Python and Statistics for Climate Informatics

Beginner's Guide for Python Data Analysis - 3

Dr. Jangho Lee

University of Illinois Chicago
Department of Earth and Environmental Sciences

November 2024

1 Matplotlib — Visualization with Python

1.1 Introduction to Matplotlib

Matplotlib is one of the most popular data visualization libraries in Python. It provides a comprehensive range of tools for creating static, animated, and interactive plots. Matplotlib is particularly useful when working with data in scientific computing, engineering, and data analysis, as it allows for clear and customizable plots that help to better understand and interpret data.

This chapter provides an introduction to Matplotlib, covering the following:

- Overview of Matplotlib and installation.
- The basic structure of a plot in Matplotlib.
- Creating simple plots (line plot, scatter plot, bar chart).
- Saving plots as image files.

1.1.1 Overview of Matplotlib

Matplotlib is a powerful library for creating visualizations in Python. It is often used in conjunction with other libraries such as NumPy and Pandas to create visual representations of datasets. The primary interface for Matplotlib is `pyplot`, which provides functions for creating and customizing plots in a simple and intuitive way.

To get started with Matplotlib, you need to install it. You can install Matplotlib using `pip`:

```
pip install matplotlib
```

Once installed, you can import Matplotlib and use it to create plots.

Code 1.1

```python
import matplotlib.pyplot as plt

# Create a simple plot
x = [0, 1, 2, 3, 4]
y = [0, 1, 4, 9, 16]

plt.plot(x, y)  # Create a line plot
plt.title("Simple Line Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()
```

Explanation: - In this example, we imported Matplotlib's `pyplot` module as `plt`. - We created a simple line plot with the data points `x` and `y`. - The `plt.plot(x, y)` function plots the data, and `plt.title()`, `plt.xlabel()`, and `plt.ylabel()` add the title and labels to the axes.

1.2 Basic Plot Types

Matplotlib supports a variety of plot types. Here are a few basic types that are commonly used:

- **Line Plot**: Displays data as a series of points connected by straight lines.
- **Scatter Plot**: Displays data as individual points, helpful for showing relationships between variables.
- **Bar Chart**: Used for comparing discrete values across categories.

We will now explore these basic plot types.

1.2.1 Line Plot

A line plot is the most basic type of plot, useful for displaying trends over time or continuous data.

Code 1.2

```
# Example: Line plot of quadratic data
x = [0, 1, 2, 3, 4]
y = [0, 1, 4, 9, 16]

plt.plot(x, y)
plt.title("Quadratic Function")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.grid(True)
plt.show()
```

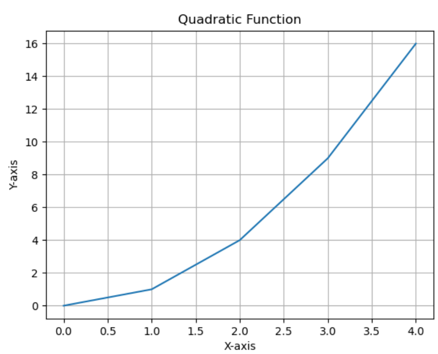

Figure 1: Line plot of a quadratic function

Explanation: - In this example, we plotted a quadratic function ($y = x^2$). - The `plt.grid(True)` adds a grid to the plot, which helps in visually comparing values. - The plot is displayed using `plt.show()`.

1.2.2 Scatter Plot

Scatter plots are useful for visualizing the relationship between two variables, where each point represents a pair of values.

Code 1.3

```
# Example: Scatter plot
x = [1, 2, 3, 4, 5]
y = [5, 4, 3, 2, 1]

plt.scatter(x, y)
plt.title("Scatter Plot")
plt.xlabel("X-axis")
```

```
plt.ylabel("Y-axis")
plt.show()
```

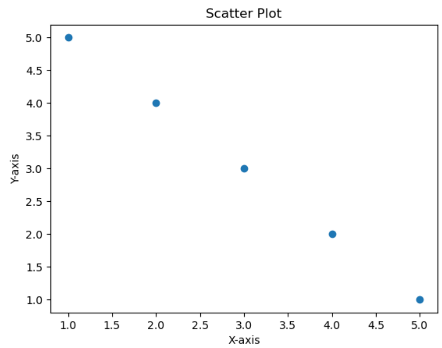

Figure 2: Scatter plot showing inverse relationship

Explanation: - The scatter plot shows an inverse relationship between `x` and `y`, as the points follow a downward trend. - Scatter plots are particularly useful for spotting correlations between variables.

1.2.3 Bar Chart

Bar charts are used for comparing quantities across different categories. You can use bar charts for both vertical and horizontal comparisons.

Code 1.4

```
# Example: Bar chart
categories = ['A', 'B', 'C', 'D', 'E']
values = [3, 7, 2, 5, 8]

plt.bar(categories, values)
plt.title("Bar Chart Example")
plt.xlabel("Category")
plt.ylabel("Value")
plt.show()
```

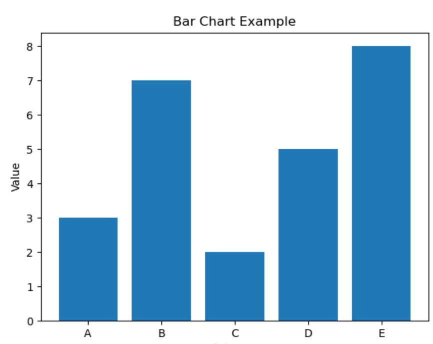

Figure 3: Bar chart comparing values across categories

Explanation: - The `plt.bar()` function is used to create vertical bars for each category. - This is a useful chart for comparing discrete data across different categories.

1.2.4 Saving Plots as Image Files

Once you create a plot, you can save it as an image file using the `savefig()` function. This is useful when you want to export your plots for reports or presentations.

Code 1.5

```
# Saving the plot as an image file
plt.plot(x, y)
plt.title("Saved Line Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.savefig("figures/004.png")  # Save the plot as a PNG image
```

Explanation: - The `savefig()` function saves the plot as an image file (in this case, a PNG). - You can specify the file format (e.g., PNG, PDF, SVG) by changing the file extension.

These fundamental tools will allow you to create simple visualizations, which can be customized and extended as needed for more complex datasets. In the next chapter, we will explore more advanced plotting techniques, including customizing plot appearance, adding annotations, and working with multiple plots.

1.3 Plot Customization

Customizing the appearance of plots is crucial for creating clear, informative, and aesthetically pleasing visualizations. Matplotlib provides extensive customization options, allowing you to modify nearly every aspect of your plot. In this chapter, we will explore how to:

- Customize plot titles, axis labels, and legends.
- Modify colors, line styles, and markers.
- Adjust axis limits and ticks.
- Add annotations to plots.

These customizations will help you create professional-looking visualizations that convey your data more effectively.

1.3.1 Customizing Titles and Labels

Titles and axis labels are essential for providing context to your plot. Matplotlib allows you to customize the title and labels with different fonts, colors, and positions.

Code 1.6

```
import matplotlib.pyplot as plt

# Data for plotting
x = [0, 1, 2, 3, 4]
y = [0, 1, 4, 9, 16]

# Create a basic line plot
plt.plot(x, y)

# Customize title and labels
plt.title("Customized Line Plot", fontsize=16, color='blue', loc='center')  # Title
    customization
```

```
plt.xlabel("X-axis", fontsize=12, color='green')  # X-axis label customization
plt.ylabel("Y-axis", fontsize=12, color='red')  # Y-axis label customization
plt.show()
```

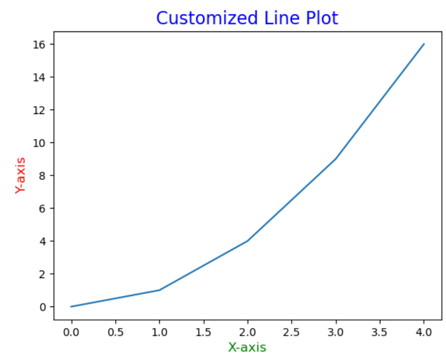

Figure 4: Line plot with customized title and labels

Explanation: - The `plt.title()` function allows you to set the title of the plot. You can customize the font size, color, and alignment using the `fontsize`, `color`, and `loc` parameters. - The `plt.xlabel()` and `plt.ylabel()` functions are used to set the x-axis and y-axis labels, with similar customization options.

1.3.2 Customizing Line Styles, Colors, and Markers

Matplotlib allows you to customize the appearance of lines, including their color, style, and markers. You can specify these properties directly in the plot function or using additional arguments.

Code 1.7

```
# Create a line plot with customized line style, color, and markers
plt.plot(x, y, linestyle='--', color='purple', marker='o', markersize=8)

# Customize title and labels
plt.title("Line Plot with Custom Styles", fontsize=16)
plt.xlabel("X-axis", fontsize=12)
plt.ylabel("Y-axis", fontsize=12)
plt.show()
```

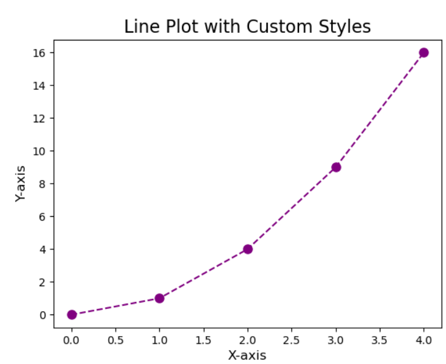

Figure 5: Line plot with customized line style, color, and markers

Explanation: - The `linestyle` argument controls the line style. In this example, `'--'` creates a dashed line. - The `color` argument specifies the color of the line. You can use color names or hexadecimal color codes. - The `marker` argument specifies the shape of the markers (e.g., 'o' for circles), and `markersize` controls their size.

1.3.3 Customizing Axis Limits and Ticks

Sometimes, you may want to adjust the axis limits or modify the tick marks for better visualization of your data. You can use the `plt.xlim()`, `plt.ylim()`, and `plt.xticks()` functions to customize the axes and ticks.

Code 1.8

```
# Create a simple plot
plt.plot(x, y)

# Customize axis limits and ticks
plt.xlim(-1, 5)  # Set x-axis limits
plt.ylim(-1, 20)  # Set y-axis limits
plt.xticks([0, 1, 2, 3, 4, 5])  # Set x-axis ticks
plt.yticks([0, 5, 10, 15])  # Set y-axis ticks

plt.title("Plot with Custom Axis Limits and Ticks", fontsize=16)
plt.xlabel("X-axis", fontsize=12)
plt.ylabel("Y-axis", fontsize=12)
plt.show()
```

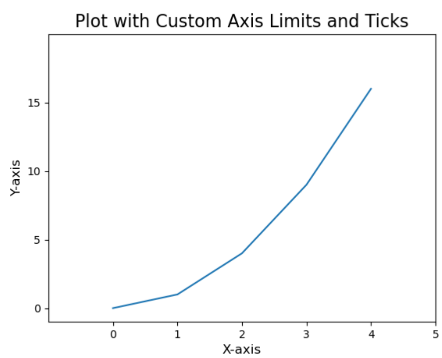

Figure 6: Line plot with custom axis limits and ticks

Explanation: - The `plt.xlim()` and `plt.ylim()` functions are used to set the limits for the x and y axes. - The `plt.xticks()` and `plt.yticks()` functions allow you to set custom tick positions on the x and y axes.

1.3.4 Adding Legends

Legends are helpful for distinguishing between different data series in a plot. You can add a legend using the `plt.legend()` function.

Code 1.9

```
# Create two line plots
plt.plot(x, y, label='y = x^2', color='red')
plt.plot(x, [i**1.5 for i in x], label='y = x^1.5', color='blue')

# Add a legend
plt.legend(title="Functions")

```

```
# Customize title and labels
plt.title("Line Plot with Legend", fontsize=16)
plt.xlabel("X-axis", fontsize=12)
plt.ylabel("Y-axis", fontsize=12)
plt.show()
```

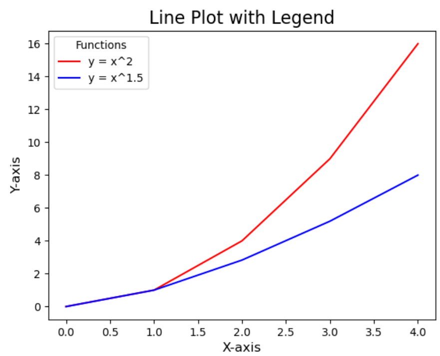

Figure 7: Line plot with multiple series and a legend

Explanation: - The `label` parameter in the `plt.plot()` function is used to specify the legend label for each line. - The `plt.legend()` function adds the legend to the plot. You can also use the `title` parameter to give the legend a title.

1.4 Multiple Subplots

When working with data visualizations, it's often helpful to display multiple plots in a single figure. Matplotlib provides several ways to create multiple plots, including the `subplots()` function for arranging plots in a grid and the `gridspec` layout for more customized arrangements. In this chapter, we will explore both methods for creating multiple subplots and customizing their appearance.

- Creating multiple subplots using `subplots()`.
- Customizing subplots with `gridspec`.
- Adjusting subplot spacing and layout.

By the end of this chapter, you will have a solid understanding of how to manage multiple subplots efficiently and customize their layout to suit your needs.

1.4.1 Creating Multiple Subplots with `subplots()`

The `subplots()` function is the simplest way to create multiple subplots in a grid. You can specify the number of rows and columns, and it will return an array of axes objects for each subplot. Each plot can then be customized independently.

Code 1.10

```
import matplotlib.pyplot as plt

# Create multiple subplots (2 rows, 2 columns)
fig, axes = plt.subplots(2, 2, figsize=(10, 8))

```

```python
# Plotting on each subplot
axes[0, 0].plot([0, 1, 2, 3], [0, 1, 4, 9], color='red')
axes[0, 0].set_title('Plot 1: Line')

axes[0, 1].bar([1, 2, 3, 4], [5, 7, 3, 4], color='blue')
axes[0, 1].set_title('Plot 2: Bar')

axes[1, 0].scatter([1, 2, 3, 4], [5, 7, 3, 4], color='green')
axes[1, 0].set_title('Plot 3: Scatter')

axes[1, 1].hist([1, 2, 2, 3, 3, 3, 4], bins=4, color='purple')
axes[1, 1].set_title('Plot 4: Histogram')

# Adjust spacing between subplots
plt.tight_layout()

plt.show()
```

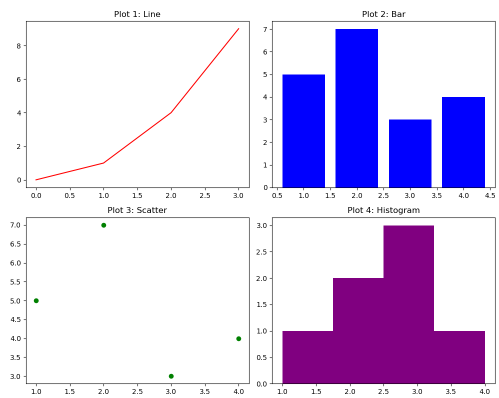

Figure 8: Multiple subplots using `subplots()`

Explanation: - In this example, we created a 2x2 grid of subplots using `plt.subplots(2, 2)`. The `figsize` parameter controls the overall figure size. - Each subplot is accessed using the `axes` array. For instance, `axes[0, 0]` corresponds to the first subplot (top-left). - We used different plot types (line, bar, scatter, and histogram) in each subplot. - The `plt.tight_layout()` function adjusts the spacing between subplots for better readability.

1.4.2 Customizing Subplots with `gridspec`

While `subplots()` is a simple way to create multiple subplots, `gridspec` provides more control over subplot layouts. It allows for complex arrangements, such as having subplots of different sizes or spanning across multiple rows or columns.

Code 1.11

```python
import matplotlib.gridspec as gridspec

# Create a figure and a gridspec layout
fig = plt.figure(figsize=(10, 8))
gs = gridspec.GridSpec(2, 2, figure=fig)

# Define subplots with specific grid positions
ax1 = fig.add_subplot(gs[0, 0])  # First subplot in position (0, 0)
ax2 = fig.add_subplot(gs[0, 1])  # First subplot in position (0, 1)
ax3 = fig.add_subplot(gs[1, :])  # Second subplot spanning across two columns (1, 0 and 1)
```

```

# Plotting on each subplot
ax1.plot([0, 1, 2, 3], [0, 1, 4, 9], color='red')
ax1.set_title('Plot 1: Line')

ax2.bar([1, 2, 3, 4], [5, 7, 3, 4], color='blue')
ax2.set_title('Plot 2: Bar')

ax3.scatter([1, 2, 3, 4], [5, 7, 3, 4], color='green')
ax3.set_title('Plot 3: Scatter')

# Adjust layout
plt.tight_layout()

plt.show()
```

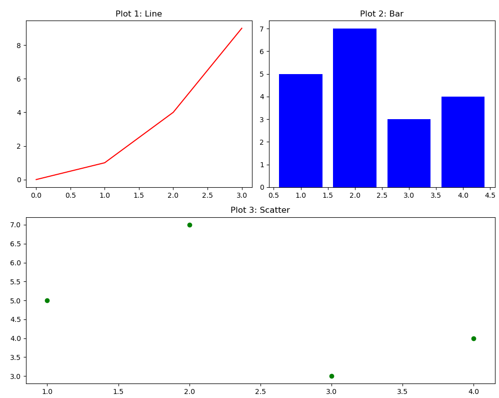

Figure 9: Multiple subplots using `gridspec`

Explanation: - We used `gridspec.GridSpec()` to define the layout of the subplots. In this case, the layout is 2 rows by 2 columns. - The subplot `ax3` spans across two columns in the second row by specifying `gs[1, :]`. - This allows more flexibility in designing the layout, with the ability to create larger plots in specific positions.

1.4.3 Adjusting Subplot Spacing and Layout

When working with multiple subplots, it is often necessary to adjust the spacing between them to ensure that the plots do not overlap or appear too close together. The `plt.tight_layout()` function is commonly used to automatically adjust spacing, but you can also manually adjust spacing using `plt.subplots_adjust()`.

Code 1.12

```
# Create a 2x2 grid of subplots
fig, axes = plt.subplots(2, 2, figsize=(10, 8))

# Plotting on each subplot
axes[0, 0].plot([0, 1, 2, 3], [0, 1, 4, 9], color='red')
axes[0, 0].set_title('Plot 1: Line')

axes[0, 1].bar([1, 2, 3, 4], [5, 7, 3, 4], color='blue')
axes[0, 1].set_title('Plot 2: Bar')

axes[1, 0].scatter([1, 2, 3, 4], [5, 7, 3, 4], color='green')
axes[1, 0].set_title('Plot 3: Scatter')

axes[1, 1].hist([1, 2, 2, 3, 3, 3, 4], bins=4, color='purple')
axes[1, 1].set_title('Plot 4: Histogram')
```

```python

# Adjust spacing manually
plt.subplots_adjust(wspace=0.4, hspace=0.4)  # Adjust horizontal and vertical spacing

plt.show()
```

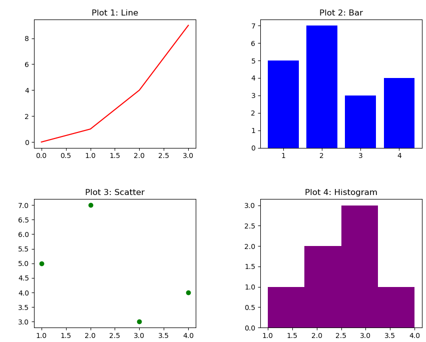

Figure 10: Adjusted subplots with custom spacing

Explanation: - The `plt.subplots_adjust()` function allows you to adjust the spacing between subplots. The `wspace` parameter controls the horizontal spacing, and `hspace` controls the vertical spacing. - This is useful when you need to make adjustments beyond what `tight_layout()` can do automatically.

1.5 Advanced Plotting Techniques

In addition to basic plotting, Matplotlib offers advanced plotting techniques that allow you to create more sophisticated and visually appealing charts. These techniques are particularly useful when you need to represent more complex data or emphasize specific patterns or relationships. In this chapter, we will cover the following:

- Stacked plots (stacked bar and line plots).
- Area charts for representing cumulative data.
- Pie charts for showing proportions.
- Error bars for showing uncertainty in data.
- Polar plots for visualizing data in polar coordinates.

Each technique has specific use cases, and we will explore examples to illustrate when and how to use them effectively.

1.5.1 Stacked Plots

Stacked plots allow you to visualize multiple data series on top of each other. This is particularly useful for displaying cumulative values or comparing parts to the whole.

Stacked Bar Plot:

A stacked bar plot is useful for comparing the composition of different categories over time or across different groups.

Code 1.13

```python
import matplotlib.pyplot as plt

# Data for stacked bar plot
categories = ['A', 'B', ' C', 'D']
values1 = [3, 7, 2, 5]
values2 = [4, 6, 8, 3]

# Create stacked bar plot
plt.bar(categories, values1, label='Series 1', color='blue')
plt.bar(categories, values2, bottom=values1, label='Series 2', color='orange')

# Customize plot
plt.title("Stacked Bar Plot")
plt.xlabel("Category")
plt.ylabel("Values")
plt.legend()

plt.show()
```

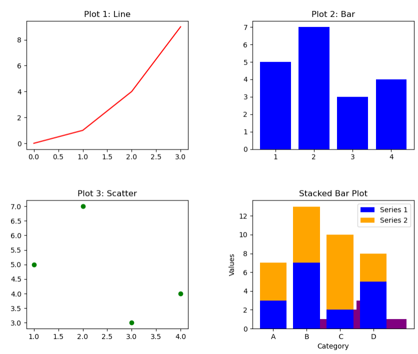

Figure 11: Stacked bar plot showing two series for each category

Explanation: - In the stacked bar plot, the bars are stacked on top of each other, allowing you to compare the contribution of each series to the total value for each category. - The `bottom` parameter in `plt.bar()` is used to stack the second series on top of the first.

Stacked Line Plot:

Stacked line plots work similarly to stacked bar plots but are useful when representing continuous data over time.

Code 1.14

```python
# Data for stacked line plot
y1 = [0, 1, 2, 3, 4]
y2 = [1, 2, 3, 4, 5]

# Create stacked line plot
plt.fill_between(x, y1, color="skyblue", alpha=0.5, label="Series 1")
plt.fill_between(x, y2, color="orange", alpha=0.5, label="Series 2")

# Customize plot
plt.title("Stacked Line Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.legend()

plt.show()
```

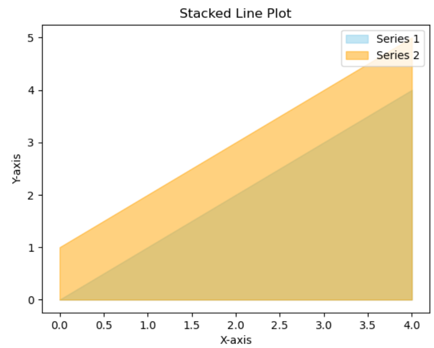

Figure 12: Stacked line plot showing the area between two series

Explanation: - The `fill_between()` function fills the area between two data points, creating a stacked effect. In this example, the two series are plotted as areas under the curves.

1.5.2 Area Charts

Area charts are useful for visualizing the cumulative total of data over a continuous range. They help emphasize the magnitude of change over time or other continuous variables.

Code 1.15

```
# Create an area chart
x = [0, 1, 2, 3, 4]
y1 = [1, 3, 5, 7, 9]
y2 = [2, 4, 6, 8, 10]

plt.fill_between(x, y1, color="skyblue", alpha=0.5, label="Series 1")
plt.fill_between(x, y2, color="orange", alpha=0.5, label="Series 2")

plt.title("Area Chart")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.legend()

plt.show()
```

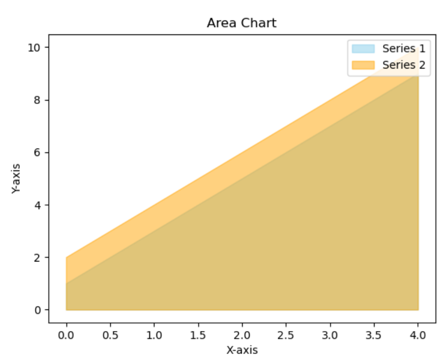

Figure 13: Area chart showing the cumulative total of two series

Explanation: - The `fill_between()` function is used to create the filled areas between the data series. The shaded areas represent the cumulative values over time, highlighting the contribution of each series.

1.5.3 Pie Charts

Pie charts are a simple way to show the proportions of a whole. Each slice of the pie represents a category's contribution to the total. While useful in some contexts, pie charts are often best suited for showing a limited number of categories.

Code 1.16

```
# Data for pie chart
labels = ['A', 'B', 'C', 'D']
sizes = [25, 35, 20, 20]

# Create a pie chart
plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=90)

plt.title("Pie Chart Example")
plt.show()
```

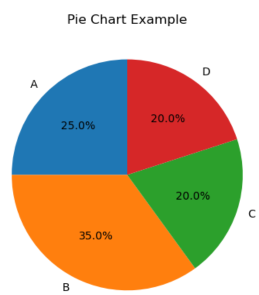

Figure 14: Pie chart showing proportions of different categories

Explanation: - `startangle=90` rotates the pie chart so the first slice starts from the top.

1.5.4 Error Bars

Error bars are used to represent uncertainty or variability in data. They can be added to various types of plots, such as line plots or scatter plots, to show the range of possible values.

Code 1.17

```
# Data for error bars
x = [0, 1, 2, 3, 4]
y = [0, 1, 4, 9, 16]
yerr = [0.5, 0.4, 0.3, 0.6, 0.4]  # Error values

# Create a plot with error bars
plt.errorbar(x, y, yerr=yerr, fmt='-o', color='blue', label="Data with error bars")

plt.title("Plot with Error Bars")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.legend()

plt.show()
```

Explanation: - The `plt.errorbar()` function adds error bars to the plot. The `yerr` parameter specifies the error values for the y-axis. - `fmt='-o'` specifies a line plot with circular markers.

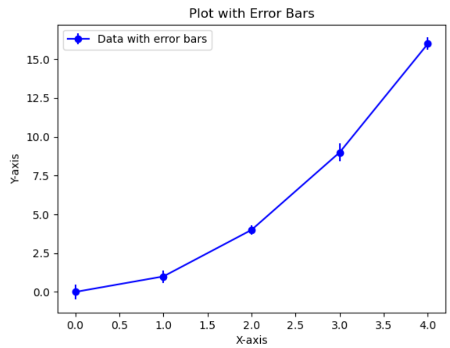

Figure 15: Line plot with error bars

1.5.5 Polar Plots

Polar plots are used for visualizing data in polar coordinates, where each data point is defined by a radius (distance from the origin) and an angle (angle from a reference axis). They are often used in applications such as radar or circular data analysis.

Code 1.18

```
# Data for polar plot
theta = [0, 1, 2, 3, 4]
r = [1, 2, 3, 4, 5]

# Create a polar plot
plt.polar(theta, r, color='red')

plt.title("Polar Plot Example")
plt.show()
```

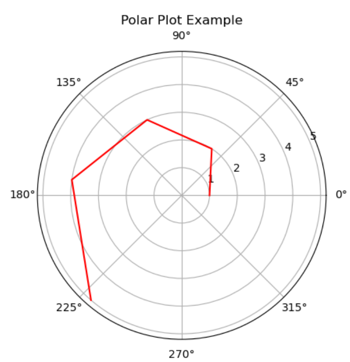

Figure 16: Polar plot showing data in polar coordinates

Explanation: - The `plt.polar()` function creates a polar plot, where `theta` represents the angle and `r` represents the radius of each data point.

1.6 Saving and Exporting Plots

Once you have created a plot in Matplotlib, it is often necessary to save the plot as an image file for later use. This is especially important when preparing figures for reports, presentations, or publications. Matplotlib offers several functions and options for saving plots to different file formats, adjusting the resolution, and customizing the output.

In this chapter, we will cover:

- Saving plots to various file formats (PNG, PDF, SVG, etc.).
- Adjusting the resolution (DPI).
- Specifying figure size and output quality.
- Saving interactive plots.

By the end of this chapter, you will be equipped with the tools to export your plots in high quality and in different formats, suitable for publishing or sharing.

1.6.1 Saving Plots to Different File Formats

Matplotlib allows you to save plots in a variety of formats, including PNG, PDF, SVG, and others. The `savefig()` function is used to save plots to a file. You can specify the file format by simply changing the file extension.

Here is an example of saving a plot as a PNG file.

Code 1.19

```
import matplotlib.pyplot as plt

# Data for plotting
x = [0, 1, 2, 3, 4]
y = [0, 1, 4, 9, 16]

# Create a simple plot
plt.plot(x, y)

# Save the plot as a PNG file
plt.savefig('figures/019.png')

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

Explanation: - The `savefig()` function is used to save the plot. By specifying the file name with a ".png" extension, the plot is saved in PNG format. - You can change the extension to `.pdf`, `.svg`, `.jpg`, or any other supported file format to save in different formats.

1.6.2 Adjusting Resolution (DPI)

Resolution is an important factor when saving plots, especially for high-quality publications. The `DPI` (dots per inch) setting controls the resolution of the saved image. Higher DPI values result in higher resolution images, which are better suited for printing or publications.

Code 1.20

```
# Save the plot with high resolution (DPI)
plt.plot(x, y)
plt.savefig('figures/020.png', dpi=300)  # Save with 300 DPI

plt.show()
```

Explanation: - The `dpi` parameter specifies the resolution of the saved image. In this case, we set the DPI to 300, which is suitable for high-quality printing. - A typical DPI for web images is 72, but for print, you might want to use a higher value, such as 300 or 600, depending on the quality requirements.

1.6.3 Specifying Figure Size

Matplotlib allows you to specify the figure size when saving a plot. This is useful if you need the plot to fit within certain dimensions (e.g., for publication or presentations). You can set the figure size either when creating the figure or when saving it.

Code 1.21

```
# Set the figure size before plotting
plt.figure(figsize=(10, 6))  # Width: 10 inches, Height: 6 inches

# Create a plot
plt.plot(x, y)

# Save the plot with the specified figure size
plt.savefig('figures/021.png', dpi=300)

plt.show()
```

Explanation: - The `figsize` parameter is used when creating the figure. It takes a tuple of width and height in inches. - By specifying a larger figure size, you can ensure that your plot fits well in the intended output format (e.g., reports or slides).

1.6.4 Saving Interactive Plots

Matplotlib also supports saving interactive plots, but this requires the use of the `plt.savefig()` function in combination with the correct interactive backend. While interactive plots can be displayed directly within Jupyter Notebooks or GUI applications, they can also be saved in formats that retain their interactivity, such as PDF or SVG.

Code 1.22

```
# Create an interactive plot
fig, ax = plt.subplots()
ax.plot(x, y)

# Save the interactive plot as a PDF (interactive plots are supported in PDF)
plt.savefig('figures/022.pdf')

plt.show()
```

Explanation: - The `savefig()` function saves interactive plots in formats such as PDF or SVG. These formats can preserve certain interactive features, such as zooming or panning, when viewed in appropriate viewers. - Note that not all formats (such as PNG or JPEG) support interactive features.

1.6.5 Vector Graphics (SVG, PDF, EPS)

For high-quality plots, especially when preparing figures for publication, it's often recommended to use vector graphics formats, such as SVG, PDF, or EPS. These formats scale well without loss of quality, making them ideal for printed materials.

Code 1.23

```python
# Save the plot as a vector graphic (SVG format)
plt.plot(x, y)
plt.savefig('figures/023.svg')  # Save as SVG (vector graphic)

plt.show()
```

Explanation: - Vector formats like SVG and PDF retain the quality of your plot at any zoom level, making them perfect for high-resolution publications. - Unlike raster graphics (e.g., PNG, JPEG), vector graphics are not pixel-based, which means they do not lose quality when resized.

1.6.6 Adjusting Image Quality and Transparency

You can adjust the quality of the saved image by setting the `quality` parameter when saving in JPEG format, or you can make the image transparent by adjusting the `transparent` parameter.

Code 1.24

```python
# Save the plot with transparent background (for PNG and SVG)
plt.plot(x, y)
plt.savefig('figures/024.png', transparent=True)

# Save the plot with specific quality (for JPEG)
plt.savefig('figures/025.jpg', quality=95)

plt.show()
```

Explanation: - The `transparent=True` parameter makes the background of the plot transparent, which is useful when overlaying plots on different backgrounds (e.g., for presentations). - The `quality` parameter controls the quality of JPEG images. Higher values result in better quality, but larger file sizes.

2 Datetime — Basic date and time types

2.1 Introduction to the `datetime` Module

Dates and times are essential components of many programs, from climate and atmospheric applications to scientific computing. Python's `datetime` module provides a robust framework for working with dates and times, making it easier to perform operations such as date arithmetic, formatting, and parsing. In this chapter, we will explore:

- Overview of the `datetime` module.
- Key classes in the `datetime` module.
- Basic usage for creating and manipulating date and time objects.

By the end of this chapter, you will have a solid understanding of how to use Python's `datetime` module for handling date and time data.

2.1.1 Overview of the `datetime` Module

The `datetime` module provides several classes to represent dates, times, and intervals. It supports operations like date and time arithmetic, comparison, and conversion between different formats. Some of the key classes in the module include:

- `datetime`: Combines both date and time into a single object.
- `date`: Represents the date (year, month, day) without the time.
- `time`: Represents the time (hour, minute, second, microsecond) without the date.
- `timedelta`: Represents the difference between two dates or times.
- `tzinfo`: A base class for dealing with time zone information.

These classes allow you to perform a wide range of operations, such as getting the current date and time, calculating the difference between two dates, and formatting dates in various ways.

2.1.2 Key Classes in `datetime`

Let's explore the core classes provided by the `datetime` module.

The `datetime` Class:

The `datetime` class is the most comprehensive class in the module. It represents a specific moment in time, combining both the date and the time. You can create a `datetime` object by passing the year, month, day, hour, minute, second, and microsecond.

Code 2.1

```python
import datetime

# Create a datetime object for a specific date and time
dt = datetime.datetime(2024, 11, 17, 15, 30, 0)
print("Datetime Object:", dt)
```

```
Datetime Object: 2024-11-17 15:30:00
```

Explanation: - The `datetime` object is created using the `datetime.datetime()` constructor. - The object represents the date and time `November 17, 2024, 3:30:00 PM`.

The `date` Class:

The `date` class represents the date (year, month, and day) without any time information. It can be created using `datetime.date()`.

Code 2.2

```
# Create a date object
d = datetime.date(2024, 11, 17)
print("Date Object:", d)
```

```
Date Object: 2024-11-17
```

Explanation: - The `date` object represents the date `November 17, 2024`. - It excludes any time-related data, such as hour, minute, or second.

The `time` Class:

The `time` class represents the time of day (hour, minute, second, microsecond), but it does not include the date.

Code 2.3

```
# Create a time object
t = datetime.time(15, 30, 0)
print("Time Object:", t)
```

```
Time Object: 15:30:00
```

Explanation: - The `time` object represents the time `15:30:00` (or 3:30 PM) without any date-related information.

The `timedelta` Class:

The `timedelta` class represents a difference between two `datetime` objects. It is often used for date arithmetic, such as adding or subtracting days, hours, or minutes.

Code 2.4

```
# Create a timedelta object
delta = datetime.timedelta(days=5, hours=3)
print("Timedelta Object:", delta)
```

```
Timedelta Object: 5 days, 3:00:00
```

Explanation: - The `timedelta` object represents a time difference of 5 days and 3 hours. - This class is useful when performing operations like adding 5 days to a given date or calculating the difference between two dates.

2.1.3 Basic Usage of `datetime`

Now that we have introduced the main classes, let's explore how to use them for common operations like getting the current date and time, comparing dates, and performing simple date arithmetic.

Getting the Current Date and Time:

You can get the current date and time by using the `datetime.now()` method.

Code 2.5

```
# Get the current date and time
now = datetime.datetime.now()
print("Current Date and Time:", now)
```

```
Current Date and Time: 2024-11-17 15:30:00.123456
```

Explanation: - The `now()` method returns the current date and time, including microseconds. - You can use this method to get the current timestamp for use in various calculations.

Extracting Components from a `datetime` Object:

Once you have a `datetime` object, you can extract individual components like the year, month, day, hour, minute, and second.

Code 2.6

```
# Extract components from a datetime object
year = now.year
month = now.month
day = now.day
hour = now.hour
minute = now.minute
second = now.second

print(f"Year: {year}, Month: {month}, Day: {day}, Hour: {hour}, Minute: {minute},
    Second: {second}")
```

```
Year: 2024, Month: 11, Day: 17, Hour: 15, Minute: 30, Second: 0
```

Explanation: - You can easily access the components of a `datetime` object using attributes such as `year`, `month`, and `hour`.

Performing Date Arithmetic with `timedelta`:

You can use `timedelta` objects to perform arithmetic operations on dates and times. For example, you can add or subtract days from a `datetime` object.

Code 2.7

```
# Add 5 days to the current date
new_date = now + datetime.timedelta(days=5)
print("New Date after Adding 5 Days:", new_date)

# Subtract 3 hours from the current time
new_time = now - datetime.timedelta(hours=3)
print("New Time after Subtracting 3 Hours:", new_time)
```

```
New Date after Adding 5 Days: 2024-11-22 15:30:00.123456
New Time after Subtracting 3 Hours: 2024-11-17 12:30:00.123456
```

Explanation: - By using `timedelta`, we can add or subtract a specific amount of time from a `datetime` object. In this case, we added 5 days and subtracted 3 hours.

2.2 Working with Date Objects

The `date` class in Python's `datetime` module is used to represent a calendar date without the time component. It is particularly useful when you need to work only with the date (year, month, day) and not the time of day. In this chapter, we will explore:

- Creating date objects.
- Accessing date components.
- Performing date arithmetic (adding and subtracting days).
- Comparing date objects.
- Handling today's date and working with the current date.

By the end of this chapter, you will have a good understanding of how to work with dates in Python and how to perform basic operations on date objects.

2.2.1 Creating Date Objects

You can create a `date` object by using the `datetime.date()` constructor, which takes three arguments: year, month, and day. Here's an example of creating a date object for November 17, 2024.

Code 2.8

```python
import datetime

# Create a date object for November 17, 2024
d = datetime.date(2024, 11, 17)
print("Date Object:", d)
```

```
Date Object: 2024-11-17
```

Explanation: - The `datetime.date()` constructor is used to create a date object. We passed the year (`2024`), month (`11`), and day (`17`) to create the date `2024-11-17`. - The resulting object is a date without any time information.

2.2.2 Accessing Date Components

Once you have a `date` object, you can access individual components like the year, month, and day using the corresponding attributes.

Code 2.9

```python
# Access components of the date object
year = d.year
month = d.month
day = d.day

print(f"Year: {year}, Month: {month}, Day: {day}")
```

```
Year: 2024, Month: 11, Day: 17
```

Explanation: - The `year`, `month`, and `day` attributes allow you to extract specific components from a `date` object.

2.2.3 Performing Date Arithmetic

You can perform arithmetic on date objects using the `timedelta` class. `timedelta` represents the difference between two dates or times. You can add or subtract days from a `date` object by using `timedelta`.

Adding and Subtracting Days:

You can add or subtract a number of days to/from a date using `timedelta`. Here's an example of adding and subtracting days from a date.

Code 2.10

```
# Add 5 days to the date
delta = datetime.timedelta(days=5)
new_date = d + delta
print("Date after Adding 5 Days:", new_date)

# Subtract 10 days from the date
delta_subtract = datetime.timedelta(days=10)
new_date_subtract = d - delta_subtract
print("Date after Subtracting 10 Days:", new_date_subtract)
```

```
Date after Adding 5 Days: 2024-11-22
Date after Subtracting 10 Days: 2024-11-07
```

Explanation: - The `timedelta(days=5)` creates a time difference of 5 days, which we then add to the original date `2024-11-17`. - Similarly, we subtract 10 days using `timedelta(days=10)`.

2.2.4 Comparing Date Objects

You can compare `date` objects using standard comparison operators (==, !=, ¡, ¡=, ¿, ¿=). This is useful when you need to check if one date is earlier or later than another.

Code 2.11

```
# Create another date object
d2 = datetime.date(2024, 11, 25)

# Compare the two dates
print("Is d before d2?", d < d2)
print("Is d equal to d2?", d == d2)
```

```
Is d before d2? True
Is d equal to d2? False
```

Explanation: - The comparison operators allow you to compare two date objects. In this case, `d` (2024-11-17) is earlier than `d2` (2024-11-25), so the first comparison is `True`. - The second comparison checks whether `d` is equal to `d2`, which is `False` because the dates are different.

2.2.5 Getting Today's Date

The `datetime.date.today()` method allows you to get the current date according to the system's local time.

Code 2.12

```
# Get today's date
today = datetime.date.today()
print("Today's Date:", today)
```

```
Today's Date: 2024-11-17
```

Explanation: - The `datetime.date.today()` method returns the current date (without time) according to the system's local time.

2.3 Working with Time Objects

In Python, the `time` class from the `datetime` module is used to represent the time of day (hours, minutes, seconds, and microseconds) without the associated date. This chapter focuses on:

- Creating time objects.
- Accessing components of time objects (hours, minutes, seconds).
- Performing time arithmetic (adding and subtracting time).
- Working with time intervals.

By the end of this chapter, you will be able to create, manipulate, and perform arithmetic on time objects in Python.

2.3.1 Creating Time Objects

The `time` class represents the time portion of the day (i.e., hour, minute, second, and microsecond). You can create a time object by using the `datetime.time()` constructor, which takes up to four arguments: hour, minute, second, and microsecond.

Code 2.13

```python
import datetime

# Create a time object for 3:30:00
t = datetime.time(15, 30, 0)
print("Time Object:", t)
```

```
Time Object: 15:30:00
```

Explanation: - The `time()` constructor creates a time object. In this example, the time object represents `15:30:00` (3:30 PM). - You can also include microseconds (if needed) by passing a value for the microsecond parameter (default is 0).

2.3.2 Accessing Components of Time Objects

Once you have created a time object, you can access its components: hours, minutes, seconds, and microseconds using the corresponding attributes.

Code 2.14

```python
# Access components of the time object
hour = t.hour
minute = t.minute
second = t.second
microsecond = t.microsecond

print(f"Hour: {hour}, Minute: {minute}, Second: {second}, Microsecond: {microsecond}")
```

```
Hour: 15, Minute: 30, Second: 0, Microsecond: 0
```

Explanation: - The `hour`, `minute`, `second`, and `microsecond` attributes allow you to extract specific components from a `time` object.

2.3.3 Performing Time Arithmetic

Just like `date` objects, `time` objects can be manipulated using `timedelta`. However, because `time` objects represent a specific point in time during the day, performing arithmetic on them may result in a `ValueError` unless you account for crossing over to the next day.

Adding and Subtracting Time:

You can add or subtract time from a `time` object using `timedelta`, but keep in mind that you may need to handle the case where the time goes beyond the 24-hour limit.

Code 2.15

```python
# Add 1 hour and 30 minutes to the time
delta = datetime.timedelta(hours=1, minutes=30)
new_time = (datetime.datetime.combine(datetime.date.today(), t) + delta).time()
print("New Time after Adding 1 Hour 30 Minutes:", new_time)

# Subtract 2 hours from the time
delta_subtract = datetime.timedelta(hours=2)
new_time_subtract = (datetime.datetime.combine(datetime.date.today(), t) -
    delta_subtract).time()
print("New Time after Subtracting 2 Hours:", new_time_subtract)
```

```
New Time after Adding 1 Hour 30 Minutes: 17:00:00
New Time after Subtracting 2 Hours: 13:30:00
```

Explanation: - To perform time arithmetic, we first combine the `time` object with a `date` object using `datetime.combine()`. This gives us a full `datetime` object that can be used in arithmetic operations. - After performing the arithmetic, we convert the result back to a `time` object using the `.time()` method.

2.3.4 Handling Time Intervals

Time intervals are a common task when working with time data, especially when you need to compute differences between time objects. You can use `timedelta` to represent the difference between two `time` objects, but it's important to remember that `timedelta` works with both date and time objects, and can span over multiple days if necessary.

Code 2.16

```python
# Time difference between two time objects
t1 = datetime.time(8, 30, 0)  # 8:30 AM
t2 = datetime.time(14, 45, 0)  # 2:45 PM

# Convert time objects to datetime objects to perform subtraction
delta_time = (datetime.datetime.combine(datetime.date.today(), t2) -
    datetime.datetime.combine(datetime.date.today(), t1)).total_seconds()
print(f"Time Difference in Seconds: {delta_time} seconds")
```

```
Time Difference in Seconds: 22500.0 seconds
```

Explanation: - In this example, we calculate the difference between two `time` objects, `t1` and `t2`, by converting them into `datetime` objects and then subtracting them. - The `total_seconds()` method of the `timedelta` object returns the difference in seconds.

2.4 Working with `datetime` Objects

The `datetime` class in Python's `datetime` module combines both the date and the time into a single object. It is the most comprehensive class in the module, allowing you to perform various date-time operations, including arithmetic, comparison, and extraction of components. In this chapter, we will explore:

- Creating `datetime` objects.
- Accessing components of `datetime` objects (year, month, day, hour, minute, second).
- Performing `datetime` arithmetic (adding and subtracting time).
- Comparing `datetime` objects.
- Working with time zones and UTC.

By the end of this chapter, you will be able to create and manipulate `datetime` objects, perform arithmetic, and extract date-time components for analysis.

2.4.1 Creating `datetime` Objects

A `datetime` object represents a specific point in time and is created by combining a date and a time. You can create a `datetime` object by passing the year, month, day, hour, minute, second, and microsecond as arguments.

Code 2.17

```
import datetime

# Create a datetime object for November 17, 2024, 15:30:00
dt = datetime.datetime(2024, 11, 17, 15, 30, 0)
print("Datetime Object:", dt)
```

```
Datetime Object: 2024-11-17 15:30:00
```

Explanation: - The `datetime.datetime()` constructor is used to create a `datetime` object. We passed the year (`2024`), month (`11`), day (`17`), hour (`15`), minute (`30`), and second (`0`). - The resulting object represents the date and time `2024-11-17 15:30:00`.

2.4.2 Accessing Components of `datetime` Objects

Once you have a `datetime` object, you can easily access individual components such as the year, month, day, hour, minute, second, and microsecond.

Code 2.18

```
# Access components of the datetime object
year = dt.year
month = dt.month
day = dt.day
hour = dt.hour
```

```python
minute = dt.minute
second = dt.second
microsecond = dt.microsecond

print(f"Year: {year}, Month: {month}, Day: {day}, Hour: {hour}, Minute: {minute},
    Second: {second}, Microsecond: {microsecond}")
```

```
Year: 2024, Month: 11, Day: 17, Hour: 15, Minute: 30, Second: 0, Microsecond: 0
```

Explanation: - The `year`, `month`, `day`, `hour`, `minute`, `second`, and `microsecond` attributes allow you to extract specific components from a `datetime` object.

2.4.3 Performing `datetime` Arithmetic

You can perform arithmetic operations on `datetime` objects using the `timedelta` class. `timedelta` represents a duration, which can be added to or subtracted from a `datetime` object.

Adding and Subtracting Time:

You can add or subtract days, hours, minutes, and other time intervals from a `datetime` object using `timedelta`.

Code 2.19

```python
# Add 5 days to the datetime object
delta = datetime.timedelta(days=5)
new_dt = dt + delta
print("Datetime after Adding 5 Days:", new_dt)

# Subtract 3 hours from the datetime object
delta_subtract = datetime.timedelta(hours=3)
new_dt_subtract = dt - delta_subtract
print("Datetime after Subtracting 3 Hours:", new_dt_subtract)
```

```
Datetime after Adding 5 Days: 2024-11-22 15:30:00
Datetime after Subtracting 3 Hours: 2024-11-17 12:30:00
```

Explanation: - By using `timedelta`, we can perform date-time arithmetic. In this case, we added 5 days to the original `datetime` object and subtracted 3 hours. - The `timedelta` class can represent various time intervals such as days, hours, minutes, seconds, and microseconds.

2.4.4 Comparing `datetime` Objects

You can compare `datetime` objects using the standard comparison operators (`==`, `!=`, `<`, `<=`, `>`, `>=`). This is useful when you need to check if one date-time is earlier, later, or the same as another.

Code 2.20

```python
# Create another datetime object
dt2 = datetime.datetime(2024, 11, 25, 15, 30, 0)

# Compare the two datetime objects
print("Is dt before dt2?", dt < dt2)
print("Is dt equal to dt2?", dt == dt2)
```

```
Is dt before dt2? True
Is dt equal to dt2? False
```

Explanation: - Comparison operators allow you to compare two `datetime` objects. In this case, `dt` (2024-11-17) is earlier than `dt2` (2024-11-25), so the first comparison is `True`. - The second comparison checks whether `dt` is equal to `dt2`, which is `False` because the dates are different.

2.4.5 Working with Time Zones and UTC

Matplotlib also supports working with time zones and UTC time. The `datetime` module has built-in support for dealing with time zones, although it's often useful to use an external library such as `pytz` for more advanced time zone operations.

Code 2.21

```
# Working with UTC time
utc_now = datetime.datetime.utcnow()
print("Current UTC Time:", utc_now)

# Convert to a specific timezone (e.g., US Eastern Time)
import pytz
eastern = pytz.timezone('US/Eastern')
eastern_time = utc_now.astimezone(eastern)
print("Eastern Time:", eastern_time)
```

Explanation: - The `utcnow()` method gets the current UTC time. This time does not account for time zone differences. - We can convert the UTC time to a specific time zone (e.g., US Eastern Time) using the `astimezone()` method and an external library like `pytz`.

2.5 Formatting and Parsing Dates and Times

Working with dates and times often requires converting them between different formats. For example, you might need to display a 'datetime' object in a human-readable format or convert a string representing a date into a 'datetime' object. Python's `datetime` module provides powerful functions for formatting and parsing dates and times. In this chapter, we will explore:

- Formatting 'datetime' objects into strings using `strftime()`.
- Parsing strings into 'datetime' objects using `strptime()`.
- Commonly used date-time format codes.
- Handling time zones in formatted strings.

By the end of this chapter, you will be able to format and parse dates and times in Python, making it easier to handle time-related data in different formats.

2.5.1 Formatting 'datetime' Objects with `strftime()`

The `strftime()` method allows you to format a 'datetime' object into a string representation. You can specify a format string, which uses various formatting codes to represent the components of the 'datetime' object (such as the year, month, day, etc.).

Common Format Codes:

- %Y: Year with century (e.g., 2024)

- %m: Month as a zero-padded decimal number (e.g., 01 for January)
- %d: Day of the month as a zero-padded decimal number (e.g., 01)
- %H: Hour (24-hour clock) as a zero-padded decimal number (e.g., 15 for 3 PM)
- %M: Minute as a zero-padded decimal number (e.g., 30)
- %S: Second as a zero-padded decimal number (e.g., 59)
- %f: Microsecond as a decimal number (e.g., 123456)

Example: Formatting a 'datetime' Object

Let's start by formatting a 'datetime' object to display it in a more readable form.

Code 2.22

```
import datetime

# Create a datetime object
dt = datetime.datetime(2024, 11, 17, 15, 30, 0)

# Format the datetime object to a string
formatted_date = dt.strftime("%Y-%m-%d %H:%M:%S")
print("Formatted Date:", formatted_date)
```

```
Formatted Date: 2024-11-17 15:30:00
```

Explanation: - The `strftime()` method converts the `datetime` object into a string formatted as `YYYY-MM-DD HH:MM:SS`. - You can customize the format string to display the components in any order, separated by symbols of your choice (such as slashes, dashes, or colons).

2.5.2 Parsing Strings into 'datetime' Objects with `strptime()`

The `strptime()` method allows you to parse a string representation of a date and time and convert it into a 'datetime' object. This is particularly useful when working with dates and times in string format (such as those coming from user input, files, or APIs).

Example: Parsing a Date String into a 'datetime' Object

Let's parse a string that represents a date-time and convert it into a 'datetime' object.

Code 2.23

```
# String representing a date
date_string = "2024-11-17 15:30:00"

# Parse the string into a datetime object
parsed_date = datetime.datetime.strptime(date_string, "%Y-%m-%d %H:%M:%S")
print("Parsed Date:", parsed_date)
```

```
Parsed Date: 2024-11-17 15:30:00
```

Explanation: - The `strptime()` function takes two arguments: the string to be parsed and the format string that specifies the format of the input string. - The format string `"%Y-%m-%d %H:%M:%S"` corresponds to the format of the input string (`"2024-11-17 15:30:00"`), and `strptime()` returns a 'datetime' object.

2.5.3 Handling Time Zones in Formatted Strings

Time zone handling is an important aspect of working with dates and times, especially when dealing with users in different time zones or when working with international data. While Python's `datetime` module has some support for time zones through the `tzinfo` class, you can also display the time zone in your formatted strings.

Example: Formatting with Time Zones

Here is an example that includes a time zone using the `strftime()` method:

Code 2.24

```
# Add time zone info to a datetime object
import pytz

# Create a datetime object with a time zone
timezone = pytz.timezone('US/Eastern')
dt_with_timezone = timezone.localize(datetime.datetime(2024, 11, 17, 15, 30, 0))

# Format datetime with time zone info
formatted_with_timezone = dt_with_timezone.strftime("%Y-%m-%d %H:%M:%S %Z%z")
print("Formatted Date with Timezone:", formatted_with_timezone)
```

```
Formatted Date with Timezone: 2024-11-17 15:30:00 EST-0500
```

Explanation: - We use the `pytz` library to localize the `datetime` object to a specific time zone (in this case, US Eastern Time). - The format code `%Z` is used to represent the time zone abbreviation (e.g., `EST` for Eastern Standard Time), and `%z` represents the time zone offset from UTC (e.g., `-0500`).

2.5.4 Common Format Codes

Here is a list of some commonly used format codes that you can use with both `strftime()` and `strptime()`:

- %Y: Year with century (e.g., 2024)
- %m: Month as a zero-padded decimal number (01, 02, . . . , 12)
- %d: Day of the month as a zero-padded decimal number (01, 02, . . . , 31)
- %H: Hour (24-hour clock) as a zero-padded decimal number (00, 01, . . . , 23)
- %M: Minute as a zero-padded decimal number (00, 01, . . . , 59)
- %S: Second as a zero-padded decimal number (00, 01, . . . , 59)
- %f: Microsecond as a decimal number (000000, 000001, . . . , 999999)
- %Z: Time zone abbreviation (e.g., UTC, PST, EST)
- %z: UTC offset in the form +HHMM or -HHMM (e.g., +0000, -0500)
- %A: Weekday name (e.g., Monday, Tuesday)
- %B: Month name (e.g., January, February)

2.6 Time Zones and UTC

Time zone handling is an essential part of many applications, especially when working with international data or users in different locations. Python's `datetime` module provides basic functionality for working with time zones through the `tzinfo` class, but it often requires the use of external libraries like `pytz` to handle time zones more effectively. In this chapter, we will explore:

- Understanding UTC (Coordinated Universal Time).
- Working with time zones using `pytz`.
- Converting between time zones.
- Handling daylight saving time (DST).
- Working with naive and aware datetime objects.

By the end of this chapter, you will have a strong understanding of how to work with time zones and UTC in Python and how to convert between time zones.

2.6.1 Understanding UTC

UTC (Coordinated Universal Time) is the standard for timekeeping worldwide and is not affected by daylight saving time (DST). It is often used as a reference time and provides the foundation for time zone calculations. In Python, you can get the current UTC time using the `utcnow()` method.

Code 2.25

```python
import datetime

# Get the current UTC time
utc_now = datetime.datetime.utcnow()
print("Current UTC Time:", utc_now)
```

```
Current UTC Time: 2024-11-17 20:30:00.123456
```

Explanation: - The `utcnow()` method returns the current time in UTC. Unlike the local time, UTC does not account for time zone differences or daylight saving time.

2.6.2 Working with Time Zones Using pytz

The `pytz` library is a third-party Python package that allows for robust handling of time zones. It provides a way to work with time zone-aware datetime objects and makes it easy to convert between time zones. You can install `pytz` using the following command:

```
pip install pytz
```

Once installed, you can use `pytz` to localize a `datetime` object to a specific time zone.

Code 2.26

```python
import pytz

# Create a datetime object without time zone information (naive)
naive_datetime = datetime.datetime(2024, 11, 17, 15, 30, 0)
```

```
# Localize the datetime object to US Eastern Time (using pytz)
eastern = pytz.timezone('US/Eastern')
localized_datetime = eastern.localize(naive_datetime)

print("Localized Date and Time (US/Eastern):", localized_datetime)
```

```
Localized Date and Time (US/Eastern): 2024-11-17 15:30:00-05:00
```

Explanation: - A naive datetime object is one that does not have time zone information associated with it. - The `localize()` method from `pytz` attaches time zone information to the naive datetime, making it aware of the time zone (in this case, US Eastern Time).

2.6.3 Converting Between Time Zones

Once you have a time zone-aware `datetime` object, you can easily convert it to another time zone using the `astimezone()` method.

Code 2.27

```
# Convert the localized datetime to UTC
utc_time = localized_datetime.astimezone(pytz.utc)
print("Converted to UTC:", utc_time)

# Convert the localized datetime to another time zone (e.g., Asia/Kolkata)
kolkata = pytz.timezone('Asia/Kolkata')
kolkata_time = localized_datetime.astimezone(kolkata)
print("Converted to Kolkata Time:", kolkata_time)
```

```
Converted to UTC: 2024-11-17 20:30:00+00:00
Converted to Kolkata Time: 2024-11-17 01:00:00+05:30
```

Explanation: - The `astimezone()` method converts a time zone-aware `datetime` object from one time zone to another. - In this case, we converted the `datetime` from US Eastern Time to UTC and then to Kolkata time.

2.6.4 Handling Daylight Saving Time (DST)

Daylight Saving Time (DST) is the practice of moving the clock forward in the spring and back in the fall to extend evening daylight in warmer months. Time zone conversions can be tricky when DST is in effect, as the time zone offset may change.

Code 2.28

```
# Convert a datetime during daylight saving time
dst_datetime = eastern.localize(datetime.datetime(2024, 6, 15, 15, 30, 0), is_dst=True)
print("Datetime in DST:", dst_datetime)

# Convert to UTC
utc_dst = dst_datetime.astimezone(pytz.utc)
print("Converted to UTC (DST):", utc_dst)
```

```
Datetime in DST: 2024-06-15 15:30:00-04:00
Converted to UTC (DST): 2024-06-15 19:30:00+00:00
```

Explanation: - When localizing a datetime during DST, you pass the `is_dst=True` argument to indicate that the time is during daylight saving time. - In this example, the datetime is correctly adjusted to reflect the DST time zone offset.

2.6.5 Naive vs Aware Datetime Objects

A **naive** datetime object is one that does not contain any time zone information. It represents a point in time, but it is not associated with any specific time zone. An **aware** datetime object, on the other hand, includes time zone information, allowing it to be properly converted between time zones and handle daylight saving time.

Example: Naive and Aware 'datetime' Objects

Code 2.29

```
# Naive datetime object (no time zone information)
naive_datetime = datetime.datetime(2024, 11, 17, 15, 30, 0)

# Aware datetime object (with time zone information)
aware_datetime = eastern.localize(naive_datetime)

print("Naive Datetime:", naive_datetime)
print("Aware Datetime:", aware_datetime)
```

```
Naive Datetime: 2024-11-17 15:30:00
Aware Datetime: 2024-11-17 15:30:00-05:00
```

Explanation: - The naive datetime object does not have any time zone information, while the aware datetime object is localized to the Eastern Time zone, making it time zone-aware.

2.7 Date and Time Arithmetic with timedelta

In many applications, it is necessary to perform arithmetic with dates and times. The `timedelta` class in Python's `datetime` module allows you to perform arithmetic operations on 'datetime' and 'date' objects. This chapter will introduce you to:

- Creating `timedelta` objects.
- Performing date and time arithmetic (adding and subtracting days, hours, etc.).
- Using `timedelta` for comparing dates and times.
- Working with larger time intervals (weeks, months, years).

By the end of this chapter, you will be comfortable using `timedelta` for manipulating and calculating date and time values.

2.7.1 Creating timedelta Objects

A `timedelta` object represents a duration, i.e., the difference between two dates or times. You can create a `timedelta` object by specifying days, seconds, microseconds, hours, minutes, and weeks. Here is an example:

Code 2.30

```
import datetime

# Create a timedelta object representing 5 days, 3 hours, and 30 minutes
delta = datetime.timedelta(days=5, hours=3, minutes=30)
print("Timedelta Object:", delta)
```

```
Timedelta Object: 5 days, 3:30:00
```

Explanation: - A `timedelta` object represents a specific time duration. In this example, we created a `timedelta` object that represents 5 days, 3 hours, and 30 minutes. - `timedelta` accepts several arguments, such as `days`, `hours`, `minutes`, and `seconds`, allowing you to specify any time duration.

2.7.2 Performing Date and Time Arithmetic

You can perform arithmetic on `datetime` and `date` objects by adding or subtracting `timedelta` objects. Here's an example of how to add and subtract days from a 'datetime' object:

Adding Time to a 'datetime' Object:

Code 2.31

```
# Create a datetime object
dt = datetime.datetime(2024, 11, 17, 15, 30)

# Add 5 days to the datetime
new_dt_add = dt + datetime.timedelta(days=5)
print("Datetime after Adding 5 Days:", new_dt_add)
```

```
Datetime after Adding 5 Days: 2024-11-22 15:30:00
```

Explanation: - By adding a `timedelta` object representing 5 days to the `datetime` object, the date becomes `2024-11-22`.

Subtracting Time from a 'datetime' Object:

Code 2.32

```
# Subtract 3 hours from the datetime
new_dt_subtract = dt - datetime.timedelta(hours=3)
print("Datetime after Subtracting 3 Hours:", new_dt_subtract)
```

```
Datetime after Subtracting 3 Hours: 2024-11-17 12:30:00
```

Explanation: - By subtracting a `timedelta` object representing 3 hours from the `datetime` object, the time becomes `12:30 PM`.

2.7.3 Using `timedelta` with 'date' Objects

You can also use `timedelta` objects with `date` objects to perform date arithmetic. The following example shows how to add or subtract days from a `date` object.

Code 2.33

```
# Create a date object
d = datetime.date(2024, 11, 17)

# Add 10 days to the date
new_date_add = d + datetime.timedelta(days=10)
print("Date after Adding 10 Days:", new_date_add)

# Subtract 7 days from the date
new_date_subtract = d - datetime.timedelta(days=7)
print("Date after Subtracting 7 Days:", new_date_subtract)
```

```
Date after Adding 10 Days: 2024-11-27
Date after Subtracting 7 Days: 2024-11-10
```

Explanation: - You can add or subtract a `timedelta` object representing a number of days to/from a `date` object. In this case, we added and subtracted 10 and 7 days, respectively.

2.7.4 Working with Larger Time Intervals

`timedelta` objects can represent larger time intervals such as weeks, months, or years. While `timedelta` has built-in support for weeks, months and years typically need to be handled manually since they are not fixed in length.

Example: Working with Weeks:

Code 2.34

```
# Add 3 weeks to the datetime
new_dt_weeks = dt + datetime.timedelta(weeks=3)
print("Datetime after Adding 3 Weeks:", new_dt_weeks)
```

```
Datetime after Adding 3 Weeks: 2024-12-08 15:30:00
```

Explanation: - The `timedelta` object accepts a `weeks` argument, which allows you to easily add or subtract weeks from a 'datetime' or 'date' object.

Note: Handling Months and Years: Months and years are not fixed durations (months vary in length and leap years affect years), so you need to handle these manually. You can use the `dateutil.relativedelta` module to add months or years.

2.7.5 Comparing Dates and Times with `timedelta`

You can use `timedelta` objects to compare two `datetime` or `date` objects. By subtracting two `datetime` or `date` objects, you obtain a `timedelta` object representing the difference between them.

Code 2.35

```
# Create two datetime objects
dt1 = datetime.datetime(2024, 11, 17, 15, 30, 0)
dt2 = datetime.datetime(2024, 11, 22, 15, 30, 0)

# Calculate the difference between the two datetimes
difference = dt2 - dt1
print("Difference in Days and Time:", difference)
```

```
Difference in Days and Time: 5 days, 0:00:00
```

Explanation: - Subtracting two `datetime` objects returns a `timedelta` object representing the difference between them. In this case, the difference is 5 days.

2.8 Handling Periods and Intervals

In many applications, we need to represent and manipulate periods (fixed lengths of time) and intervals (differences between two points in time). These are especially important in fields such as time series analysis, financial modeling, and scheduling. Python's `datetime` module offers basic functionality for performing operations on dates and times, but to handle more complex recurring time periods (like months, quarters, or years), you often need to rely on external libraries such as `pandas` or `dateutil`. In this chapter, we will explore:

- Understanding and working with time periods.
- Handling intervals with `timedelta`.
- Using `pandas` Period and Timedelta objects for advanced interval handling.
- Working with relative periods (like adding months or years).
- Common use cases of periods and intervals in real-world applications.

By the end of this chapter, you will have a deep understanding of how to work with and manipulate periods and intervals in Python.

2.8.1 Understanding Periods and Intervals

In date and time operations, **periods** refer to recurring lengths of time, such as days, months, or years, whereas **intervals** refer to the difference between two specific points in time.

For example:

- A period could be 1 week, 3 months, or 5 years.
- An interval could be the difference between two dates, such as 3 days, or the number of hours between two times.

While `timedelta` handles intervals (differences between two points in time), handling periods (like months or years) requires more advanced handling because these periods are not fixed in duration. For instance, a month can have 28, 29, 30, or 31 days.

2.8.2 Handling Intervals with `timedelta`

The `timedelta` class allows us to represent a difference between two dates or times. It supports operations like adding or subtracting days, hours, minutes, seconds, and microseconds.

Example: Calculating an Interval

Let's subtract two `datetime` objects to get a `timedelta` object representing the interval between them.

Code 2.36

```
import datetime

# Create two datetime objects
dt1 = datetime.datetime(2024, 11, 17, 15, 30)
dt2 = datetime.datetime(2024, 11, 20, 15, 30)

# Calculate the interval between two datetime objects
interval = dt2 - dt1
print("Interval between Dates:", interval)
```

```
Interval between Dates: 3 days, 0:00:00
```

Explanation: - Subtracting two `datetime` objects returns a `timedelta` object, which represents the difference between the two dates and times. - In this example, the interval is 3 days.

2.8.3 Working with Periods Using pandas

Python's pandas library provides more advanced tools for working with periods. The Period and Timedelta classes in pandas allow for handling time-based data, including managing periods like months, quarters, and years, which are often necessary in time series analysis.

Example: Working with Period Objects

Let's create a Period object and perform operations with it. Periods in pandas can represent various types of durations (days, months, years, etc.).

Code 2.37

```
import pandas as pd

# Create a Period object for a specific month
p = pd.Period('2024-11', freq='M')
print("Period Object:", p)

# Add 2 months to the Period
p_plus = p + 2
print("Period after Adding 2 Months:", p_plus)
```

```
Period Object: 2024-11
Period after Adding 2 Months: 2024-01
```

Explanation: - The pd.Period() constructor creates a period object, with the frequency (e.g., monthly, yearly) specified using the freq argument. - The resulting period represents the month of November 2024. Adding 2 months to this period results in January 2024.

2.8.4 Working with Timedelta in pandas

pandas also provides a Timedelta class that is similar to datetime.timedelta but is designed to handle more complex operations on time-based data, such as handling larger periods and differences.

Code 2.38

```
# Create a Timedelta object
timedelta_obj = pd.Timedelta(days=5, hours=3)
print("Timedelta Object:", timedelta_obj)

# Add the Timedelta to a Period object
new_period = p + timedelta_obj
print("New Period after Adding Timedelta:", new_period)
```

```
Timedelta Object: 5 days 03:00:00
New Period after Adding Timedelta: 2024-11-06 03:00:00
```

Explanation: - The pd.Timedelta() constructor creates a Timedelta object that represents a time duration of 5 days and 3 hours. - Adding this Timedelta object to a Period object results in a new period with the corresponding time adjustment.

2.8.5 Handling Recurring Periods

Many applications involve recurring periods, such as daily, weekly, or monthly cycles. pandas provides the pd.date_range() function to generate a sequence of dates with a specified frequency, which is useful for working with time series data.

Code 2.39

```python
# Create a range of dates with a daily frequency
date_range = pd.date_range('2024-11-01', periods=5, freq='D')
print("Date Range with Daily Frequency:", date_range)

# Create a range of dates with a monthly frequency
month_range = pd.date_range('2024-11-01', periods=3, freq='M')
print("Date Range with Monthly Frequency:", month_range)
```

```
Date Range with Daily Frequency: DatetimeIndex(['2024-11-01', '2024-11-02',
    '2024-11-03', '2024-11-04', '2024-11-05'], dtype='datetime64[ns]', freq='D')
Date Range with Monthly Frequency: DatetimeIndex(['2024-11-01', '2024-12-01',
    '2025-01-01'], dtype='datetime64[ns]', freq='M')
```

Explanation: - The `pd.date_range()` function generates a range of dates with a specified frequency. In the first case, we created a range of 5 daily dates starting from November 1, 2024. - In the second case, we created a range of 3 dates with a monthly frequency starting from November 1, 2024.

2.9 Advanced Time Manipulation

In many real-world applications, especially in fields such as finance, astronomy, and climate science, time manipulation can become complex due to varying time intervals, leap years, time zone differences, and irregular time series. Python's `datetime` module, along with third-party libraries like `pandas` and `dateutil`, allows for advanced time-based operations. This chapter will explore:

- Handling leap years and irregular time intervals.
- Working with time zones in detail.
- Performing advanced time arithmetic, such as adding business days.
- Working with irregular intervals (e.g., fiscal years, calendar months).
- Time series manipulation in data analysis.

By the end of this chapter, you will have mastered advanced techniques for working with dates and times in Python, allowing you to manipulate and analyze time-based data in complex scenarios.

2.9.1 Handling Leap Years and Irregular Time Intervals

Leap years, which occur every four years, add an extra day (February 29th) to the calendar. This makes the length of a year 366 days instead of the usual 365. To handle this and work with irregular time intervals, we need to consider the specific rules of the calendar.

Example: Working with Leap Year

Let's calculate the number of days between two dates, accounting for a leap year.

Code 2.40

```python
# Create two datetime objects, one in a leap year and the other in a non-leap year
dt_leap = datetime.date(2024, 2, 28)  # Leap year
dt_non_leap = datetime.date(2023, 2, 28)  # Non-leap year

# Add 1 day to the leap year date (should be February 29)
next_day_leap = dt_leap + datetime.timedelta(days=1)
print("Next Day after Feb 28, 2024 (Leap Year):", next_day_leap)

```

```
# Add 1 day to the non-leap year date (should be March 1)
next_day_non_leap = dt_non_leap + datetime.timedelta(days=1)
print("Next Day after Feb 28, 2023 (Non-Leap Year):", next_day_non_leap)
```

```
Next Day after Feb 28, 2024 (Leap Year): 2024-02-29
Next Day after Feb 28, 2023 (Non-Leap Year): 2023-03-01
```

Explanation: - In the leap year (2024), February 29th is added as the next day after February 28th. - In the non-leap year (2023), the next day after February 28th is March 1st. - By using `timedelta`, Python automatically accounts for leap years when performing date arithmetic.

2.9.2 Working with Time Zones in Detail

Time zone manipulation can be complex, especially when working with different geographic regions or Daylight Saving Time (DST). The Python `pytz` library provides advanced time zone handling capabilities. You can localize 'datetime' objects to specific time zones and convert them between time zones as needed.

Example: Converting Between Multiple Time Zones

We will now see how to convert a `datetime` object from one time zone to another and also handle DST.

Code 2.41

```
import pytz

# Create a naive datetime object (without time zone)
dt_naive = datetime.datetime(2024, 11, 17, 15, 30)

# Localize to US Eastern Time
eastern = pytz.timezone('US/Eastern')
dt_eastern = eastern.localize(dt_naive)

# Convert to UTC
dt_utc = dt_eastern.astimezone(pytz.utc)

# Convert to Tokyo time
tokyo = pytz.timezone('Asia/Tokyo')
dt_tokyo = dt_eastern.astimezone(tokyo)

print("Eastern Time:", dt_eastern)
print("Converted to UTC:", dt_utc)
print("Converted to Tokyo Time:", dt_tokyo)
```

```
Eastern Time: 2024-11-17 15:30:00-05:00
Converted to UTC: 2024-11-17 20:30:00+00:00
Converted to Tokyo Time: 2024-11-18 05:30:00+09:00
```

Explanation: - We localize a naive 'datetime' object (which has no time zone information) to US Eastern Time. - We then convert it to UTC and Tokyo time using the `astimezone()` method. Note how the time adjusts depending on the time zone.

2.9.3 Performing Advanced Time Arithmetic

Python provides a range of options for performing more advanced time-based arithmetic. For instance, we can add or subtract specific business days, which are typically weekdays excluding weekends (and sometimes holidays).

Example: Adding Business Days

The pandas library provides a useful function called BDay, which represents a business day. We can use it to add or subtract business days from a ‘datetime‘ object.

Code 2.42

```
import pandas as pd

# Create a datetime object for November 17, 2024 (a Sunday)
dt_weekend = datetime.datetime(2024, 11, 17)

# Add 3 business days to the datetime
business_day = pd.tseries.offsets.BDay(3)
new_dt_business_day = dt_weekend + business_day
print("Datetime after Adding 3 Business Days:", new_dt_business_day)
```

```
Datetime after Adding 3 Business Days: 2024-11-20 00:00:00
```

Explanation: - The BDay() function adds business days, skipping weekends (and holidays, if specified). - In this case, adding 3 business days to November 17, 2024 (a Sunday), results in November 20, 2024 (a Wednesday).

2.9.4 Working with Irregular Time Intervals

Handling irregular time intervals, such as those based on fiscal years, school terms, or custom schedules, requires manual adjustments. Some intervals are not based on fixed durations (e.g., a fiscal year may start in a particular month, and its length may vary).

Example: Custom Time Interval (Fiscal Year)

Let's calculate the start and end dates of a fiscal year that starts on April 1st and lasts 12 months.

Code 2.43

```
# Create a datetime object for April 1, 2024 (Fiscal Year Start)
fy_start = datetime.date(2024, 4, 1)

# Calculate the end date of the fiscal year (12 months later)
fy_end = fy_start + datetime.timedelta(days=365)  # Assume no leap year
print("Fiscal Year Start:", fy_start)
print("Fiscal Year End:", fy_end)
```

```
Fiscal Year Start: 2024-04-01
Fiscal Year End: 2025-03-31
```

Explanation: - This example shows how to manually calculate the start and end dates of a fiscal year by adding 365 days to the start date. This approach assumes no leap year.

2.9.5 Time Series Manipulation in Data Analysis

Time series data is a key area where advanced time manipulation is often required. Time series data typically involves tracking data points over a period of time, such as daily stock prices, temperature readings, or sales data. Python's pandas library makes time series manipulation easier with features like resampling, rolling windows, and shifting.

Example: Resampling Time Series Data

Let's resample a time series of daily data to weekly data.

Code 2.44

```python
# Create a time series with daily data
date_range = pd.date_range('2024-01-01', periods=7, freq='D')
data = pd.Series([10, 20, 30, 40, 50, 60, 70], index=date_range)

# Resample the data to weekly frequency, using the sum for each week
weekly_data = data.resample('W').sum()
print("Weekly Resampled Data:", weekly_data)
```

```
Weekly Resampled Data:
2024-01-07    210
2024-01-14    180
Freq: W-SUN, dtype: int64
```

Explanation: - We created a time series with daily frequency and resampled it to a weekly frequency using the `resample()` method. - The `sum()` function aggregates the data within each week.