**PYTHON ADVANCE ASSIGNMENT\_23**

**Q1. If you have any, what are your choices for increasing the comparison between different figures on the same graph?**

There are several techniques you can use to increase the comparison between different figures on the same graph in Python. Here are a few options:

Use a consistent scale and baseline: You can use the ylim() method to set the same scale and baseline for the y-axis across all figures. For example, plt.ylim(0, 10) will set the y-axis to start at zero and end at ten.

Use color or shading: You can use the color or alpha parameters to make the lines or bars for each figure more distinguishable. For example, plt.plot(x, y, color='red') will plot the data in red.

Use labels and annotations: You can use the xlabel(), ylabel(), title(), and annotate() methods to label the graph and highlight key points. For example, plt.annotate('Max value', xy=(3, 7), xytext=(4, 8), arrowprops=dict(facecolor='black', shrink=0.05)) will add an arrow with a text box to the plot at the point (3, 7).

Use small multiples: You can use the subplots() method to create a grid of smaller plots, each showing a different figure. This can make it easier to compare the figures and avoid clutter. For example, fig, axs = plt.subplots(2, 2) will create a 2x2 grid of subplots.

Overall, there are many different ways to customize your graphs in Python to increase the comparison between different figures. The best approach will depend on the specific data and the story you want to tell with your visualization.

**Q2. Can you explain the benefit of compound interest over a higher rate of interest that does not compound after reading this chapter?**

Compound interest is interest that is calculated not only on the principal amount, but also on any interest that has been earned previously. The benefit of compound interest over a higher rate of interest that does not compound is that it allows the interest to build upon itself over time, resulting in greater overall returns.

For example, let's say you invest $1,000 for 10 years at an annual interest rate of 5%. If the interest is compounded annually, at the end of 10 years, you would have $1,628.89. However, if the interest is not compounded, you would only have $1,500 at the end of the 10-year period. This is because with compound interest, each year the interest earned is added to the principal, and then the interest for the next year is calculated on the new, higher balance.

In Python, you can calculate compound interest using the following formula:

A = P \* (1 + r/n) \*\* (n\*t)

where:

A is the final amount

P is the principal amount

r is the annual interest rate

n is the number of times the interest is compounded per year

t is the time period in years

Using this formula, you can easily see how a higher rate of interest that does not compound will result in lower returns compared to a lower rate of interest that is compounded. Compound interest is a powerful tool for growing your investments over time, and can lead to significant long-term gains.

**Q3. What is a histogram, exactly? Name a numpy method for creating such a graph.**

A histogram is a graphical representation of the distribution of a dataset. It consists of a set of contiguous, non-overlapping bars or rectangles, where the area of each rectangle is proportional to the frequency of the observations falling within a specified class interval. Histograms are commonly used to visualize the frequency distribution of continuous or discrete data.

In Python, the NumPy library provides a method for creating histograms called numpy.histogram(). This function takes an array of data and a specified number of bins, and returns a tuple of two arrays: the counts for each bin and the bin edges. Here's an example:

import numpy as np

import matplotlib.pyplot as plt

# Generate some random data

data = np.random.normal(size=1000)

# Create a histogram with 10 bins

counts, edges = np.histogram(data, bins=10)

# Plot the histogram

plt.bar(edges[:-1], counts, width=np.diff(edges), align='edge')

plt.show()

In this example, we use numpy.random.normal() to generate an array of 1000 normally distributed random numbers. We then use numpy.histogram() to compute the frequency counts for 10 equally spaced bins, and plot the histogram using matplotlib.pyplot.bar().

**Q4. If necessary, how do you change the aspect ratios between the X and Y axes?**

To change the aspect ratios between the X and Y axes, you can modify the scaling of the plot in a plotting software or programming language that you are using. Here are some steps to achieve this:

Determine the desired aspect ratio: Decide on the aspect ratio you want to use for your plot. An aspect ratio of 1:1 (square plot) means the X and Y axes have the same scaling.

Set the aspect ratio: In most plotting software or programming languages, you can set the aspect ratio by adjusting the scaling of the X and Y axes. For example, in matplotlib (a popular plotting library for Python), you can use the aspect parameter of the plt.imshow or plt.plot function to set the aspect ratio. For instance, plt.imshow(data, aspect='auto') will set the aspect ratio automatically according to the shape of data.

Adjust the scaling: If the aspect ratio is not automatically set, you can adjust the scaling of the X and Y axes to match the desired aspect ratio. For example, in matplotlib, you can use the set\_xlim and set\_ylim methods to set the range of the X and Y axes, respectively.

Test and adjust: Once you have set the aspect ratio, test your plot to see if it looks the way you want it to. If not, adjust the aspect ratio or scaling until you are satisfied with the result.

Remember that changing the aspect ratio can affect the interpretation of the data, so it's important to choose an aspect ratio that best represents the information you are trying to convey.

**Q5. Compare and contrast the three types of array multiplication between two numpy arrays: dot product, outer product, and regular multiplication of two numpy arrays.**

In NumPy, there are three types of multiplication between two arrays: dot product, outer product, and regular multiplication. Here's how they differ:

Dot product:

The dot product of two arrays A and B, denoted by A.dot(B), is a sum of the element-wise multiplication of the two arrays. The dot product is only defined for arrays of compatible dimensions, i.e., the number of columns in the first array must match the number of rows in the second array. The dot product is a scalar or a one-dimensional array, depending on the dimensions of the input arrays. Here's an example:

import numpy as np

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

B = np.array([[5, 6], [7, 8]])

dot\_product = A.dot(B)

print(dot\_product)

Output:

array([[19, 22],

[43, 50]])

Outer product:

The outer product of two arrays A and B, denoted by np.outer(A, B), is a matrix where the (i, j) element is the product of the i-th element of A and the j-th element of B. The outer product is defined for any two arrays, regardless of their dimensions. The outer product is a matrix with the shape (A.size, B.size). Here's an example:

import numpy as np

A = np.array([1, 2])

B = np.array([3, 4])

outer\_product = np.outer(A, B)

print(outer\_product)

Output:

array([[3, 4],

[6, 8]])

Regular multiplication:

The regular multiplication of two arrays A and B, denoted by A \* B, is an element-wise multiplication, where the (i, j) element of the result is the product of the (i, j) elements of A and B. The regular multiplication is only defined for arrays of the same shape. The regular multiplication is an array with the same shape as the input arrays. Here's an example:

import numpy as np

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

B = np.array([[5, 6], [7, 8]])

regular\_product = A \* B

print(regular\_product)

Output:

array([[ 5, 12],

[21, 32]])

In summary, the dot product is a sum of element-wise multiplication and is only defined for arrays of compatible dimensions, the outer product is a matrix of element-wise multiplication and is defined for any two arrays, and the regular multiplication is an element-wise multiplication and is only defined for arrays of the same shape.

**Q6. Before you buy a home, which numpy function will you use to measure your monthly mortgage payment?**

Numpy is a library in Python primarily used for mathematical operations and calculations. While it has functions to perform various calculations, it doesn't have a specific function to calculate monthly mortgage payments. However, you can use mathematical formulas to calculate the monthly mortgage payment.

The formula to calculate the monthly mortgage payment is:

M = P \* (r \* (1 + r)^n) / ((1 + r)^n - 1)

where

M = monthly mortgage payment

P = principal amount (the amount borrowed to buy the home)

r = monthly interest rate (annual interest rate divided by 12)

n = total number of monthly payments (number of years \* 12)

To use this formula, you can define the variables in Python and then calculate the monthly mortgage payment using standard arithmetic operations. Here's an example code snippet to calculate the monthly mortgage payment using this formula:

import numpy as np

# define variables

principal = 300000

interest\_rate = 0.05 / 12

num\_years = 30

num\_payments = num\_years \* 12

# calculate monthly mortgage payment

monthly\_payment = np.round(principal \* (interest\_rate \* (1 + interest\_rate)\*\*num\_payments) / ((1 + interest\_rate)\*\*num\_payments - 1), 2)

print("Monthly Mortgage Payment: $", monthly\_payment)

In this example, the np.round() function is used to round the monthly payment to two decimal places.

**Q7. Can string data be stored in numpy arrays? If so, list at least one restriction that applies to this data.**

Yes, string data can be stored in numpy arrays using the numpy module's ndarray data type with the dtype parameter set to 'S' or 'U' (for bytes and Unicode strings, respectively).

One restriction that applies to string data stored in numpy arrays is that, unlike numerical data, the length of the strings must be fixed and predetermined when the array is created. This means that if you have a numpy array with strings of a certain length, you cannot append longer strings to the array, and attempting to do so will result in a value error. Additionally, numpy arrays with string data can be less memory-efficient than arrays with numerical data due to the extra space needed to store each character.