Lecture 03 - Accessing Elements

Once we have put the data in Numpy Arrays, obviously we should be able to read/access the data. If we can't retreive the data back, there is no point in storing it at first place.

In Numpy, there are three ways to access the data, which we will see in detail in the upcoming sections.

1. **Basic Indexing and Slicing**
2. **Boolean Indexing/Masking**
3. **Array Indexing/Fancy Indexing**

So, let's jump into these topics.

03.1 - Basic Indexing and Slicing

Basic indexing in NumPy allows you to access individual elements or a range of elements in an array using square brackets `[]`. It is the simplest and most common way to retrieve specific elements from an array.

Consider a scenario where you have an array representing the daily temperatures in a week:

In [1]:
```
import numpy as np

temperatures = np.array([25, 26, 28, 24, 23, 26, 27])
```

Now, let's explore different ways to perform basic indexing:

1. **Accessing Individual Elements:**

You can access individual elements by specifying the index of the desired element within the square brackets.

In [2]:
```
print(temperatures[0])   # Access the temperature on the first day
print(temperatures[3])   # Access the temperature on the fourth day
```

```
25
24
```

2. **Accessing a Range of Elements:**

You can access a range of elements by specifying the starting and ending indices within the square brackets using the colon `:`.

In [3]:
```
print(temperatures[1:4])   # Access temperatures from the second to the f
print(temperatures[2:])    # Access temperatures from the third day to th
```

```
[26 28 24]
[28 24 23 26 27]
```

It's important to note that indexing in NumPy starts from 0, so the first element is accessed using index 0, the second element with index 1, and so on.

03.2 - Boolean Indexing (also known as Masking)

Boolean indexing in NumPy allows you to select elements from an array based on a boolean condition or mask. It provides a powerful way to filter and extract data that satisfies a particular criterion.

Consider a scenario where you have an array representing the scores of students in a class:

In [4]:
```
scores = np.array([85, 90, 78, 92, 88, 95, 80])
```

Now, let's explore how to use boolean indexing to filter and extract elements.

First, you need to create a boolean mask by applying a condition to the array. The result is a boolean array of the same shape as the original array, where each element indicates whether the corresponding element in the original array satisfies the condition.

In [5]:
```
passing_mask = scores >= 90
print(passing_mask)
```

```
[False  True False  True False  True False]
```

In this example, scores >= 90 creates a boolean mask where True corresponds to scores that are equal to or greater than 90. The resulting mask indicates which students passed the exam.

Once you have the boolean mask, you can use it to select the elements that satisfy the condition by passing the mask inside the square brackets.

In [6]:
```
passing_scores = scores[passing_mask]
print(passing_scores)
```

```
[90 92 95]
```

In this example, `scores[passing_mask]` retrieves the scores of students who passed the exam based on the boolean mask.

03.3 - Array Indexing (also known as Fancy Indexing)

Integer array indexing in NumPy allows you to select elements from an array using an array of integer indices. It provides a way to access specific elements by specifying their positions with the help of another array.

Imagine you have a list of students in a class, and you want to select specific students based on their positions in the list. Each student is assigned a unique ID,

and you have an array representing their IDs:

In [7]:
```
student_ids = np.array([101, 105, 110, 115, 120, 125])
```

Now, let's explore how to use integer array indexing to select specific students:

To select specific students, you need to create another array containing the indices of the students you want to choose. These indices represent the positions of the students in the original list.

```
indices = np.array([1, 3, 5])
```
In this example, indices is an array that contains the positions of the students you want to select.

Once you have the array of indices, you can use it to select the desired students from the original array by passing it inside the square brackets.

```
selected_students = student_ids[indices]
print(selected_students)
```
In this example, `student_ids[indices]` retrieves the students' IDs at the positions specified by the indices array.

In [8]:
```
indices = np.array([1, 3, 5])
selected_students = student_ids[indices]
print(selected_students)
```

```
[105 115 125]
```

By utilizing these various indexing techniques, you can selectively access and manipulate elements in NumPy arrays based on their positions, offering flexibility and control over data extraction and manipulation tasks.

03.4 Working with 2D Arrays

Imagine you have a 2D array representing the daily closing prices of different stocks over a certain period of time. Each row corresponds to a stock, and each column represents a trading day.

In [9]:
```
# Create a 2D array of daily closing prices (5 stocks x 30 trading days)
closing_prices = np.random.randint(low=50, high=200, size=(5, 30))
print("Closing Prices:")
print(closing_prices)
```

```
Closing Prices:
[[131 148 137  85 197 181 102  68 114 197  73 114 148 183  58  79 165 12
8
  175  56 191  85 163 148 141 149 101 126 123  61]
 [ 60 183 130 195 105 136  74 188  68 106 152 108 188 157 199 191  53 10
6
  100 180  96 140 180 165 107  77  79  70 150  50]
 [187 194 155 135 129 132 132 181 152 171 189 183 117  81  83 131 119 19
2
  188  95  75 122  93 162 135 105 116 198  75 151]
 [197 180  61  76 164 176 180  92 156 123 161  97 178 185 105  69  89 10
9
  106  85  97  96 133 190 199  95 103 180 132  64]
 [188 139  71  70 140 166  72 112 126  71 184 150  62 125  98  94 149  8
2
  138  99 139  80  84  91 171 195 121 146 115 172]]
```

1. **Accessing a Row:**

In this case, you may want to access a specific row corresponding to a particular stock. You can do this by specifying the row index.

In [10]:
```
# Access the closing prices of the 3rd stock (row index 2)
stock_3_prices = closing_prices[2]
print("Closing Prices for Stock 3:")
print(stock_3_prices)
```

```
Closing Prices for Stock 3:
[187 194 155 135 129 132 132 181 152 171 189 183 117  81  83 131 119 192
 188  95  75 122  93 162 135 105 116 198  75 151]
```

2. **Accessing Multiple Rows:**

To access multiple rows, you can specify a range of row indices using slicing.

In [11]:
```
# Access the closing prices for stocks 2, 4, and 5 (row indices 1, 3, and
selected_stocks_prices = closing_prices[[1, 3, 4]]
print("Closing Prices for Selected Stocks:")
print(selected_stocks_prices)
```

```
Closing Prices for Selected Stocks:
[[ 60 183 130 195 105 136  74 188  68 106 152 108 188 157 199 191  53 10
6
  100 180  96 140 180 165 107  77  79  70 150  50]
 [197 180  61  76 164 176 180  92 156 123 161  97 178 185 105  69  89 10
9
  106  85  97  96 133 190 199  95 103 180 132  64]
 [188 139  71  70 140 166  72 112 126  71 184 150  62 125  98  94 149  8
2
  138  99 139  80  84  91 171 195 121 146 115 172]]
```

3. **Accessing a Column:**

To access a specific column, you can use indexing with the column index.

In [12]:
```
# Access the closing prices for the 5th trading day (column index 4)
day_5_prices = closing_prices[:, 4]
```

```
print("Closing Prices for Day 5:")
print(day_5_prices)
```

```
Closing Prices for Day 5:
[197 105 129 164 140]
```

4. **Accessing Multiple Columns:**

To access multiple columns, you can use slicing with column indices.

In [13]:
```
# Access the closing prices for the 3rd, 5th, and 7th trading days (colum
selected_days_prices = closing_prices[:, [2, 4, 6]]
print("Closing Prices for Selected Days:")
print(selected_days_prices)
```

```
Closing Prices for Selected Days:
[[137 197 102]
 [130 105  74]
 [155 129 132]
 [ 61 164 180]
 [ 71 140  72]]
```

5. **Boolean Indexing Example:**

Boolean indexing allows you to select specific rows based on a condition or criteria.

In [14]:
```
# Access the closing prices for stocks with prices above 150
high_price_stocks = closing_prices[closing_prices > 150]
print("Closing Prices for High Price Stocks:")
print(high_price_stocks)
```

```
Closing Prices for High Price Stocks:
[197 181 197 183 165 175 191 163 183 195 188 152 188 157 199 191 180 180
 165 187 194 155 181 152 171 189 183 192 188 162 198 151 197 180 164 176
 180 156 161 178 185 190 199 180 188 166 184 171 195 172]
```

6. **Array Indexing Example:**

Array indexing can be used to extract specific data points based on given indices.

In [15]:
```
# Define the indices for the desired data points
row_indices = np.array([0, 2, 4])
column_indices = np.array([1, 5, 9])

# Access the closing prices for the specified data points
selected_prices = closing_prices[row_indices[:, np.newaxis], column_indic
print("Selected Closing Prices:")
print(selected_prices)
```

```
Selected Closing Prices:
[[148 181 197]
 [194 132 171]
 [139 166  71]]
```

These situations demonstrate how to access rows, multiple rows, columns, multiple columns, and how to use array indexing for rows and columns in NumPy arrays. By

applying appropriate indexing techniques, you can extract and analyze specific subsets of data for further analysis or processing.
