example of how to use Matplotlib to create a line plot:

```
import matplotlib.pyplot as plt

# data to plot
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

# plot data
plt.plot(x, y)

# add labels and title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Line Plot')

# show plot
plt.show()
```

This will create a line plot of the data in **x** and **y**, with labeled X and Y axis and a title.

You can also create other types of plots like bar plots, scatter plots, histograms, etc.

```
import matplotlib.pyplot as plt
```

```
# data to plot

x = [1, 2, 3, 4, 5]

y = [2, 4, 6, 8, 10]


# plot data

plt.bar(x, y)


# add labels and title

plt.xlabel('X-axis')

plt.ylabel('Y-axis')

plt.title('Bar Plot')


# show plot

plt.show()
```

This will create a bar plot of the data in **x** and **y**, with labeled X and Y axis and a title.

4.seaborn

Seaborn is a data visualization library in Python that is based on Matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics. It is designed to work well with Pandas dataframes and NumPy arrays and provides functions to visualize univariate and bivariate distributions, linear relationships, and aggregate statistical models.

Here is a simple example that shows how to use Seaborn to create a scatter plot with regression line:

```
import seaborn as sns

import matplotlib.pyplot as plt

import numpy as np

import pandas as pd
```

```
np.random.seed(0)

x = np.random.normal(0, 1, 100)

y = x + np.random.normal(0, 0.5, 100)


df = pd.DataFrame({'x': x, 'y': y})


sns.regplot(x='x', y='y', data=df)

plt.show()
```

This will generate a scatter plot with a regression line that shows the relationship between **x** and **y**. The **sns.regplot** function fits a linear regression model to the data and plots the regression line with 95% confidence interval as shaded area.

Seaborn provides many other functions for visualizing data, such as histograms, bar plots, violin plots, box plots, and so on. You can easily customize the appearance of plots with the help of built-in themes and color palettes.

5.TensorFlow

TensorFlow is a popular open-source software library for data flow and differentiable programming. It was developed by the Google Brain team and is used for a wide range of applications such as building and training machine learning models, deep neural networks, and natural language processing.

Here's a simple example in Python to demonstrate how TensorFlow works:

```
import tensorflow as tf


# Define a constant tensor

a = tf.constant(2)

b = tf.constant(3)
```

```
# Define an operation
c = a + b

# Start a TensorFlow session to evaluate the tensor
sess = tf.Session()
result = sess.run(c)
print(result)

# Output: 5
```

In the above example, we first import TensorFlow and then create two constant tensors **a** and **b** with values 2 and 3, respectively. We then define an operation to add the two tensors and assign the result to **c**. To evaluate the tensor, we start a TensorFlow session **sess** and use the **run** method to evaluate the tensor **c**. Finally, we print the result, which is **5**.

This is just a basic example to demonstrate how TensorFlow works, but it can be used for much more complex operations and models. TensorFlow provides a rich set of tools for building and training machine learning models, and it is widely used in both academia and industry.

6.Keras

Keras is a high-level neural network API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed to make it easier to develop, train, and evaluate deep learning models. With Keras, you can build complex deep learning models with just a few lines of code.

Here's a simple example in Python to demonstrate how Keras works:

```
import numpy as np
from tensorflow import keras
```

```
# Define the model
model = keras.Sequential([
    keras.layers.Dense(units=1, input_shape=[1])
])

# Compile the model
model.compile(optimizer=keras.optimizers.Adam(1), loss='mean_squared_error')

# Generate some fake data for training
xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)

# Train the model
model.fit(xs, ys, epochs=500)

# Make predictions
print(model.predict([10.0]))

# Output: [[18.982435]]
```

7. PyTorch.

PyTorch is an open-source machine learning library for Python, based on Torch, used for applications such as computer vision and natural language processing. It provides a simple and flexible way to build and train deep learning models.

Here's a simple example in Python to demonstrate how PyTorch works:

In the above example, we first import PyTorch and then create a tensor **x** with a single value **5.0**. We set the **requires_grad** flag to **True** to indicate that we want to compute gradients with

respect to this tensor. We then define an operation to square the tensor **x** and assign the result to **y**.

Next, we use the **backward** method to compute the gradients with respect to the tensor **x**. Finally, we print the gradient, which is **tensor([10.])**.

```
import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import DataLoader

from torchvision import datasets, transforms



# Set device to use

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")



# Define transform to apply to data

transform = transforms.Compose([

    transforms.ToTensor(),

    transforms.Normalize((0.1307,), (0.3081,))

])



# Load training and testing datasets

train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)

test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)
```

```
# Create data loaders
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=True)


# Define the neural network model
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout2d(0.25)
        self.dropout2 = nn.Dropout2d(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = nn.functional.relu(x)
        x = self.conv2(x)
        x = nn.functional.relu(x)
```

```
        x = nn.functional.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = nn.functional.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = nn.functional.log_softmax(x, dim=1)
        return output

model = Net().to(device)

# Define the optimizer and loss function
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
criterion = nn.CrossEntropyLoss()

# Train the model
for epoch in range(10):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
```

```
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % 100 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))


# Test the model
model.eval()
test_loss = 0
correct = 0
with torch.no_grad():
    for data, target in test_loader:
        data, target = data.to(device), target.to(device)
        output = model(data)
        test_loss += criterion(output, target).item()
        pred = output.argmax(dim=1, keepdim=True)
        correct += pred.eq(target.view_as(pred)).sum().item()
```

```
test_loss /= len(test_loader.dataset)

print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(

    test_loss, correct, len(test_loader.dataset),

    100. * correct
```

This is just a simple example to demonstrate how PyTorch works, but it can be used for much more complex operations and models. With its dynamic computational graph and support for hardware acceleration, PyTorch provides a fast and flexible platform for building and training deep learning models.

8. Scikit-learn

Scikit-learn is a popular machine learning library in Python, used for a variety of applications, such as regression, classification, clustering, and dimensionality reduction. It provides a simple and efficient tool for machine learning, including pre-processing, model selection, and evaluation.

Here's a simple example in Python to demonstrate how scikit-learn works:

```
import numpy as np

from sklearn import linear_model


# Generate some fake data for training

xs = np.array([[1], [2], [3], [4]], dtype=float)

ys = np.array([1, 2, 3, 4], dtype=float)


# Define the model

model = linear_model.LinearRegression()
```

```
# Train the model
model.fit(xs, ys)


# Make predictions
print(model.predict([[5]]))


# Output: [5.]
```

In the above example, we first import NumPy and scikit-learn. We then generate some fake data for training and use the **LinearRegression** class from scikit-learn to define a linear regression model. We train the model using the **fit** method and then make a prediction for an input of **[5]**. Finally, we print the prediction, which is **[5.]**.

This is just a simple example, but scikit-learn provides a wide range of algorithms and tools for machine learning, including pre-processing, model selection, and evaluation, making it a popular choice for many machine learning tasks.

9.scipy

SciPy is a Python library for scientific computing, which builds on NumPy and provides a range of algorithms for optimization, signal processing, linear algebra, and statistics. It provides efficient and easy-to-use functions for many common tasks in scientific computing.

Here's a simple example in Python to demonstrate how SciPy works:

```
import numpy as np
from scipy import optimize


# Define a function to optimize
def f(x):
    return x**2 + 10*np.sin(x)


# Use the BFGS optimization algorithm to find the minimum of the function
```

```
result = optimize.minimize(f, x0=0, method='BFGS')


# Print the minimum value and the location at which it occurs

print(result.fun)

print(result.x)


# Output:

# 8.141802202235456

# [-1.30644001]
```

In the above example, we first import NumPy and SciPy. We then define a simple function **f(x) = x^2 + 10*sin(x)** that we want to optimize. We use the **minimize** function from the **optimize** module in SciPy to find the minimum of the function. We pass in the function **f** and an initial guess for the location of the minimum **x0=0**, and we specify the optimization algorithm **method='BFGS'**. Finally, we print the minimum value and the location at which it occurs, which are **8.141802202235456** and **[-1.30644001]**, respectively.

This is just a simple example to demonstrate how SciPy works, but it can be used for much more complex optimization problems, as well as for other tasks such as signal processing, linear algebra, and statistics.

10.NLtk

NLTK (Natural Language Toolkit) is a popular Python library for natural language processing. It provides a suite of tools and resources for tasks such as tokenization, stemming, and tagging, making it a comprehensive toolkit for natural language processing.

Here's a simple example in Python to demonstrate how NLTK works:

```
import nltk


# Download the required data and resources

nltk.download('punkt')


# Tokenize a sentence into words

sentence = "This is a simple sentence."
```

```
words = nltk.word_tokenize(sentence)

print(words)



# Output: ['This', 'is', 'a', 'simple', 'sentence', '.']
```

In the above example, we first import NLTK and download the required data and resources. We then tokenize a sentence into words using the **word_tokenize** function, which splits the sentence into individual words. Finally, we print the resulting list of words, which is **['This', 'is', 'a', 'simple', 'sentence', '.']**.

This is just a simple example to demonstrate how NLTK works, but it can be used for much more complex natural language processing tasks, such as part-of-speech tagging, named entity recognition, and sentiment analysis. With its wide range of tools and resources, NLTK is a popular choice for natural language processing tasks in Python.

11. Flask

Flask is a popular micro-web framework in Python for building web applications. It provides a simple and lightweight way to build and serve web applications, making it a great choice for small- to medium-sized projects.

Here's a simple example in Python to demonstrate how Flask works:

```
from flask import Flask



# Create a new Flask application

app = Flask(__name__)



# Define a simple route

@app.route('/')

def hello():

    return "Hello, World!"



# Run the application

if __name__ == '__main__':
```

```
app.run()
```

In the above example, we first import Flask. We then create a new Flask application using the **Flask** class and assign it to the variable **app**. We define a simple route using the **route** decorator and a function that returns the string "Hello, World!". Finally, we run the application by checking if the script is being run as the main program, and if so, calling the **run** method on the **app** object.

When you run this code, Flask will start a web server and you can access the "Hello, World!" message by visiting **http://localhost:5000/** in your web browser.

This is just a simple example to demonstrate how Flask works, but it can be used to build much more complex web applications, with support for templates, database integration, and more. With its simple and lightweight approach, Flask is a popular choice for web development in Python.

12. Django

Django is a high-level web framework in Python for building web applications. It provides a full-featured and robust framework for building complex web applications, making it a great choice for large-scale projects.

Here's a simple example in Python to demonstrate how Django works:

```
# First, create a new Django project

$ django-admin startproject myproject



# Change into the new project directory

$ cd myproject



# Create a new Django app

$ python manage.py startapp myapp



# Edit myapp/views.py

from django.http import HttpResponse



def hello(request):
```

```
    return HttpResponse("Hello, World!")


# Edit myapp/urls.py

from django.urls import path

from . import views


urlpatterns = [

    path('', views.hello, name='hello'),

]


# Edit myproject/urls.py

from django.urls import include, path


urlpatterns = [

    path('', include('myapp.urls')),

]


# Run the development server

$ python manage.py runserver
```

In the above example, we first create a new Django project using the **django-admin** command. We then create a new Django app using the **startapp** management command. We edit the **views.py** file to define a simple view that returns the string "Hello, World!" as an HTTP response. We then define a URL pattern for the view in the **urls.py** file for the app, and include that pattern in the project's **urls.py** file. Finally, we run the development server using the **runserver** management command.

When you run this code, Django will start a web server and you can access the "Hello, World!" message by visiting **http://localhost:8000/** in your web browser.

This is just a simple example to demonstrate how Django works, but it can be used to build much more complex web applications, with support for models, templates, forms, and more. With its full-featured and robust framework, Django is a popular choice for web development in Python.

13. Requests

The **requests** library in Python is a popular library for making HTTP requests to RESTful web APIs. It provides a simple and easy-to-use interface for making HTTP requests, abstracting away many of the low-level details of HTTP.

Here's a simple example in Python to demonstrate how **requests** works:

```
import requests



# Make a GET request to a URL

response = requests.get('https://httpbin.org/get')



# Check the status code of the response

if response.status_code == 200:

    # If the response was successful, print the content of the response

    print(response.content)

else:

    # If the response was not successful, print an error message

    print("Request failed with status code:", response.status_code)
```

In the above example, we first import the **requests** library. We then use the **get** method from the **requests** module to make a GET request to the URL **https://httpbin.org/get**. We check the status code of the response to make sure it was successful (status code 200), and if so, we print the content of the response. If the response was not successful, we print an error message with the status code of the response.

This is just a simple example to demonstrate how **requests** works, but it can be used to make more complex HTTP requests, including POST requests with data, handling cookies and headers, and more. With its simple and easy-to-use interface, **requests** is a popular choice for making HTTP requests in Python.

14. BeautifulSoup

Beautiful Soup is a Python library that is used for web scraping purposes to pull the data out of HTML and XML files. It creates a parse tree from page source code that can be used to extract data in a hierarchical and more readable manner.

Here's an example of how you can use Beautiful Soup to extract data from a web page:

```
import requests
from bs4 import BeautifulSoup

# Make a request to the website
url = 'https://www.example.com/page-with-data'
page = requests.get(url)

# Use Beautiful Soup to parse the page
soup = BeautifulSoup(page.content, 'html.parser')

# Extract data
data = soup.find('div', {'class': 'data-container'})
print(data.text)
```

In this example, we first make a request to the website using the **requests** library and then parse the page using Beautiful Soup's **BeautifulSoup** function. We pass in the **page.content** as the source of the parse tree and specify **html.parser** as the parser we want to use.

Once the parse tree is created, we use the **find** method to search for a **div** element with the class **data-container**. The **find** method returns the first element that matches the search criteria. In this case, it returns the first **div** with the class **data-container**. Finally, we use the **text** attribute of the **data** variable to extract the text content inside the **div**.

Note that this is just a simple example, and you can use Beautiful Soup to extract any type of data from a web page, including links, images, and more.

15. OpenCV

OpenCV (Open Source Computer Vision Library) is a library of programming functions mainly used for real-time computer vision. It is an open-source library and provides a Python interface.

Here is a simple example that shows how to use OpenCV in Python to read an image and display it on the screen.

```
import cv2

# Read an image using imread() function
img = cv2.imread("image.jpg")
```

```
# Display the image using imshow() function
cv2.imshow("Image", img)

# Wait until a key is pressed using waitKey() function
cv2.waitKey(0)

# Destroy all windows using destroyAllWindows() function
cv2.destroyAllWindows()
```

In the above code, we first import the cv2 module, then use the **imread** function to read an image file. The **imshow** function is used to display the image on the screen. The **waitKey** function is used to wait until a key is pressed. Finally, the **destroyAllWindows** function is used to destroy all windows.

16.Pygame

Pygame is a set of Python modules designed for writing video games. It is an open-source library that provides functionalities to create 2D games and multimedia applications.

Here is an example of a simple game in Pygame that involves moving a rectangle across the screen:

```
import pygame

# Initialize Pygame
pygame.init()

# Set the window size
window_size = (400, 300)

# Create the window
screen = pygame.display.set_mode(window_size)

# Set the title of the window
pygame.display.set_caption("Pygame Example")
```

```
# Define the colors
white = (255, 255, 255)
black = (0, 0, 0)

# Create the rectangle
rect = pygame.Rect(0, 0, 50, 50)

# Set the speed of the rectangle
rect_speed = 5

# Start the game loop
running = True
while running:
    # Handle events
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    # Move the rectangle
    rect.x += rect_speed
    rect.y += rect_speed

    # Check if the rectangle has reached the edge of the screen
    if rect.right > window_size[0] or rect.left < 0:
        rect_speed = -rect_speed
    if rect.bottom > window_size[1] or rect.top < 0:
        rect_speed = -rect_speed

    # Fill the screen with black
    screen.fill(black)
```

```
    # Draw the rectangle on the screen
    pygame.draw.rect(screen, white, rect)

    # Update the screen
    pygame.display.update()

# Quit Pygame
pygame.quit()
```

In this example, we first initialize Pygame using **pygame.init()**. Then, we create a window of size **(400, 300)** using **pygame.display.set_mode()**. We set the title of the window using **pygame.display.set_caption()**.

Next, we define two colors - white and black - using RGB values. We then create a rectangle using **pygame.Rect** and set its speed using the **rect_speed** variable.

The game loop starts with **running = True** and continues until **running** becomes **False**. In each iteration of the loop, we handle events using **pygame.event.get()** and check if the **QUIT** event has been triggered, in which case we set **running** to **False** to exit the loop.

Next, we move the rectangle by adding **rect_speed** to its **x** and **y** coordinates. We then check if the rectangle has reached the edge of the screen and change its speed if necessary.

Finally, we fill the screen with black, draw the rectangle on the screen in white, and update the screen using **pygame.display.update()**. After the game loop ends, we quit Pygame using **pygame.quit()**.

This is just a simple example of what you can do with Pygame. You can use this library to create much more complex and interactive games.

17. PyQt

PyQt is a set of Python bindings for the Qt application framework and runs on all platforms supported by Qt including Windows, OS X, Linux, iOS, and Android. PyQt5 is the most current version of PyQt.

Here's a simple example of how you can use PyQt5 to create a graphical user interface (GUI) in Python:

```
import sys
from PyQt5.QtWidgets import QApplication, QMainWindow, QPushButton, QTextEdit

app = QApplication(sys.argv)
```

```
window = QMainWindow()
window.setWindowTitle("PyQt Example")

text_edit = QTextEdit(window)
text_edit.setText("Hello PyQt")

button = QPushButton("Click Me", window)
button.clicked.connect(lambda: text_edit.setText("Button was clicked"))

window.show()

sys.exit(app.exec_())
```

This example creates a main window with a QTextEdit widget and a QPushButton widget. The button's **clicked** signal is connected to a lambda function that sets the text in the text edit widget. The **app.exec_()** method is called to start the application and display the main window.

18.pygtk

PyGTK is a set of Python bindings for the GTK+ GUI toolkit. It allows you to create graphical user interfaces (GUIs) for your Python applications using the GTK+ library.

Here's a simple example of how you can use PyGTK to create a GUI in Python:

```
import gtk

def on_button_clicked(button):
    print("Button was clicked")

window = gtk.Window(gtk.WINDOW_TOPLEVEL)
window.connect("delete_event", gtk.main_quit)
window.set_title("PyGTK Example")

vbox = gtk.VBox()
window.add(vbox)

button = gtk.Button("Click Me")
```

```
button.connect("clicked", on_button_clicked)

vbox.pack_start(button)


window.show_all()

gtk.main()
```

This example creates a main window with a single button. The button's **clicked** signal is connected to the **on_button_clicked** function that simply prints a message to the console. The **gtk.main()** method is called to start the GTK+ main loop and display the main window. When the main window is closed, the **gtk.main_quit** function is called to exit the main loop and terminate the application.

19.matplotlib

Matplotlib is a 2D plotting library for Python. It allows you to create a wide range of static, animated, and interactive visualizations in Python.

Here's a simple example of how you can use Matplotlib to create a line plot in Python:

```
import matplotlib.pyplot as plt


x = [1, 2, 3, 4, 5]

y = [2, 4, 6, 8, 10]


plt.plot(x, y)

plt.xlabel("X Axis")

plt.ylabel("Y Axis")

plt.title("Matplotlib Line Plot Example")

plt.show()
```

This example creates a line plot of the (x, y) data points. The **xlabel**, **ylabel**, and **title** functions are used to add labels to the x-axis, y-axis, and plot title, respectively. The **show** function is called to display the plot.

Note: You may need to install Matplotlib in your environment before you can use it. You can install it by running **pip install matplotlib**.

19.plotly

Plotly is an interactive, open-source, and browser-based graphing library for Python. It allows you to create a wide range of interactive visualizations, including line plots, scatter plots, bar plots, pie charts, 3D plots, and more.

Here's a simple example of how you can use Plotly to create a line plot in Python:

```
import plotly.express as px

import pandas as pd


df = pd.DataFrame({

    "x": [1, 2, 3, 4, 5],

    "y": [2, 4, 6, 8, 10]

})


fig = px.line(df, x="x", y="y", title="Plotly Line Plot Example")

fig.show()
```

This example creates a line plot of the (x, y) data points stored in a Pandas DataFrame. The **px.line** function is used to create the line plot, and the **x** and **y** arguments specify the column names for the x-axis and y-axis data. The **title** argument is used to specify the plot title. The **show** function is called to display the plot.

Note: You may need to install Plotly in your environment before you can use it. You can install it by running **pip install plotly**.

These might not be all of python liberaries…

DECIMAL…