Data Visualization – VTU QP Solution

1.

a.

The Importance of Data Visualization

Instead of just looking at data in the columns of an Excel spreadsheet, we get a better idea of what our data contains by using visualization. For instance, it's easy to see a pattern emerge from the numerical data that's given in the following scatter plot. It shows the correlation between body mass and the maximum longevity of various animals grouped by class. There is a positive correlation between body mass and maximum longevity:

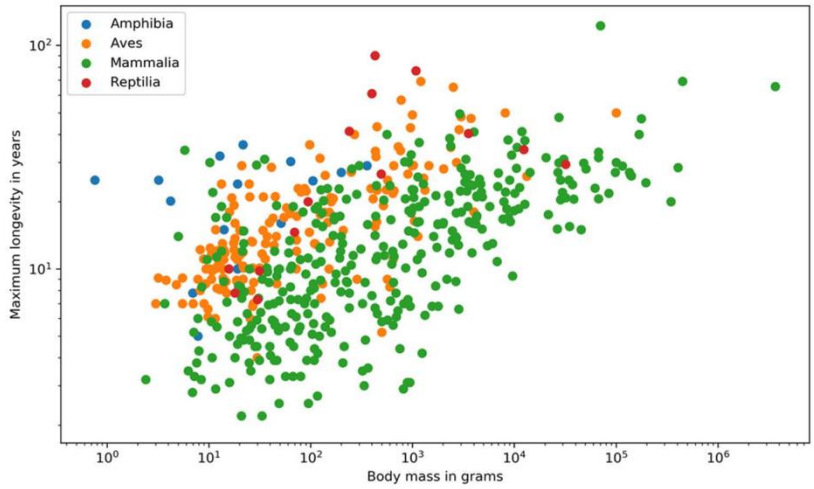


Figure 1.1: A simple example of data visualization

---

b.

Data Wrangling

Data wrangling is the process of transforming raw data into a suitable representation for various tasks. It is the discipline of augmenting, cleaning, filtering, standardizing, and enriching data in a way that allows it to be used in a downstream task, which in our case is data visualization.

Look at the following data wrangling process flow diagram to understand how accurate and actionable data can be obtained for business analysts to work on. The following steps explain the flow of the data wrangling process:

1. First, the Employee Engagement data is in its raw form.
2. Then, the data gets imported as a DataFrame and is later cleaned.
3. The cleaned data is then transformed into graphs, from which findings can be derived.
4. Finally, we analyze this data to communicate the final results.

For example, employee engagement can be measured based on raw data gathered from feedback surveys, employee tenure, exit interviews, one-on-one meetings, and so on. This data is cleaned and made into graphs based on parameters such as referrals, faith in leadership, and scope of promotions. The percentages, that is, information derived from the graphs, help us reach our result, which is to determine the measure of employee engagement:

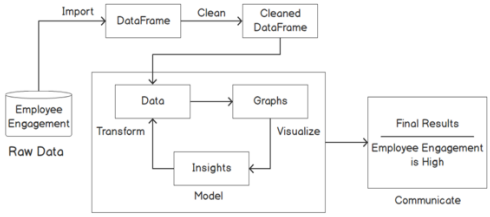


Figure 1.2: Data wrangling process to measure employee engagement

c.

The different measures of dispersion are as follows:

- **Variance**: The variance is the expected value of the squared deviation from the mean. It describes how far a set of numbers is spread out from their mean. Variance is calculated as follows:

$$Var(X) = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2$$

Figure 1.6: Formula for mean

- **Standard deviation**: This is the square root of the variance.
- **Range**: This is the difference between the largest and smallest values in a dataset.
- **Interquartile range**: Also called the **midspread** or **middle 50%**, this is the difference between the 75th and 25th percentiles, or between the upper and lower quartiles.

2.

a.

Basic NumPy Operations

In this section, we will learn about basic NumPy operations such as indexing, slicing, splitting, and iterating and implement them in an exercise.

Indexing

Indexing elements in a NumPy array, at a high level, works the same as with built-in Python lists. Therefore, we can index elements in multi-dimensional matrices:

```
dataset[0]       # index single element in outermost dimension
dataset[-1]      # index in reversed order in outermost dimension
dataset[1, 1]    # index single element in two-dimensional data
dataset[-1, -1]  # index in reversed order in two-dimensional data
```

Slicing

Slicing has also been adapted from Python's lists. Being able to easily slice parts of lists into new ndarrays is very helpful when handling large amounts of data:

```
dataset[1:3]         # rows 1 and 2
dataset[:2, :2]      # 2x2 subset of the data
dataset[-1, ::-1]    # last row with elements reversed
dataset[-5:-1, :6:2] # last 4 rows, every other element up to index 6
```

Splitting

Splitting data can be helpful in many situations, from plotting only half of your time-series data to separating test and training data for machine learning algorithms.

There are two ways of splitting your data, horizontally and vertically. Horizontal splitting can be done with the **`hsplit`** method. Vertical splitting can be done with the **`vsplit`** method:

```
np.hsplit(dataset, (3))  # split horizontally in 3 equal lists
np.vsplit(dataset, (2))  # split vertically in 2 equal lists
```

Sorting

Sorting each row of a dataset can be really useful. Using NumPy, we are also able to sort on other dimensions, such as columns.

In addition, **`argsort`** gives us the possibility to get a list of indices, which would result in a sorted list:

```
np.sort(dataset)          # values sorted on last axis
np.sort(dataset, axis=0)  # values sorted on axis 0
np.argsort(dataset)       # indices of values in sorted list
```

Combining

Stacking rows and columns onto an existing dataset can be helpful when you have two datasets of the same dimension saved to different files.

Given two datasets, we use **vstack** to stack **`dataset_1`** on top of **`dataset_2`**, which will give us a combined dataset with all the rows from **`dataset_1`**, followed by all the rows from **`dataset_2`**.

If we use **hstack**, we stack our datasets "next to each other," meaning that the elements from the first row of **`dataset_1`** will be followed by the elements of the first row of **`dataset_2`**. This will be applied to each row:

```
np.vstack([dataset_1, dataset_2])        # combine datasets vertically
np.hstack([dataset_1, dataset_2])        # combine datasets horizontally
np.stack([dataset_1, dataset_2], axis=0) # combine datasets on axis 0
```

---

b.

Advantages of pandas over NumPy

The following are some of the advantages of pandas:

- **High level of abstraction**: pandas have a higher abstraction level than NumPy, which gives it a simpler interface for users to interact with. It abstracts away some of the more complex concepts, such as high-performance matrix multiplications and joining tables, and makes it easier to use and understand.
- **Less intuition**: Many methods, such as joining, selecting, and loading files, are used without much intuition and without taking away much of the powerful nature of pandas.
- **Faster processing**: The internal representation of DataFrames allows faster processing for some operations. Of course, this always depends on the data and its structure.
- **Easy DataFrame design**: DataFrames are designed for operations with and on large datasets.

Disadvantages of pandas

The following are some of the disadvantages of pandas:

- **Less applicable**: Due to its higher abstraction, it's generally less applicable than NumPy. Especially when used outside of its scope, operations can get complex.
- **More disk space**: Due to the internal representation of DataFrames and the way pandas trades disk space for a more performant execution, the memory usage of complex operations can spike.
- **Performance problems**: Especially when doing heavy joins, which is not recommended, memory usage can get critical and might lead to performance problems.
- **Hidden complexity**: Less experienced users often tend to overuse methods and execute them several times instead of reusing what they've already calculated. This hidden complexity makes users think that the operations themselves are simple, which is not the case.

3.

a.

Distribution Plots

Distribution plots give a deep insight into how your data is distributed. For a single variable, a histogram is effective. For multiple variables, you can either use a box plot or a violin plot. The violin plot visualizes the densities of your variables, whereas the box plot just visualizes the median, the interquartile range, and the range for each variable.

Histogram

A **histogram** visualizes the distribution of a single numerical variable. Each bar represents the frequency for a certain interval. Histograms help get an estimate of statistical measures. You see where values are concentrated, and you can easily detect outliers. You can either plot a histogram with absolute frequency values or, alternatively, normalize your histogram. If you want to compare distributions of multiple variables, you can use different colors for the bars.

Use

Get insights into the underlying distribution for a dataset.

Example

The following diagram shows the distribution of the **Intelligence Quotient (IQ)** for a test group. The dashed lines represent the standard deviation each side of the mean (the solid line):

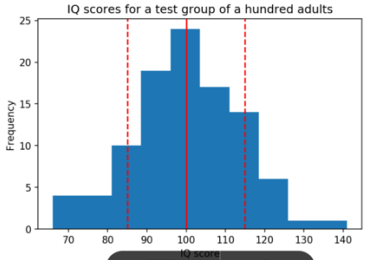


Figure 2.30: Distribution of IQ for a test group of a hundred adults

Design Practice

- Try different numbers of bins (data intervals), since the shape of the histogram can vary significantly.

Density Plot

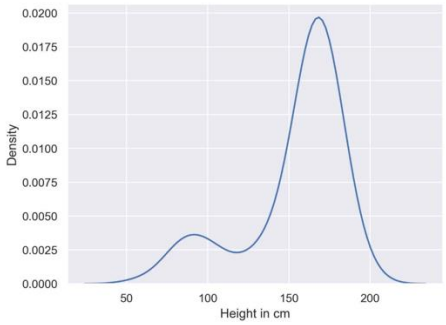
A **density plot** shows the distribution of a numerical variable. It is a variation of a histogram that uses **kernel smoothing**, allowing for smoother distributions. One advantage these have over histograms is that density plots are better at determining the distribution shape since the distribution shape for histograms heavily depends on the number of bins (data intervals).

Use

To compare the distribution of several variables by plotting the density on the same axis and using different colors.

Example

The following diagram shows a basic density plot:



Box Plot

The **box plot** shows multiple statistical measurements. The box extends from the lower to the upper quartile values of the data, thus allowing us to visualize the interquartile range (IQR). The horizontal line within the box denotes the median. The parallel extending lines from the boxes are called **whiskers**; they indicate the variability outside the lower and upper quartiles. There is also an option to show data **outliers**, usually as circles or diamonds, past the end of the whiskers.

Use

Compare statistical measures for multiple variables or groups.

Examples

The following diagram shows a basic box plot that shows the height of a group of people:

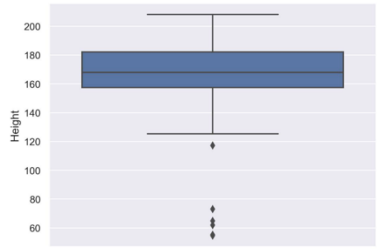


Figure 2.33: Box plot showing a single variable

Violin Plot

Violin plots are a combination of box plots and density plots. Both the statistical measures and the distribution are visualized. The thick black bar in the center represents the interquartile range, while the thin black line corresponds to the whiskers in a box plot. The white dot indicates the median. On both sides of the centerline, the density is visualized.

Use

Compare statistical measures and density for multiple variables or groups.

Examples

The following diagram shows a violin plot for a single variable and shows how students have performed in **Math**:

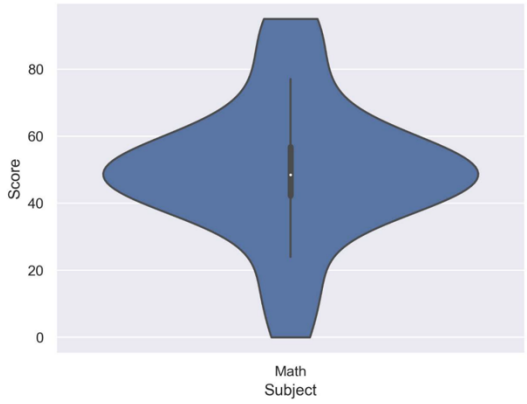


Figure 2.35: Violin plot for a single variable (Math)

From the preceding diagram, we can analyze that most of the students have scored around 40-60 in the **Math** test.

b.

Labels

Matplotlib provides a few **label** functions that we can use for setting labels to the x- and y-axes. The **`plt.xlabel()`** and **`plt.ylabel()`** functions are used to set the label for the current axes. The **`set_xlabel()`** and **`set_ylabel()`** functions are used to set the label for specified axes.

Example:

```
ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
```

You should (always) add labels to make a visualization more self-explanatory. The same is valid for titles, which will be discussed now.

Titles

A **title** describes a particular chart/graph. The titles are placed above the axes in the center, left edge, or right edge. There are two options for titles – you can either set the **Figure title** or the title of an **Axes**. The **`suptitle()`** function sets the title for the current and specified Figure. The **`title()`** function helps in setting the title for the current and specified axes.

Example:

```
fig = plt.figure()
fig.suptitle('Suptitle', fontsize=10, fontweight='bold')
```

This creates a bold Figure title with a text subtitle and a font size of 10:

```
plt.title('Title', fontsize=16)
```

The **`plt.title`** function will add a title to the Figure with text as **`Title`** and font size of **`16`** in this case.

Text

There are two options for **text** – you can either add text to a Figure or text to an Axes. The **`figtext(x, y, text)`** and **`text(x, y, text)`** functions add text at locations **`x`** or **`y`** for a Figure.

Example:

```
ax.text(4, 6, 'Text in Data Coords', bbox={'facecolor': 'yellow',
'alpha':0.5, 'pad':10})
```

Annotations

Compared to text that is placed at an arbitrary position on the Axes, **annotations** are used to annotate some features of the plot. In annotations, there are two locations to consider: the annotated location, **`xy`**, and the location of the annotation, text **`xytext`**. It is useful to specify the parameter **`arrowprops`**, which results in an arrow pointing to the annotated location.

Example:

```
ax.annotate('Example of Annotate', xy=(4,2), xytext=(8,4),
arrowprops=dict(facecolor='green', shrink=0.05))
```

This creates a green arrow pointing to the data coordinates (4, 2) with the text **Example of Annotate** at data coordinates (8, 4):

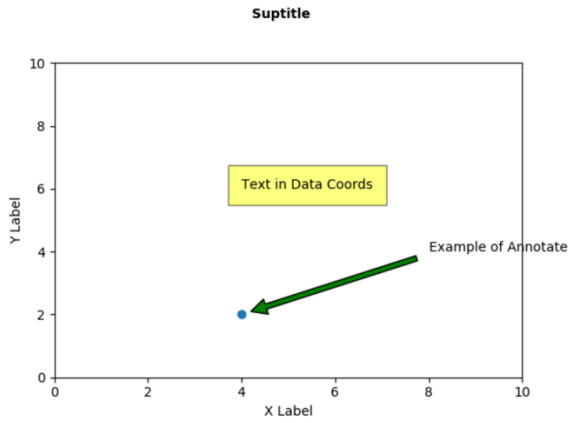


Figure 3.13: Implementation of text commands

Legends

Legend describes the content of the plot. To add a **legend** to your Axes, we have to specify the **`label`** parameter at the time of plot creation. Calling **`plt.legend()`** for the current Axes or **`Axes.legend()`** for a specific Axes will add the legend. The **`loc`** parameter specifies the location of the legend.

Example:

```
plt.plot([1, 2, 3], label='Label 1')
plt.plot([2, 4, 3], label='Label 2')
plt.legend()
```

This example is illustrated in the following diagram:

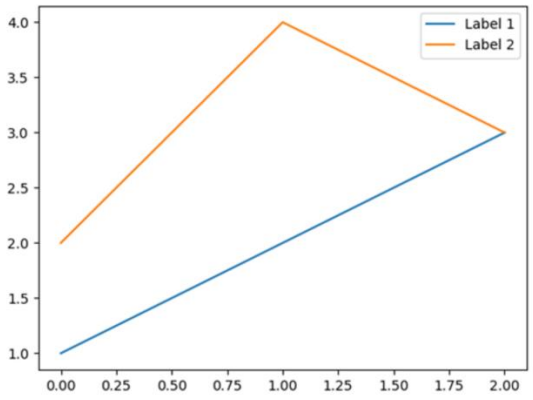


Figure 3.14: Legend example

4.

a.

Creating Figures

You can use **`plt.figure()`** to create a new **Figure**. This function returns a Figure instance, but it is also passed to the backend. Every Figure-related command that follows is applied to the current Figure and does not need to know the Figure instance.

By default, the Figure has a width of 6.4 inches and a height of 4.8 inches with a **`dpi`** (dots per inch) of 100. To change the default values of the Figure, we can use the parameters **`figsize`** and **`dpi`**.

The following code snippet shows how we can manipulate a Figure:

```
plt.figure(figsize=(10, 5)) #To change the width and the height
plt.figure(dpi=300) #To change the dpi
```

Even though it is not necessary to explicitly create a Figure, this is a good practice if you want to create multiple Figures at the same time.

Closing Figures

Figures that are no longer used should be closed by explicitly calling **`plt.close()`**, which also cleans up memory efficiently.

If nothing is specified, the **`plt.close()`** command will close the current Figure. To close a specific Figure, you can either provide a reference to a Figure instance or provide the Figure number. To find the **number** of a Figure object, we can make use of the **`number`** attribute, as follows:

```
plt.gcf().number
```

The **`plt.close('all')`** command is used to close all active Figures. The following example shows how a Figure can be created and closed:

```
plt.figure(num=10) #Create Figure with Figure number 10
plt.close(10) #Close Figure with Figure number 10
```

Displaying Figures

`plt.show()` is used to display a Figure or multiple Figures. To display Figures within a Jupyter Notebook, simply set the **`%matplotlib inline`** command at the beginning of the code.

If you forget to use **`plt.show()`**, the plot won't show up. We will learn how to save the Figure in the next section.

Saving Figures

The **`plt.savefig(fname)`** saves the current Figure. There are some useful optional parameters you can specify, such as **`dpi`**, **`format`**, or **`transparent`**. The following code snippet gives an example of how you can save a Figure:

```
plt.figure()
plt.plot([1, 2, 4, 5], [1, 3, 4, 3], '-o')
plt.savefig('lineplot.png', dpi=300, bbox_inches='tight')
#bbox_inches='tight' removes the outer white margins
```

---

b.

Pie Chart

The **`plt.pie(x, [explode], [labels], [autopct])`** function creates a pie chart.

Important parameters:

- **`x`**: Specifies the slice sizes.
- **`explode`** (optional): Specifies the fraction of the radius offset for each slice. The **`explode-array`** must have the same length as the **`x-array`**.
- **`labels`** (optional): Specifies the labels for each slice.
- **`autopct`** (optional): Shows percentages inside the slices according to the specified format string. Example: **`'%1.1f%%'`**.

Example:

```
plt.pie([0.4, 0.3, 0.2, 0.1], explode=(0.1, 0, 0, 0), labels=['A', 'B',
'C', 'D'])
```

The result of the preceding code is visualized in the following diagram:

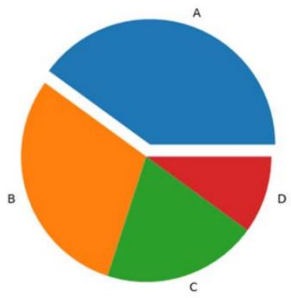


Figure 3.19: Basic pie chart

Scatter Plot

Scatter plots show data points for two numerical variables, displaying a variable on both axes. **`plt.scatter(x, y)`** creates a scatter plot of **`y`** versus **`x`**, with optionally varying marker size and/or color.

Important parameters:

- **`x`**, **`y`**: Specifies the data positions.
- **`s`**: (optional) Specifies the marker size in points squared.
- **`c`**: (optional) Specifies the marker color. If a sequence of numbers is specified, the numbers will be mapped to the colors of the color map.

Example:

```
plt.scatter(x, y)
```

The result of the preceding code is shown in the following diagram:

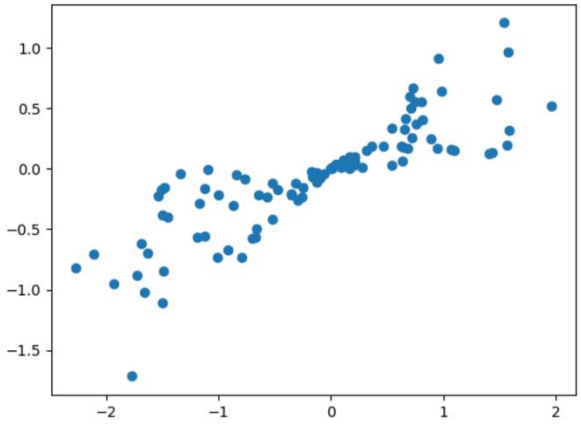


Figure 3.31: Scatter plot

5.

a.

Color Palettes

Color is a very important factor for your visualization. Color can reveal patterns in data if used effectively or hide patterns if used poorly. Seaborn makes it easy to select and use **color palettes** that are suited to your task. The **`color_palette()`** function provides an interface for many of the possible ways to generate color palettes.

The **`seaborn.color_palette([palette], [n_colors], [desat])`** command returns a list of colors, thus defining a color palette.

The parameters are as follows:

- **`palette`** (optional): Name of palette or **`None`** to return the current palette.
- **`n_colors`** (optional): Number of colors in the palette. If the specified number of colors is larger than the number of colors in the palette, the colors will be cycled.
- **`desat`** (optional): The proportion to desaturate each color by.

You can set the palette for all plots with **`set_palette()`**. This function accepts the same arguments as **`color_palette()`**. In the following sections, we will explain how color palettes are divided into different groups.

Categorical Color Palettes

Categorical palettes (or qualitative color palettes) are best suited for distinguishing categorical data that does not have an inherent ordering. The color palette should have colors as distinct from one another as possible, resulting in palettes where mainly the hue changes. When it comes to human perception, there is a limit to how many different colors are perceived. A rule of thumb is that if you have double-digit categories, it is advisable to divide the categories into groups. Different shades of color could be used for a group. Another way to keep groups apart could be to use hues that are close together in the color wheel within a group and hues that are far apart for different groups.

Some examples where it is suitable to use categorical color palettes are line charts showing stock trends for different companies, and a bar chart with subcategories; basically, any time you want to group your data.

There are six default themes in Seaborn: **`deep`**, **`muted`**, **`bright`**, **`pastel`**, **`dark`**, and **`colorblind`**. The code and output for each theme are provided in the following diagram. Out of these color palettes, it doesn't really matter which one you use. Choose the one you prefer and the one that best fits the overall theme of the visualization. It's never a bad idea to use the colorblind palette to account for colorblind people. The following is the code to create a deep color palette:

```
import seaborn as sns
palette1 = sns.color_palette("deep")
sns.palplot(palette1)
```

Sequential Color Palettes

Sequential color palettes are appropriate for sequential data ranges from low to high values, or vice versa. It is recommended to use bright colors for low values and dark ones for high values. Some examples of sequential data are absolute temperature, weight, height, or the number of students in a class.

One of the sequential color palettes that Seaborn offers is cubehelix palettes. They have a linear increase or decrease in brightness and some variation in hue, meaning that even when converted to black and white, the information is preserved.

The default palette returned by **`cubehelix_palette()`** is illustrated in the following diagram. To customize the cubehelix palette, the hue at the start of the helix can be set with **`start`** (a value between 0 and 3), or the number of rotations around the hue wheel can be set with **`rot`**:

Figure 4.20: Cubehelix palette

Creating custom sequential palettes that only produce colors that start at either light or dark desaturated colors and end with a specified color can be accomplished with **`light_palette()`** or **`dark_palette()`**. Two examples are given in the following:

```
custom_palette2 = sns.light_palette("magenta")
sns.palplot(custom_palette2)
```

Diverging Color Palettes

Diverging color palettes are used for data that consists of a well-defined midpoint. An emphasis is placed on both high and low values. For example, if you are plotting any population changes for a particular region from some baseline population, it is best to use diverging colormaps to show the relative increase and decrease in the population. The following code snippet and output provides a better understanding of diverging plots, wherein we use the **`coolwarm`** template, which is built into Matplotlib:

```
custom_palette4 = sns.color_palette("coolwarm", 7)
sns.palplot(custom_palette4)
```

---

b.

Seaborn Figure Styles

To control the plot style, Seaborn provides two methods: **`set_style(style, [rc])`** and **`axes_style(style, [rc])`**.

`seaborn.set_style(style, [rc])` sets the aesthetic style of the plots.

Parameters:

- **`style`**: A dictionary of parameters or the name of one of the following preconfigured sets: **`darkgrid`**, **`whitegrid`**, **`dark`**, **`white`**, or **`ticks`**
- **`rc`** (optional): Parameter mappings to override the values in the preset Seaborn-style dictionaries

Here is an example:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("whitegrid")
plt.figure()
x1 = [10, 20, 5, 40, 8]
x2 = [30, 43, 9, 7, 20]
plt.plot(x1, label='Group A')
plt.plot(x2, label='Group B')
plt.legend()
plt.show()
```

This results in the following plot:

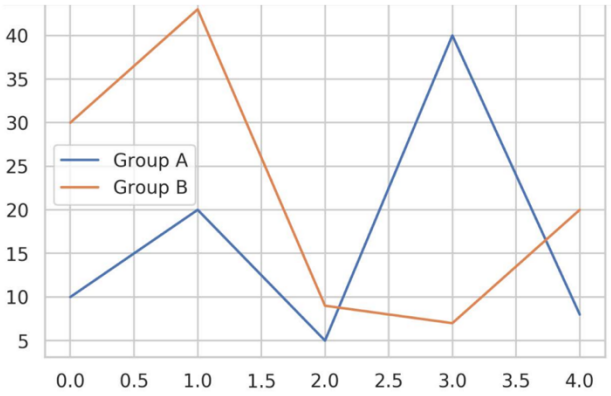


Figure 4.4: Seaborn line plot with whitegrid style

`seaborn.axes_style(style, [rc])` returns a parameter dictionary for the aesthetic style of the plots. The function can be used in a **`with`** statement to temporarily change the style parameters.

Here are the parameters:

- **`style`**: A dictionary of parameters or the name of one of the following pre-configured sets: **`darkgrid`**, **`whitegrid`**, **`dark`**, **`white`**, or **`ticks`**
- **`rc`** (optional): Parameter mappings to override the values in the preset Seaborn-style dictionaries

Here is an example:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
plt.figure()
x1 = [10, 20, 5, 40, 8]
x2 = [30, 43, 9, 7, 20]
with sns.axes_style('dark'):
    plt.plot(x1, label='Group A')
    plt.plot(x2, label='Group B')
plt.legend()
plt.show()
```

The aesthetics are only changed temporarily. The result is shown in the following diagram:

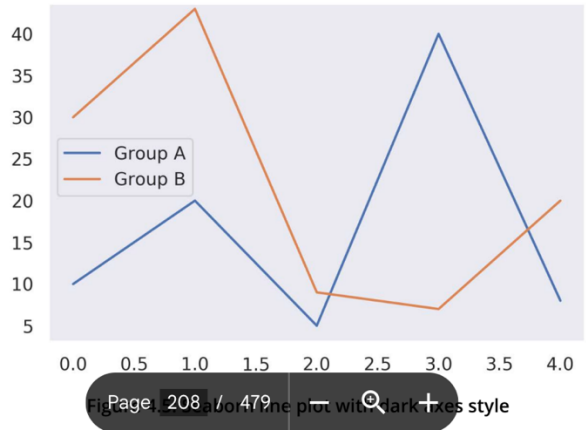


Figure 4.1: Seaborn line plot with dark axes style

Removing Axes Spines

Sometimes, it might be desirable to remove the top and right axes spines. The **`despine()`** function is used to remove the top and right axes spines from the plot:

```
seaborn.despine(fig=None, ax=None, top=True, right=True, left=False,
bottom=False, offset=None, trim=False)
```

The following code helps to remove the axes spines:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("white")
plt.figure()
x1 = [10, 20, 5, 40, 8]
x2 = [30, 43, 9, 7, 20]
plt.plot(x1, label='Group A')
plt.plot(x2, label='Group B')
sns.despine()
plt.legend()
plt.show()
```

This results in the following plot:

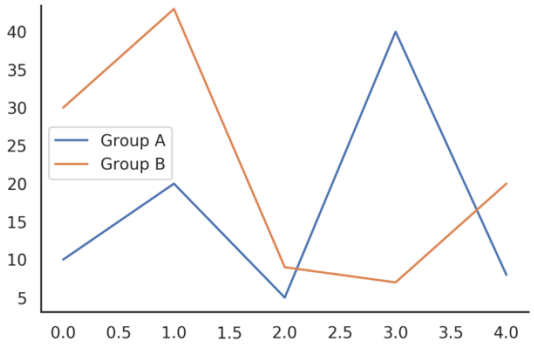


Figure 4.6: Despined Seaborn line plot

6.

a.

Kernel Density Estimation

It is often useful to visualize how variables of a dataset are distributed. Seaborn offers handy functions to examine univariate and bivariate distributions. One possible way to look at a univariate distribution in Seaborn is by using the **`distplot()`** function. This will draw a histogram and fit a **kernel density estimate** (**KDE**), as illustrated in the following example:

```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
data = pd.read_csv('../../Datasets/age_salary_hours.csv')
sns.distplot(data.loc[:, 'Age'])
plt.xlabel('Age')
plt.ylabel('Density')
```

The result is shown in the following diagram:

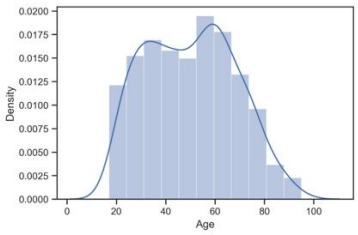


Figure 4.30: KDE with a histogram for a univariate distribution

To just visualize the KDE, Seaborn provides the **`kdeplot()`** function:

```
sns.kdeplot(data.loc[:, 'Age'], shade=True)
plt.xlabel('Age')
plt.ylabel('Density')
```

The KDE plot is shown in the following diagram, along with a shaded area under the curve:

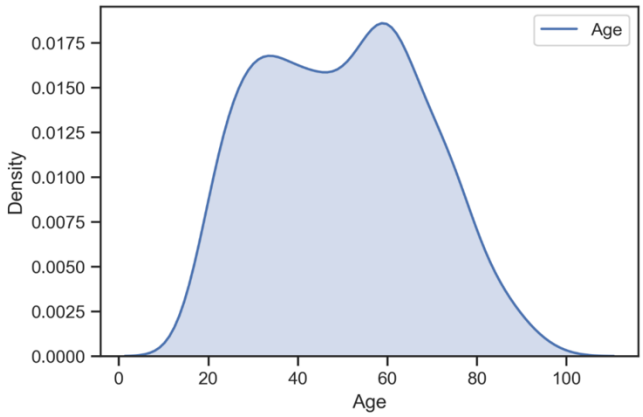


Figure 4.31: KDE for a univariate distribution

Plotting Bivariate Distributions

For visualizing **bivariate distributions**, we will introduce three different plots. The first two plots use the **`jointplot()`** function, which creates a multi-panel figure that shows both the joint relationship between both variables and the corresponding marginal distributions.

A scatter plot shows each observation as points on the **`x`** and **`y`** axes. Additionally, a histogram for each variable is shown:

```
import pandas as pd
import seaborn as sns
data = pd.read_csv('../../Datasets/age_salary_hours.csv')
sns.set(style="white")
sns.jointplot(x="Annual Salary", y="Age", data=data))
```

The scatter plot with marginal histograms is shown in the following diagram:

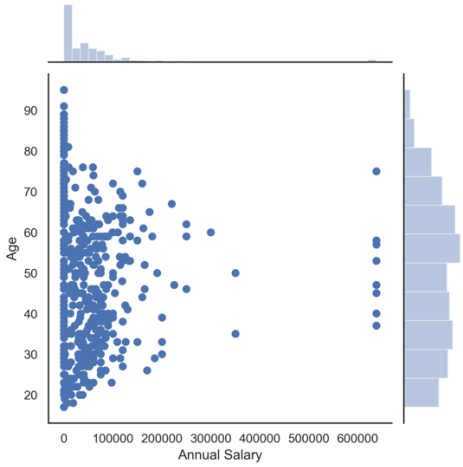


Figure 4.32: Scatter plot with marginal histograms

It is also possible to use the KDE procedure to visualize bivariate distributions. The joint distribution is shown as a contour plot, as demonstrated in the following code:

```
sns.jointplot('Annual Salary', 'Age', data=subdata, kind='kde', xlim=(0,
500000), ylim=(0, 100))
```

b.

Removing Axes Spines

Sometimes, it might be desirable to remove the top and right axes spines. The **`despine()`** function is used to remove the top and right axes spines from the plot:

```
seaborn.despine(fig=None, ax=None, top=True, right=True, left=False,
bottom=False, offset=None, trim=False)
```

The following code helps to remove the axes spines:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("white")
plt.figure()
x1 = [10, 20, 5, 40, 8]
x2 = [30, 43, 9, 7, 20]
plt.plot(x1, label='Group A')
plt.plot(x2, label='Group B')
sns.despine()
plt.legend()
plt.show()
```

This results in the following plot:

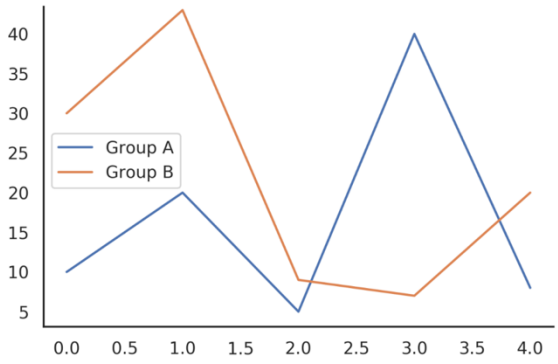


Figure 4.6: Despined Seaborn line plot

c.

The following example demonstrates the usage of violin plots:

```
import pandas as pd
import seaborn as sns
data = pd.read_csv("../../Datasets/salary.csv")
sns.set(style="whitegrid")
sns.violinplot('Education', 'Salary', hue='Gender', data=data, split=True,
cut=0)
```

The result appears as follows:

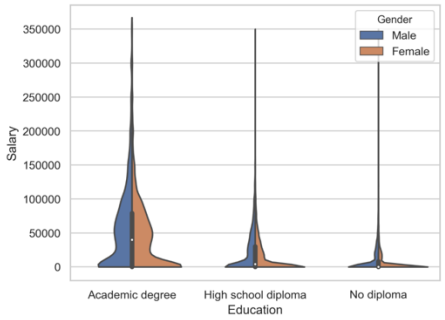


Figure 4.35: Seaborn violin plot

The violin plot shows both statistical measures and the probability distribution. The data is divided into education groups, which are shown on the x-axis, and gender groups, which are highlighted by different colors.

With the next activity, we will conclude the section about advanced plots. In this section, multi-plots in Seaborn are introduced.

7.

a.

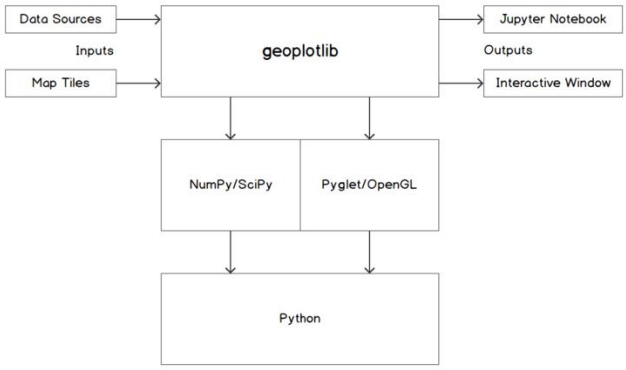


Figure 5.1: Conceptual architecture of geoplotlib

geoplotlib uses the concept of **layers** that can be placed on top of one another, providing a powerful interface for even complex visualizations. It comes with several common visualization layers that are easy to set up and use.

From the preceding diagram, we can see that **geoplotlib** is built on top of **NumPy/SciPy** and **Pyglet/OpenGL**. These libraries take care of numerical operations and rendering. Both components are based on Python, therefore enabling the use of the full Python ecosystem.

The Design Principles of geoplotlib

Taking a closer look at the internal design of geoplotlib, we can see that it is built around three design principles:

- **Integration**: geoplotlib visualizations are purely Python-based. This means that generic Python code can be executed, and other libraries such as pandas can be used for **data wrangling** purposes. We can manipulate and enrich our datasets using pandas **DataFrames** and later convert them into a geoplotlib `DataAccessObject`, which we need for optimal compatibilities, as follows:

```
import pandas as pd
from geoplotlib.utils import DataAccessObject

# data wrangling with pandas DataFrames here
dataset_obj = DataAccessObject(dataset_filtered)
```

  geoplotlib fully integrates into the Python ecosystem. This even enables us to plot geographical data inline inside our Jupyter Notebooks. This possibility allows us to design our visualizations quickly and iteratively.

- **Simplicity**: Looking at the example provided here, we can quickly see that geoplotlib abstracts away the complexity of plotting map tiles and already-provided layers such as **dot density** and **histogram**. It has a simple API that provides common visualizations. These visualizations can be created using custom data with only a few lines of code.

- **Performance**: As we mentioned before, geoplotlib can handle large amounts of data due to the use of NumPy for accelerated numerical operations and **OpenGL** for accelerated graphical rendering.

---

b.

Adding Widgets

One of the most powerful features of Bokeh is the ability to use **widgets** to interactively change the data that's displayed in a visualization. To understand the importance of interactivity in your visualizations, imagine seeing a static visualization about stock prices that only shows data for the last year.

```
# importing the widgets
from ipywidgets import interact, interact_manual

# creating an input text
@interact(Value='Input Text')
def text_input(Value):
print(Value)
```

The following screenshot shows the output of the preceding code:

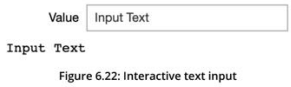


Figure 6.22: Interactive text input

8.

a.

Bokeh is an interactive visualization library focused on modern browsers and the web. Other than Matplotlib or geoplotlib, the plots and visualizations we are going to create in this chapter will be based on JavaScript widgets. Bokeh allows us to create visually appealing plots and graphs nearly out of the box without much styling. In addition to that, it helps us construct performant interactive dashboards based on large static datasets or even streaming data.

Bokeh has been around since 2013, with *version* 1.4.0 being released in November 2019. It targets modern web browsers to present interactive visualizations to users rather than static images. The following are some of the features of Bokeh:

- **Simple visualizations**: Through its different interfaces, it targets users of many skill levels, providing an API for quick and straightforward visualizations as well as more complex and extremely customizable ones.
- **Excellent animated visualizations**: It provides high performance and can, therefore, work on large or even streaming datasets, which makes it the go-to choice for animated visualizations and data analysis.
- **Inter-visualization interactivity**: This is a web-based approach; it's easy to combine several plots and create unique and impactful dashboards with visualizations that can be interconnected to create inter-visualization interactivity.
- **Supports multiple languages**: Other than Matplotlib and geoplotlib, Bokeh has libraries for both Python and JavaScript, in addition to several other popular languages.
- **Multiple ways to perform a task**: Adding interactivity to Bokeh visualizations can be done in several ways. The simplest built-in way is the ability to zoom and pan in and out of your visualization. This gives the users better control of what they want to see. It also allows users to filter and transform the data.
- **Beautiful chart styling**: The tech stack is based on Tornado in the backend and is powered by D3 in the frontend. D3 is a JavaScript library for creating outstanding visualizations. Using the underlying D3 visuals allows us to create beautiful plots without much custom styling.

---

b.

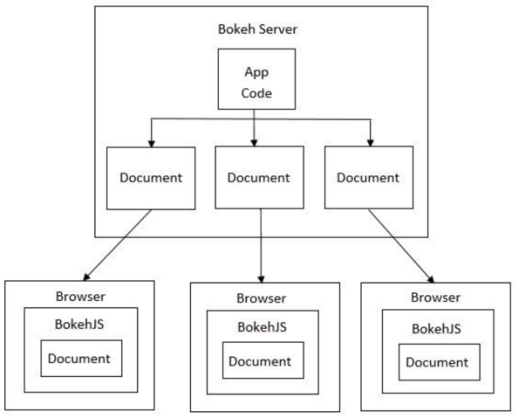
Bokeh Server

Bokeh creates **scene graph** JSON objects that will be interpreted by the BokehJS library to create the visualization output. This process gives you a unified format for other languages to create the same Bokeh plots and visualizations, independently of the language used.

To create more complex visualizations and leverage the tooling provided by Python, we need a way to keep our visualizations in sync with one another. This way, we can not only filter data but also do calculations and operations on the server-side, which updates the visualizations in real-time.

In addition to that, since we will have an entry point for data, we can create visualizations that get fed by streams instead of static datasets. This design provides a way to develop more complex systems with even greater capabilities.

Looking at the scheme of this architecture, we can see that the documents are provided on the server-side, then moved over to the browser, which then inserts it into the BokehJS library. This insertion will trigger the interpretation by BokehJS, which will then create the visualization. The following diagram describes how the Bokeh server works:



C.

Voronoi Tessellation

In a **Voronoi tessellation**, each pair of data points is separated by a line that is the same distance from both data points. The separation creates cells that, for every given point, marks which data point is closer. The closer the data points, the smaller the cells.

The following example shows how you can simply use the **`voronoi`** method to create this visualization:

```
# plotting our dataset as voronoi plot
geoplotlib.voronoi(dataset_filtered, line_color='b')
geoplotlib.set_smoothing(True)

geoplotlib.show()
```

As we can see, the code to create this visualization is relatively short.

After importing the dependencies we need, we read the dataset using the **`read_csv`** method of pandas (or geoplotlib). We then use it as data for our **`voronoi`** method, which handles all of the complex logic of plotting the data on the map.

In addition to the data itself, we can set several parameters, such as general smoothing using the **`set_smoothing`** method. The smoothing of the lines uses anti-aliasing:

Delaunay Triangulation

A **Delaunay triangulation** is related to Voronoi tessellation. When connecting each data point to every other data point that shares an edge, we end up with a plot that is triangulated. The closer the data points are to each other, the smaller the triangles will be. This gives us a visual clue about the density of points in specific areas. When combined with color gradients, we get insights about points of interest, which can be compared with a heatmap:

```
# plotting our dataset as a delaunay
geoplotlib.delaunay(dataset_filtered, cmap='hot_r')
geoplotlib.set_smoothing(True)

geoplotlib.show()
```

9.

a.

What is a Socket?

A **socket** is an endpoint for sending or receiving data across a **network**. It acts as an **interface** between an application and the underlying **network protocol (TCP/IP, UDP, etc.)**.

- In **TCP/IP networking**, a socket is identified by:
  - **IP Address** (e.g., `192.168.1.1`)
  - **Port Number** (e.g., `8080`)
  - **Protocol** (e.g., TCP or UDP)

How Socket Connection is Established Over TCP/IP

TCP (Transmission Control Protocol) is a **connection-oriented** protocol, meaning it requires a **handshake** before data transfer. A socket connection over TCP/IP is established in **three main steps**:

1. Socket Creation

Both the client and server create a socket using a programming language (e.g., Python, Java, C++).
The server binds its socket to an **IP address and port**.

2. TCP Three-Way Handshake

Before communication begins, TCP performs a **three-way handshake** to establish a connection:

Step 1: SYN (Synchronize)

- The **client** sends a SYN request to the server to initiate a connection.

Step 2: SYN-ACK (Acknowledge)

- The **server** responds with a SYN-ACK to confirm the connection request.

Step 3: ACK (Acknowledge)

- The **client** sends an ACK to finalize the connection.

Now, the socket connection is **established**, and data transmission can begin.

3. Data Transmission

Once the connection is established, the client and server can exchange data **bidirectionally** using **send()** and **recv()** functions.

---

4. Closing the Connection

When communication is complete, the connection is **closed** using the **FIN (Finish) and ACK (Acknowledge) process**.

b.

13.4 JavaScript Object Notation - JSON

The JSON format was inspired by the object and array format used in the JavaScript language. But since Python was invented before JavaScript, Python's syntax for dictionaries and lists influenced the syntax of JSON. So the format of JSON is nearly identical to a combination of Python lists and dictionaries.

Here is a JSON encoding that is roughly equivalent to the simple XML from above:

```
{
  "name" : "Chuck",
  "phone" : {
    "type" : "intl",
    "number" : "+1 734 303 4456"
```



```
  },
  "email" : {
    "hide" : "yes"
  }
}
```

You will notice some differences. First, in XML, we can add attributes like "intl" to the "phone" tag. In JSON, we simply have key-value pairs. Also the XML "person" tag is gone, replaced by a set of outer curly braces.

In general, JSON structures are simpler than XML because JSON has fewer capabilities than XML. But JSON has the advantage that it maps *directly* to some combination of dictionaries and lists. And since nearly all programming languages have something equivalent to Python's dictionaries and lists, JSON is a very natural format to have two cooperating programs exchange data.

JSON is quickly becoming the format of choice for nearly all data exchange between applications because of its relative simplicity compared to XML.

13.5 Parsing JSON

13.5 Parsing JSON

We construct our JSON by nesting dictionaries and lists as needed. In this example, we represent a list of users where each user is a set of key-value pairs (i.e., a dictionary). So we have a list of dictionaries.

In the following program, we use the built-in `json` library to parse the JSON and read through the data. Compare this closely to the equivalent XML data and code above. The JSON has less detail, so we must know in advance that we are getting a list and that the list is of users and each user is a set of key-value pairs. The JSON is more succinct (an advantage) but also is less self-describing (a disadvantage).

```
import json

data = '''
[
  { "id" : "001",
    "x" : "2",
    "name" : "Chuck"
  } ,
  { "id" : "009",
    "x" : "7",
    "name" : "Brent"
  }
]'''

info = json.loads(data)
print('User count:', len(info))

for item in info:
    print('Name', item['name'])
```



```
    print('Id', item['id'])
    print('Attribute', item['x'])

# Code: https://www.py4e.com/code3/json2.py
```

If you compare the code to extract data from the parsed JSON and XML you will see that what we get from `json.loads()` is a Python list which we traverse with a `for` loop, and each item within that list is a Python dictionary. Once the JSON has been parsed, we can use the Python index operator to extract the various bits of data for each user. We don't have to use the JSON library to dig through the parsed JSON, since the returned data is simply native Python structures.

The output of this program is exactly the same as the XML version above.

```
User count: 2
Name Chuck
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

10.

a.

13.1 eXtensible Markup Language - XML

XML looks very similar to HTML, but XML is more structured than HTML. Here is a sample of an XML document:

```
<person>
  <name>Chuck</name>
  <phone type="intl">
    +1 734 303 4456
  </phone>
  <email hide="yes" />
</person>
```

Each pair of opening (e.g., `<person>`) and closing tags (e.g., `</person>`) represents a *element* or *node* with the same name as the tag (e.g., `person`). Each element can have some text, some attributes (e.g., `hide`), and other nested elements. If an XML element is empty (i.e., has no content), then it may be depicted by a self-closing tag (e.g., `<email />`).

Often it is helpful to think of an XML document as a tree structure where there is a top element (here: `person`), and other tags (e.g., `phone`) are drawn as *children* of their *parent* elements.

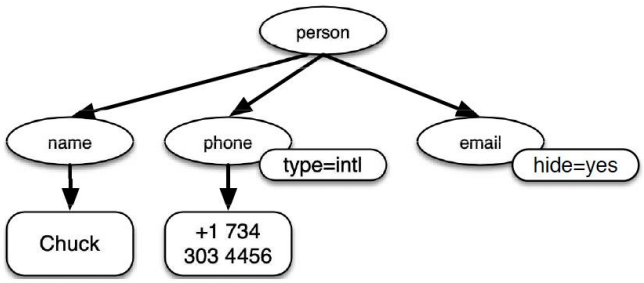


Figure 13.1: A Tree Representation of XML

13.2 Parsing XML

Here is a simple application that parses some XML and extracts some data elements from the XML:

```
import xml.etree.ElementTree as ET

data = '''
<person>
  <name>Chuck</name>
  <phone type="intl">
    +1 734 303 4456
  </phone>
  <email hide="yes" />
</person>'''

tree = ET.fromstring(data)
print('Name:', tree.find('name').text)
print('Attr:', tree.find('email').get('hide'))

# Code: https://www.py4e.com/code3/xml1.py
```

The triple single quote (`'''`), as well as the triple double quote (`"""`), allow for the creation of strings that span multiple lines.

Calling `fromstring` converts the string representation of the XML into a "tree" of XML elements. When the XML is in a tree, we have a series of methods we can call to extract portions of data from the XML string. The `find` function searches through the XML tree and retrieves the element that matches the specified tag.

```
Name: Chuck
Attr: yes
```

Using an XML parser such as `ElementTree` has the advantage that while the XML in this example is quite simple, it turns out there are many rules regarding

valid XML, and using `ElementTree` allows us to extract data from XML without worrying about the rules of XML syntax.

b.

```python
import requests
from bs4 import BeautifulSoup

# Fetch the webpage
url = "https://example.com"  # Replace with any webpage URL
response = requests.get(url)

# Parse HTML content
soup = BeautifulSoup(response.text, "html.parser")

# Extract the page title
print("Page Title:", soup.title.string)

# Extract all paragraph texts
print("\nParagraphs:")
for para in soup.find_all("p"):
    print(para.text)

# Extract all links
print("\nLinks:")
for link in soup.find_all("a"):
    print(link.get("href"))
```

3. Explanation

- `requests.get(url)` : Fetches HTML content from the webpage.
- `BeautifulSoup(html, "html.parser")` : Parses the HTML.
- `soup.title.string` : Extracts the page title.
- `soup.find_all("p")` : Finds all paragraph (`<p>`) tags.
- `soup.find_all("a")` : Finds all anchor (`<a>`) tags and extracts links.

---

4. Output Example

sh Copy Edit

```
Page Title: Example Domain

Paragraphs:
This domain is for use in illustrative examples in documents.

Links:
https://www.iana.org/domains/example
```