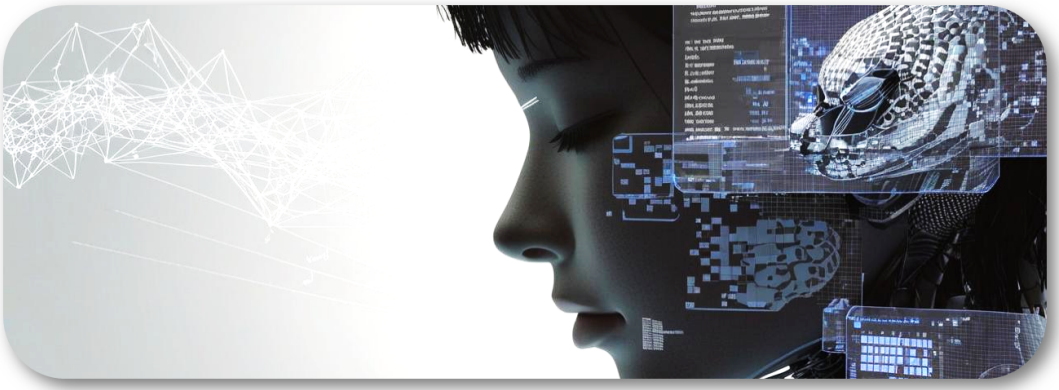


COMP 1023 Introduction to Python Programming

Tutorial 10: NumPy & Matplotlib

COMP 1023 Teaching Team

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology, Hong Kong SAR, China

Part I

NumPy

Prelude

NumPy is probably one of the most difficult topics in this course.
Take a deep breath, and try to understand how it works!

It consists of two main aspects:

1. The core concepts of NumPy, such as arrays, axes, indexing, and broadcasting.
2. The numerous functions / operations of NumPy.

In the course, the only thing we can teach is the core concepts. As there are numerous functions / operations, we would not be able to cover all of them.

Array Creation and Shape

```
import numpy as np
# Convert a Python list (iterable) to a NumPy array
a = np.array([0, 1, 2, 3, 4, 5])
# Reshape the array to 2 rows and 3 columns
print(a.reshape(2, 3))
# Reshape the array to 2 rows and 3 columns, then
# reshape the array to 1D
print(a.reshape(2, 3).reshape(-1))
# Generate 5 evenly spaced numbers between 0 and 1
print(np.linspace(0, 1, 5))
# Generate numbers from 0 until 42 with a step of 10
# Analogous to Python range(0, 42, 10)
print(np.arange(0, 42, 10))
```

Array Creation and Shape - Output

```
# a.reshape(2, 3)
[[0 1 2]
 [3 4 5]]

# a.reshape(2, 3).reshape(-1)
[0 1 2 3 4 5]

# np.linspace(0, 1, 5)
[0.   0.25 0.5  0.75 1.  ]

# np.arange(0, 42, 10)
[ 0 10 20 30 40]
```

Basic Operations

```
import numpy as np

a = np.array([0, 1, 2])
b = np.array([3, 4, 5])

# Element-wise addition
print(a + b)
# Element-wise multiplication
print(a * b)

# Similarly...
print(a - b)
print(a / b)
print(a % b)
print(a ** b)
print(a // b)
```

Basic Operations - Output

```
# a + b
[3 5 7]

# a * b
[ 0  4 10]
```

```
# a - b
[-3 -3 -3]
# a / b
[0.   0.25 0.4 ]
# a % b
[0 1 2]
# a ** b
[ 0  1 32]
# a // b
[0 0 0]
```

Array Indexing and Slicing

```
import numpy as np

a = np.arange(12).reshape(3, 4)
print(a)
# The element at row 0, column 0
print(a[0, 0])
# The elements at all rows, column 0
print(a[:, 0])
# The elements at row 0, all columns
print(a[0, :])
# The elements at all rows, all columns
print(a[:, :])
# The elements from row 1 to end, column 2 to end
print(a[1:, 2:])
```

Array Indexing and Slicing - Output

```
# a
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
# a[0, 0]
0
# a[:, 0]
[0 4 8]
# a[0, :]
[0 1 2 3]
```

```
# a[:, :]
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

# a[1:, 2:]
[[ 6  7]
 [10 11]]
```

Multidimensional array indexing

In Python, we use a pair of square brackets for indexing each axis of multidimensional lists, however, in NumPy arrays, we use tuples of slice objects instead.

Boolean Array Indexing

```
import numpy as np

arr = np.array([+1, -2, +3, -4, +5, -6, +7, -8])
a = arr > 0         # positive numbers becomes True
b = arr % 2 == 0    # even numbers becomes True

print(a)
print(arr[a])       # positive

print(b)
print(arr[b])       # even

print(arr[a & b])   # positive AND even
print(arr[a | b])   # positive OR  even
```

Boolean Array Indexing - Output

```
# a
[ True False  True False  True False  True False]
# arr[a]      # positive
[1 3 5 7]
# b
[False  True False  True False  True False  True]
# arr[b]      # even
[-2 -4 -6 -8]
# arr[a & b]  # positive AND even
[]
# arr[a | b]  # positive OR  even
[ 1 -2  3 -4  5 -6  7 -8]
```

Integer Array Indexing

```
import numpy as np

arr = np.arange(9).reshape(3, 3)
print(arr)

rows = np.array([0, 1, 2])
cols = np.array([2, 1, 0])

# Elements of rows 0, 1, 2; columns 2, 1, 0
# i.e., at (0, 2), (1, 1), (2, 0).
print(arr[rows, cols])
```

Integer Array Indexing - Output

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

# arr[rows, cols]
[2 4 6]
```

Dot / Matrix Product

```
import numpy as np

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

print(f"a:\n{a}\nb:\n{b}\na @ b:\n{a @ b}")

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

print(f"a:\n{a}\nb:\n{b}\na @ b:\n{a @ b}")

a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])

print(f"a:\n{a}\nb:\n{b}\na @ b:\n{a @ b}")
```

Dot / Matrix Product - Output

```
a:
[1 2 3]
b:
[1 2 1]
a @ b:
8
```

```
a:
[[1 2]
 [3 4]]
b:
[[1]
 [3]]
a @ b:
[[ 7]
 [15]]
```

```
a:
[[1 2]
 [3 4]]
b
[[5 6]
 [7 8]]
a @ b:
[[19 22]
 [43 50]]
```

newaxis and expand_dims

```
import numpy as np

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

print(arr[:, np.newaxis])
print(arr[np.newaxis, :])

print(np.expand_dims(arr, axis=0))
print(np.expand_dims(arr, axis=1))

print(arr.reshape(1, -1))
print(arr.reshape(-1, 1))

print(arr[np.newaxis, :, np.newaxis])
```

newaxis and expand_dims - Output

```
# arr[:, np.newaxis]
[[1]
 [2]
 [3]
 [4]]
# arr[np.newaxis, :]
[[1 2 3 4]]

# np.expand_dims(arr, axis=0)
[[1 2 3 4]]
# np.expand_dims(arr, axis=1)
[[1]
 [2]
 [3]
 [4]]
```

```
# arr.reshape(1, -1)
[[1 2 3 4]]
# arr.reshape(-1, 1)
[[1]
 [2]
 [3]
 [4]]

# arr[np.newaxis, :, np.newaxis]
[[[1]
  [2]
  [3]
  [4]]]
```

Aggregation

```
import numpy as np

arr = np.arange(0, 9).reshape(3, 3)
print(arr)

print(np.sum(arr))
print(np.sum(arr, axis=0))
print(np.sum(arr, axis=1))

print(np.max(arr, axis=1))

print(np.mean(arr, axis=0))
```

Aggregation - Output

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

# np.sum(arr)
36
# np.sum(arr, axis=0)
[ 9 12 15]
# np.sum(arr, axis=1)
[ 3 12 21]
```

```
# np.max(arr, axis=1)
[2 5 8]

# np.mean(arr, axis=0)
[3. 4. 5.]
```

iPRS Question 1

A NumPy array `gps` contains the grades points that a student got in 3 courses. The NumPy array `cs` contains the credits of the 3 courses. Which of the following code snippet, A, B, C, or D, can be used to compute the CGA of the student?

$$\text{CGA} = \frac{\sum(\text{grade-point} \times \text{credit})}{\sum \text{credit}}$$

A. `(gps @ cs) / np.sum(cs)`

B. `(gps * cs) / np.sum(cs)`

C. `(gps + cs) / np.sum(cs)`

D. `(gps ** cs) / np.sum(cs)`

iPRS Question 1 - Answer

Explanation

A. `(gps @ cs) / np.sum(cs)`

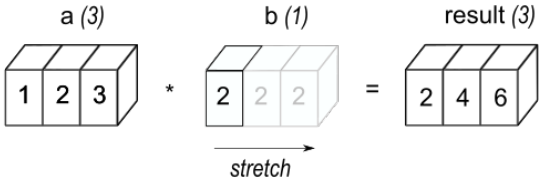
If you see a formula that involves summation of products, it is often a sign that you can use the dot product (i.e., `@` operator) to compute it efficiently.

Broadcasting

```
import numpy as np

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

a = np.array([[ 0,  0,  0], [10, 10, 10], [20, 20, 20], [30, 30, 30]])
b = np.array([1, 2, 3])
print(a + b)

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

a = np.array([0, 10, 20, 30])
b = np.array([1, 2, 3])
print(a[:, np.newaxis] + b)
```

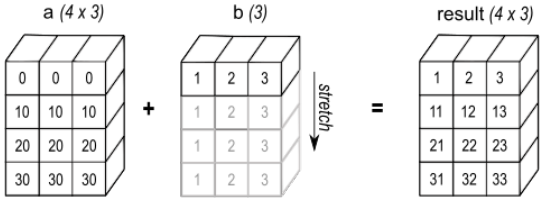
Broadcasting - Output

```
# a = np.array([1, 2, 3])
# a * 2
[2 4 6]
```



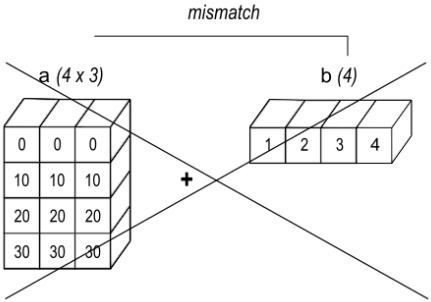
Broadcasting - Output

```
# a = np.array([[ 0,  0,  0], [10, 10, 10], [20, 20, 20], [30, 30, 30]])
# b = np.array([1, 2, 3])
# a + b
[[ 1  2  3]
 [11 12 13]
 [21 22 23]
 [31 32 33]]
```



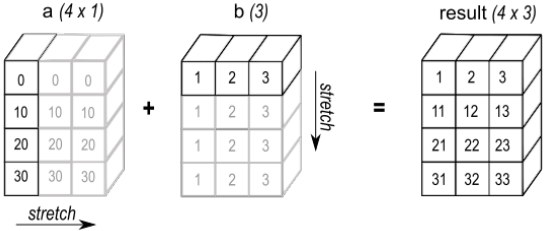
Broadcasting - Output

```
# a = np.array([[ 0,  0,  0], [10, 10, 10], [20, 20, 20], [30, 30, 30]])
# b = np.array([1, 2, 3, 4])
# a + b
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (4,3) (4,)
```



Broadcasting - Output

```
# a = np.array([0, 10, 20, 30])
# b = np.array([1, 2, 3])
# a[:, np.newaxis] + b
[[ 1  2  3]
 [11 12 13]
 [21 22 23]
 [31 32 33]]
```



Broadcasting - Acknowledgement

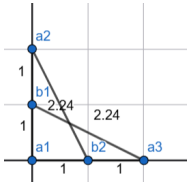
The examples and figures are from:
`https://numpy.org/doc/stable/user/basics.broadcasting.html`.

It is highly recommended to read it. It contains more detailed explanations and examples.

iPRS Question 2

The arrays a and b contain 3 points, $(0,0), (2,0), (0,2)$ and 2 points $(1,0), (0,1)$ in 2D space. Which of the following code snippet, A, B, C, or D, can be inserted to the blank to compute the Euclidean distances between each pair of points in *a* and *b*?

```
a = np.array([[0, 0],
              [0, 2],
              [2, 0]])
b = np.array([[0, 1],
              [1, 0]])
np.sqrt(np.sum(np.square(np.abs(
    _________
)), axis=-1))
```



A.
```
a[np.newaxis, :, :] - \
  b[:, np.newaxis, :]
```

B.
```
a[:, np.newaxis, :] - \
  b[:, np.newaxis, :]
```

C.
```
a[np.newaxis, :, :] - \
  b[:, :, np.newaxis]
```

D.
```
a[:, np.newaxis, :] - \
  b[:, :, np.newaxis]
```

iPRS Question 2 - Answer

Exaplanation

A. `a[np.newaxis, :, :] - b[:, np.newaxis, :]`

The last dimension represents the 2D coordinates (x, y), so it should not be included in the broadcasting dimensions. Instead, we only broadcast the first two dimensions only, which are the number of points in `b` and `a` respectively.

		a1	*a2*	*a3*
		`[0, 0]`	`[0, 2]`	`[2, 0]`
b1	`[0, 1]`	1	1	2.24
b2	`[1, 0]`	1	2.24	1

Part II

Matplotlib

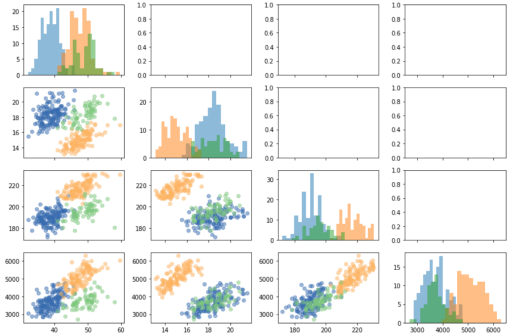
Introduction

- Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. It is widely used for data visualization across various fields, including data science, engineering, and research.
- Key Features:
  - Static Visualizations: Generate high-quality static plots, such as line graphs, bar charts, and scatter plots.
  - Animated Visualizations: Create dynamic visualizations that illustrate changes over time or other variables.
  - Interactive Visualizations: Build interactive plots that allow users to explore data through zooming, panning, and real-time updates.

In this course, we will use version 3.10.8, and we will focus on static visualizations.

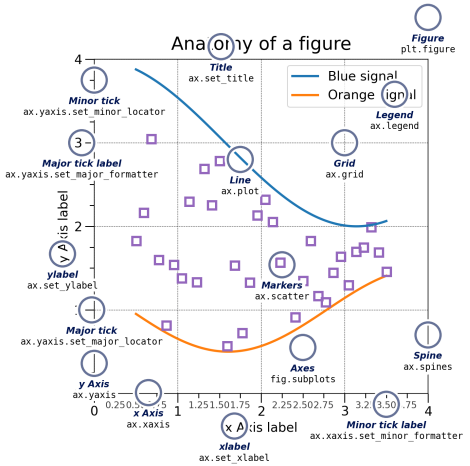
Why Matplotlib?

- Wide Adoption: Matplotlib is one of the most widely used libraries for data visualization in the Python ecosystem.
- Customization: It offers extensive options for customizing visual elements, including colors, labels, and styles.
- Integration: Matplotlib easily integrates with other libraries such as NumPy, Pandas, and Seaborn (a statistical data visualization library) for enhanced data analysis and visualization.



Key Components of a Matplotlib Figure

- A Matplotlib figure is composed of several key elements:
  - Figure: This serves as the primary container for all visual elements, functioning as the canvas for the entire plot.
  - Axes: These are the specific regions within the figure where data visualization occurs; a single figure can incorporate multiple axes.
  - Axis: The axes define the horizontal (x-axis) and vertical (y-axis) dimensions, including their limits, tick marks, and labels essential for data interpretation.
  - Lines and Markers: Lines are utilized to connect data points, illustrating trends, while markers highlight individual data points, particularly in scatter plots.
  - Title and Labels: The title of the plot provides overarching context, while axis labels clarify the data being represented on each respective axis.



Introduction to Matplotlib Pyplot

- Pyplot is a module of Matplotlib designed for creating static, interactive, and animated visualizations in Python.
- To effectively utilize Pyplot, follow these steps:
  1. Import the Module: Begin by importing the module using `import matplotlib.pyplot as plt`.
  2. Prepare Data: Organize your data into lists or arrays for plotting.
  3. Create the Plot: Generate the plot by calling `plt.plot()` with your data.
  4. Enhance the Visualization: Customize the plot by adding titles, axis labels, and other features using `plt.title()`, `plt.xlabel()`, and `plt.ylabel()`.
  5. Display the Plot: Finally, render the plot on the screen with `plt.show()`.

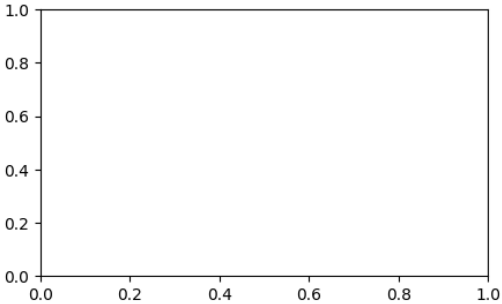
Figures and Axes

- The Figure object contains all the plots.
- You can have multiple plots inside the Figure object.
- You can also save the figure as an image (e.g., JPEG, PNG, etc.)
- You can have one or more Axes inside the figure object, and each Axes object corresponds to a plot.
- Each Axes has an x_axis and an y_axis that represent the data that you want to plot.

Creating a New Figure with a Plot

- To create a Figure, use the `subplots()` function from the Matplotlib library.
- The `subplots()` function returns a new Figure and its Axes object.

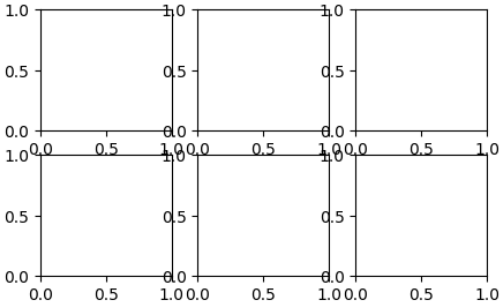
```python
# Filename: single_plot_create.py
import matplotlib.pyplot as plt

# The width and height are 5 and 3 inches
fig, ax = plt.subplots(figsize=(5, 3))
plt.show()
```



Creating a New Figure with Multiple Plots

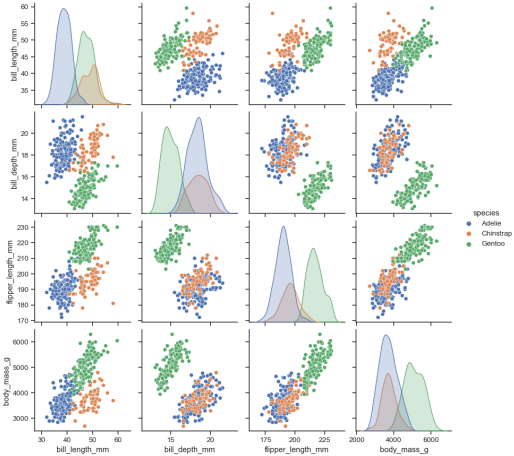
- To create multiple plots, specify the number of rows and columns in the parameters.
- The first two parameters of `plt.subplots()` define the number of plots as rows and columns. It returns the Figure object and the Axes as a NumPy array, allowing you to access each chart using NumPy indexing methods.

```python
# Filename: multiple_plots_create.py
import matplotlib.pyplot as plt

# Multiple plots with 2 rows and 3 columns
fig, ax = plt.subplots(2, 3, figsize=(5, 3))
plt.show()
```



Matplotlib – Part I

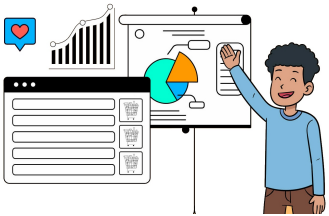
Pairwise Data Visualization



Pairwise Data Visualization

- Visualization techniques for pairwise data include plots of (x,y) coordinates, tabular data (var_0, ..., var_n), and functional relationships of the form $f(x) = y$.
  - `plot(x, y)`: Creates a line plot connecting the data points.
  - `scatter(x, y)`: Generates a scatter plot to display individual data points.
  - `bar(x, height)`: Produces a bar chart representing categorical data.
  - `stem(x, y)`†: Generates a stem plot for discrete data representation.
  - `fill_between(x, y1, y2)`†: Fills the area between two curves.
  - `stackplot(x, y)`†: Creates a stacked area plot to visualize cumulative data.
  - `stairs(values)`†: Generates a step plot for visualizing changes in data.

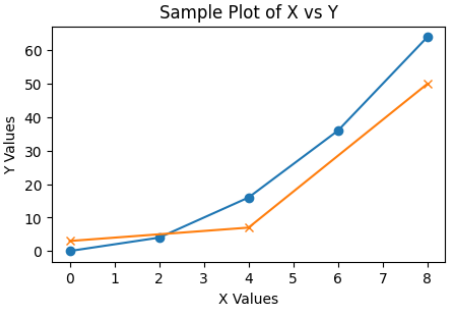
†: Self-explore



Drawing a Line Curve

- To draw a line curve, use the `plot()` method of the Axes object (`ax.plot`). This method takes two lists or NumPy arrays as input: the first list contains the x-coordinates, and the second contains the y-coordinates of the points in the line.
- You can customize the appearance of the plot by specifying the marker type using the marker attribute, and by adding a legend for the data points with the label attribute.
- You can specify the labels (i.e., names) of the x and y axes using `set_xlabel()` and `set_ylabel()`, respectively, and add a title to the plot, use the `set_title()` method.

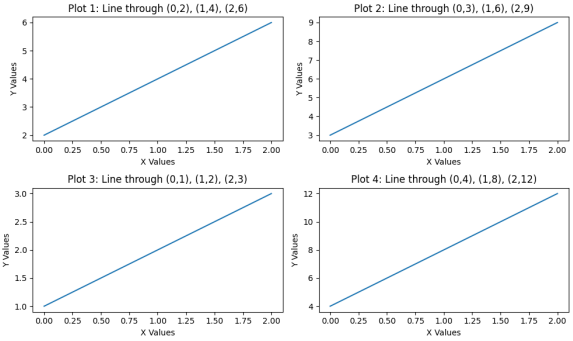
```
import matplotlib.pyplot as plt  # Filename: line_curve_draw.py

fig, ax = plt.subplots(figsize=(5, 3))
# Line through (x,y) -> (0,0), (2,4), (4,16), (6,36), (8, 64)
ax.plot([0, 2, 4, 6, 8], [0, 4, 16, 36, 64],
        marker='o', label="Data Points 1")
# Another line through (x,y) -> (0,3), (4,7), (8,50)
ax.plot([0, 4, 8], [3, 7, 50],
        marker='x', label="Data Points 2")

ax.set_xlabel("X Values")
ax.set_ylabel("Y Values")
ax.set_title("Sample Plot of X vs Y")
plt.show()
```



Drawing a Line Plot in Multiple Rows and Multiple Columns

- Each subplot can represent different datasets.
- x and y axis labels can be added using `set_xlabel()` and `set_ylabel()`.
- Titles for each subplot are set using the `set_title()` method.

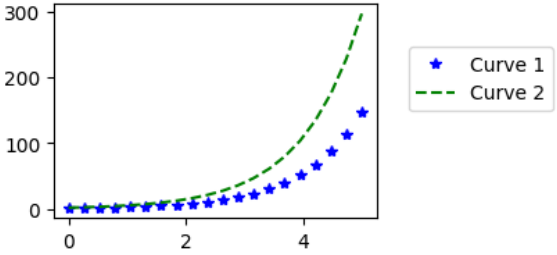
```
import matplotlib.pyplot as plt  # Filename: line_plot_draw_multiples.py

fig, ax = plt.subplots(2, 2, figsize=(10, 6))

ax[0, 0].plot([0, 1, 2], [2, 4, 6])  # First subplot
ax[0, 0].set_xlabel("X Values")
ax[0, 0].set_ylabel("Y Values")
ax[0, 0].set_title("Plot 1: Line through (0,2), (1,4), (2,6)")

ax[0, 1].plot([0, 1, 2], [3, 6, 9])  # Second subplot
ax[0, 1].set_xlabel("X Values")
ax[0, 1].set_ylabel("Y Values")
ax[0, 1].set_title("Plot 2: Line through (0,3), (1,6), (2,9)")

ax[1, 0].plot([0, 1, 2], [1, 2, 3])  # Third subplot
ax[1, 0].set_xlabel("X Values")
ax[1, 0].set_ylabel("Y Values")
ax[1, 0].set_title("Plot 3: Line through (0,1), (1,2), (2,3)")

ax[1, 1].plot([0, 1, 2], [4, 8, 12]) # Fourth subplot
ax[1, 1].set_xlabel("X Values")
ax[1, 1].set_ylabel("Y Values")
ax[1, 1].set_title("Plot 4: Line through (0,4), (1,8), (2,12)")

plt.tight_layout()
plt.show()
```



Drawing a Sequence of Points/Segments

- The `plt.plot()` function also allows you to display a sequence of points or segments that share the same properties (e.g., size, color, width).

```
# Filename: sequence_points_segments_draw.py
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 5, 20)
y = np.exp(x)

fig, ax = plt.subplots(figsize=(3, 2))

ax.plot(x, y, c="blue", linestyle="",
        marker='*', label="Curve 1")
ax.plot(x, 2*y, c="green",
        linestyle="--", label="Curve 2")

# Specify the legend position with relative position
ax.legend(loc=(1.1, 0.5))
plt.show()
```

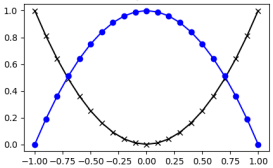


iPRS Question 3

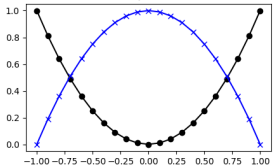
Which figure will the following program produce?

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(5,3))
X = [x / 100 for x in range(-100, 101, 10)]
Y = [x ** 2 for x in X]
ax.plot(X, Y, marker='o', color="black")
X = [x / 100 for x in range(-100, 101, 10)]
Y = [-(x ** 2) + 1 for x in X]
ax.plot(X, Y, marker='x', color="blue")
ax.set_title("iPRS Question 3")
plt.show()
```

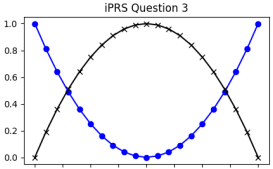
A



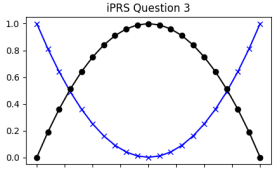
B



C



D

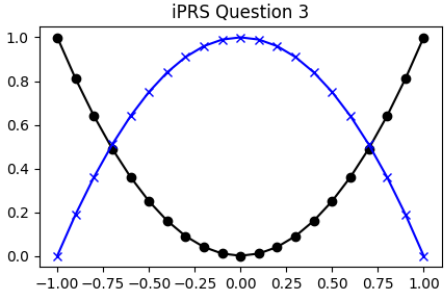


iPRS Question 3 - Answer

Explanation

B.

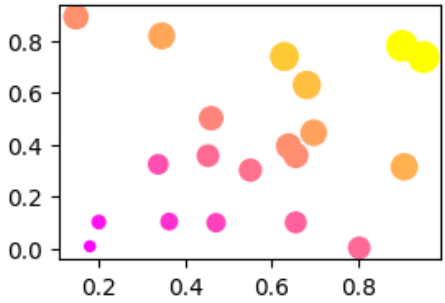
1. The first curve is `y = x**2`, plotted with black circle markers (`'o'`).
2. The second curve is `y = -x**2 + 1`, plotted with blue x markers (`'x'`).
3. So the correct figure must have a black upward parabola and a blue downward parabola.



Scatter Plot: Displaying a Set of Points With Colormap

- To show a set of points and assign them custom properties (e.g., color, size), use the `ax.scatter()` method.
- You need to specify the lists of x and y coordinates of all your points as parameters (as NumPy arrays).
- You can also specify a colormap using the `cmap` parameter, and the size of the points using the `s` parameter. The size is expressed as the area in square points (dpi).

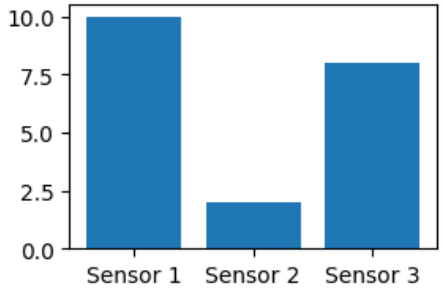
```
import numpy as np  # Filename: scatterplot.py
import matplotlib.pyplot as plt

x = np.random.rand(20)
y = np.random.rand(20)

# Color as a function of the positions of the points
colors = x + y
# Size as a function of the positions of the points
area = 100 * (x + y)
fig, ax = plt.subplots(figsize=(3, 2))

# Specify the colormap as "spring"
ax.scatter(x, y, c=colors, cmap="spring", s=area)
plt.show()
```



Bar Chart: Displaying a Sequence of Numbers as Bars

- To plot vertical or horizontal bars, use the `ax.bar()` method. You can specify the position of each bar on the x-axis as a list, and the height of each bar as a corresponding list (both lists should have the same size).
- You can assign text labels to the ticks on the horizontal axis using the `tick_label` parameter.

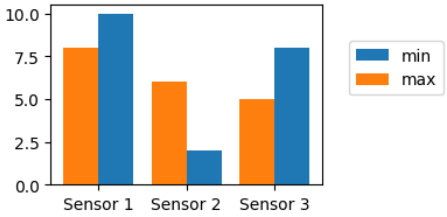
```python
# Filename: barchart_single.py
import numpy as np
import matplotlib.pyplot as plt

# Position of the bars on the x-axis
x = [1, 2, 3]
# The height of each bar
height = [10, 2, 8]

labels = ["Sensor 1", "Sensor 2", "Sensor 3"]
fig, ax = plt.subplots(figsize=(3, 2))
ax.bar(x, height, tick_label=labels)
plt.show()
```



Bar Chart: Displaying Multiples Bars

- You can group multiple bars side-by-side.
- Position the two bar plots at $x + \frac{\text{width}}{2}$ and $x - \frac{\text{width}}{2}$.

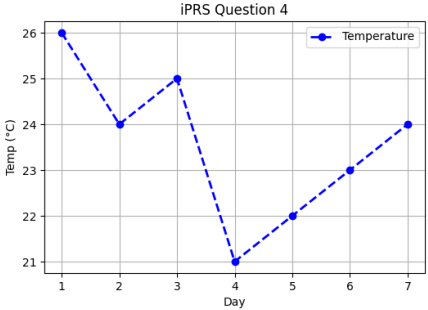
```python
# Filename: barchart_multiples.py
import numpy as np
import matplotlib.pyplot as plt
heightMin = [10, 2, 8]
heightMax = [8, 6, 5]
x = np.arange(3)
width = 0.4
labels = ["Sensor 1", "Sensor 2", "Sensor 3"]

fig, ax = plt.subplots(figsize=(3, 2))
# Blue bars
ax.bar(x + width / 2, heightMin, width=width, label="Min")
# Orange bars
ax.bar(x - width / 2, heightMax, width=width, label="Max")

ax.set_xticks(x)
ax.set_xticklabels(labels)
ax.legend(loc=(1.1, 0.5))
plt.show()
```



iPRS Question 4

Which option should replace the blank?

```
import matplotlib.pyplot as plt

days = [1, 2, 3, 4, 5]
temp = [22, 24, 21, 25, 23]

plt.figure(figsize=(6, 4))
plt.plot(days, temp, ________)
plt.title("iPRS Question 4")
plt.xlabel("Day")
plt.ylabel("Temp (°C)")
plt.legend()
plt.grid(True)
plt.show()
```



A. `color='blue', marker='o', linestyle='--', linewidth=2, label='Temperature'`

B. `color='red', marker='x', linestyle='-', linewidth=1, label='Temperature'`

C. `c='blue', marker='o', linestyle=':', width=2, label='Temp'`

D. `color='blue', marker='o', style='--', linewidth=2, legend='Temperature'`

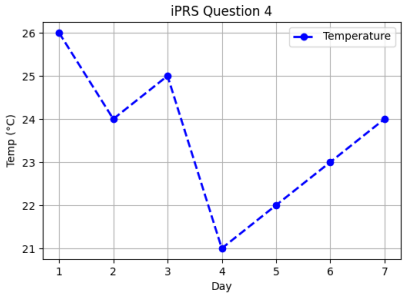
iPRS Question 4 - Answer

Explanation

A.

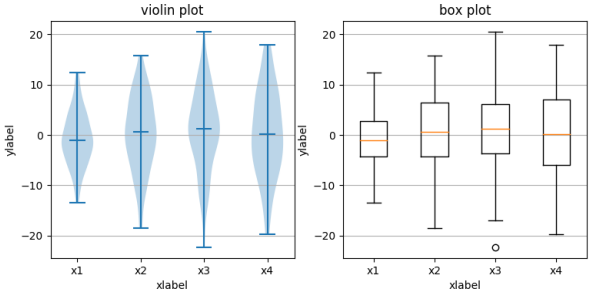
The target plot is a blue dashed line with circle markers, thicker line width, and legend text `Temperature`. Only option A matches all of these settings exactly.

Options C and D use invalid keyword arguments for `plt.plot()` (`width`, `style`, `legend`), and option B has a different color/marker/linestyle.



Matplotlib – Part II

Statistical Distributions



Statistical Distributions

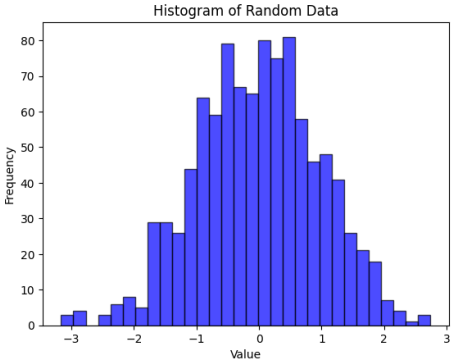
- Visualization techniques for depicting the distribution of one or more variables within a dataset.
- Many of these methods also provide calculations of the distributions.
  - `hist(x)`: Creates a histogram to visualize the frequency distribution of a single variable.
  - `boxplot(X)`: Generates a box plot to summarize the distribution of data through their quartiles.
  - `errorbar(x, y, yerr, xerr)`†: Displays data points with error bars indicating variability.
  - `violinplot(D)` †: Combines a box plot and a density plot to show the distribution of data.
  - `eventplot(D)`†: Visualizes events along an axis, useful for time series data.
  - `hist2d(x, y)`†: Creates a 2D histogram to display the joint distribution of two variables.
  - `hexbin(x, y, C)`†: Generates a hexagonal bin plot for bivariate data visualization.
  - `pie(x)`†: Creates a pie chart to represent proportions of categorical data.
  - `ecdf(x)`†: Computes and plots the empirical cumulative distribution function.

†: Self-explore

Creating a Histogram

- The `hist(x)` function is used to create a histogram, which visualizes the distribution of a dataset. `x` is a one-dimensional array or list containing numerical data.
- You can customize the number of bins to adjust the granularity of the histogram using the `bins` parameter.
- Additional options such as `color`, `alpha`, and `edgecolor` can enhance the visual appeal of the histogram.

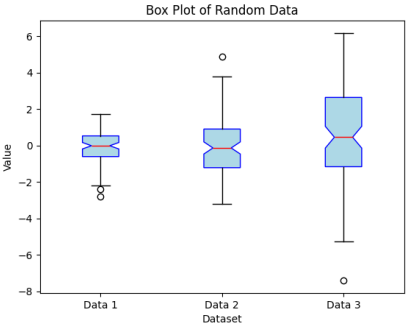
```python
# Filename: histogram.py
import matplotlib.pyplot as plt
import numpy as np

data = np.random.randn(1000) # Generate random data

plt.hist(data, bins=30, color='blue',
         alpha=0.7, edgecolor='black')
plt.title("Histogram of Random Data")
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.show()
```



Creating a Box Plot

- The `boxplot(X)` function creates a box plot, which visualizes the distribution of a dataset through its quartiles. `X` is a one-dimensional array or a two-dimensional array (for multiple box plots).
- Box plots provide a summary of the central tendency, variability, and potential outliers in the data.
- You can customize the appearance of the box plot using parameters such as `color`, `notch`, and `vert` (to control orientation).

```
import matplotlib.pyplot as plt  # Filename: boxplot.py
import numpy as np

data = [np.random.normal(0, std, 100) for std in range(1, 4)]

plt.boxplot(data, notch=True, patch_artist=True,
            boxprops=dict(facecolor='lightblue',
            color='blue'), medianprops=dict(color='red'))
plt.title("Box Plot of Random Data")
plt.xlabel("Dataset"); plt.ylabel("Value")
plt.xticks([1, 2, 3], ['Data 1', 'Data 2', 'Data 3'])
plt.show()
```



Save a Plot to File

- Generated figures can be saved to files in various formats (e.g., JPEG, PNG, PDF, EPS, etc.).
- Use the `fig.savefig()` method to save the figure.

```python
# Filename: save_plot.py
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(3, 2))

ax.plot([0, 1, 2], [2, 4, 6])
ax.plot([0, 1, 2], [3, 6, 9])

fig.savefig("test.png") # or .jpg, .eps, .pdf
```

A Lot More to Explore

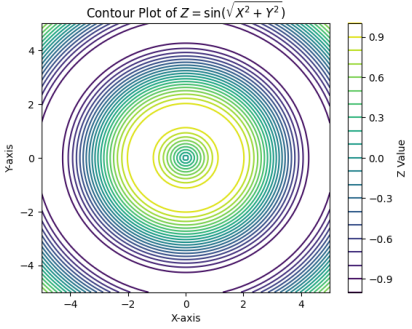
- Gridded Data
  - `imshow(Z)`†
  - `pcolormesh(X, Y, Z)`†
  - `contour(X, Y, Z)`
  - `contourf(X, Y, Z)`†
  - `barbs(X, Y, U, V)`†
  - `quiver(X, Y, U, V)`†
  - `streamplot(X, Y, U, V)`†
- Irregularly Gridded Data
  - `tricontour(x, y, z)`
  - `tricontourf(x, y, z)`†
  - `tripcolor(x, y, z)`†
  - `triplot(x, y)`†
- 3D and Volumetric Data
  - `bar3d(x, y, z, dx, dy, dz)`†
  - `fill_between(x1, y1, z1, x2, y2, z2)`
  - `plot(xs, ys, zs)`†
  - `quiver(X, Y, Z, U, V, W)`†
  - `scatter(xs, ys, zs)`†
  - `streamplot(x, y, u, v)`†
  - `plot_surface(X, Y, Z)`†
  - `plot_trisurf(x, y, z)`†
  - `voxels([x, y, z], filled)`†
  - `plot_wireframe(X, Y, Z)`†

†: Self-explore

Creating Contour Plots with `contour()`

- The `contour()` function is used to create contour plots, which represent 3D data in two dimensions using contour lines. A grid of data points represented by 2D arrays for the x and y coordinates and a corresponding z value.
- You can customize the contour levels using the `levels` parameter to specify the number of contour lines or specific z-values. Additional parameters such as `cmap` allow you to choose color maps for better visualization.

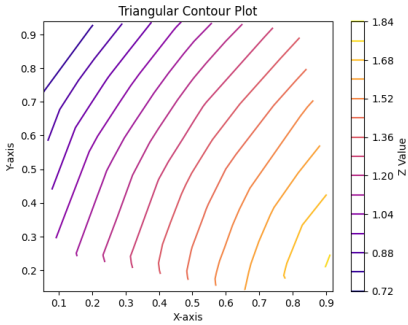
```python
# Filename: contour.py
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-5, 5, 100); y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2))

plt.contour(X, Y, Z, levels=20, cmap='viridis')
plt.colorbar(label='Z Value')
plt.title("Contour Plot of $Z = \sin(\sqrt{X^2 + Y^2})$")
plt.xlabel("X-axis"); plt.ylabel("Y-axis")
plt.show()
```



Creating Triangular Contour Plots with `tricontour()`

- The `tricontour()` function is used to create contour plots for irregularly spaced data defined by triangles. Three 1D arrays representing the x and y coordinates of the points, and a corresponding z value for each point.
- You can specify the contour levels using the `levels` parameter to control the number of contour lines or specific z-values. The `triangulate` parameter allows you to define the triangulation of the data for better visualization.

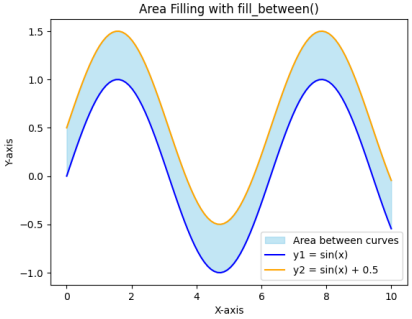
```python
# Filename: tricontour.py
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.tri import Triangulation

x = np.random.rand(30); y = np.random.rand(30)
z = np.sin(x) + np.cos(y)
triang = Triangulation(x, y) # Create triangulation

plt.tricontour(triang, z, levels=14, cmap='plasma')
plt.colorbar(label='Z Value')
plt.title("Triangular Contour Plot")
plt.xlabel("X-axis"); plt.ylabel("Y-axis")
plt.show()
```



Using `fill_between()` for Area Filling

- The `fill_between()` function is used to fill the area between two horizontal curves, useful for highlighting regions in a plot. x-coordinates and two sets of y-coordinates are defined the boundaries of the filled area.
- You can customize the appearance of the filled area using parameters like `color`, `alpha` (transparency), and `label` for legends.

```
# Filename: area_filling.py
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 10, 100)
y1 = np.sin(x); y2 = np.sin(x) + 0.5

plt.fill_between(x, y1, y2, color='skyblue',
                 alpha=0.5, label='Area between curves')
plt.plot(x, y1, label='y1 = sin(x)', color='blue')
plt.plot(x, y2, label='y2 = sin(x) + 0.5', color='orange')
plt.title("Area Filling with fill_between()")
plt.xlabel("X-axis"); plt.ylabel("Y-axis")
plt.legend()
plt.show()
```



Key Terms

- Animated Visualizations
- Area Filling
- Axes
- Axis
- Bar Chart
- Box Plot
- Color Map
- Contour Plot
- Data Visualization
- Error Bars
- Figure
- fill_between
- Gridded Data
- Histogram
- Interactive Visualizations
- Irregularly Gridded Data
- Matplotlib
- Plot
- Pyplot
- Scatter Plot
- Static Visualizations
- Triangular Contour Plot

Review Questions

Fill in the blanks in each of the following sentences about the Python environment.

1. Matplotlib is a comprehensive library for creating _____, animated, and interactive visualizations in Python.
2. The _____ module is designed for creating static, interactive, and animated visualizations.
3. A _____ serves as the primary container for all visual elements in a plot.
4. The _____ defines the horizontal (x-axis) and vertical (y-axis) dimensions of a plot.
5. The _____ function is used to create a histogram, which visualizes the distribution of a dataset.
6. To create a _____ plot, you can use the `boxplot()` function.

Answer: 1. static plots, 2. Pyplot, 3. figure, 4. axis, 5. hist(), 6. box.

Review Questions

Fill in the blanks in each of the following sentences about the Python environment.

7. The `scatter()` method is used to show a set of _____ on a plot.
8. You can fill the area between two curves using the _____ function.
9. The `contour()` function is used to create _____ plots that represent 3D data in two dimensions.
10. To create multiple plots in one figure, use the `subplots()` function with specified _____ and _____.
11. The _____ parameter in the `plot()` method allows you to customize the appearance of the plot.
12. You can save a figure to a file using the `fig.savefig()` method, which allows saving in various _____.

Answer: 7. points, 8. fill_between, 9. contour, 10. rows; columns, 11. marker, 12. formats.

That's all!

Any question?