Fourth Week

Ungraded Problems

TA Solution by Abhisek Sarkar

as20ms091@iiserkol.ac.in

Read Numpy Documentation at https://numpy.org/doc/stable/user/absolute_beginners.html

Read Python list documentation at https://docs.python.org/3/tutorial/datastructures.html

Read python dictionaries documentation at https://docs.python.org/3/tutorial/datastructures.html#dictionaries

Q1. Use list comprehension to create a list containing all the elements in list L1 and not in L2.

In [1]:
```
L1 = [1, 2, 3, 4, 5]
L2 = [4, 5, 6, 7, 8]

# List comprehension to get elements in L1 but not in L2
result = [item for item in L1 if item not in L2]

print(result)
```

```
[1, 2, 3]
```

List Comprehension to Find Elements in L1 Not in L2

Problem:

We want to create a list that contains all the elements that are in list `L1` but not in list `L2`.

Approach:

List Comprehension: List comprehension in Python allows us to create a new list by applying an expression to each item in an existing list. It has the following syntax:

```
[expression for item in iterable if condition]
```
Here:

- `expression` : The value or transformation to be applied to each item.
- `iterable` : The sequence (like a list) that we are iterating over.
- `condition` : An optional filter that only includes items satisfying the condition.

Solution Explanation: We want to find all items in `L1` that are not present in `L2`. The condition used in the list comprehension checks if an item from `L1` is not in `L2`.

- `item for item in L1` : This part iterates over each element in `L1`.
- `if item not in L2` : This condition filters out any item in `L1` that also exists in `L2`.

Code Walkthrough:

- We loop through each element in `L1`.
- For each element, we check whether it is present in `L2` using the `not in` operator.
- If the element is not found in `L2`, it is added to the result list.

Efficiency:

- The `not in` operation checks membership in `L2` for each element of `L1`, which is an O(n) operation.
- Thus, for m elements in `L1` and n elements in `L2`, the time complexity is approximately O(m * n).

Example: Given:

```
L1 = [1, 2, 3, 4, 5]
    L2 = [4, 5, 6, 7, 8]
```
The result will be:

```
   result = [1, 2, 3]
```

Final Code:

```
L1 = [1, 2, 3, 4, 5]
    L2 = [4, 5, 6, 7, 8]
    result = [item for item in L1 if item not in L2]
    print(result)
```

Conclusion: List comprehension is a concise and efficient way to solve problems like this. The solution is simple to understand and leverages Python's powerful built-in functionality.

Q2. Use numpy array to generate the following pattern for n (taken from user) lines (shown for n = 4):

1.0

1.0 1.5 2.0

1.0 1.5 2.0 2.5 3.0

1.0 1.5 2.0 2.5 3.0 3.5 4.0

In [10]:
```
import numpy as np

# Function to generate the pattern
def generate_pattern(n):
    start = 1.0  # Start at 1.0

    for i in range(1, n + 1):
        # Generate the sequence for the current row with appropriate step and length
        row = np.arange(1.0, 1.0 + 0.5 * (2 * i - 1), 0.5)

        # Print the row
        print(" ".join(map(str, row)))

# Input from the user for number of lines (n)
n = int(input("Enter the number of lines (n): "))

# Generate and print the pattern
generate_pattern(n)
```

```
1.0
1.0 1.5 2.0
1.0 1.5 2.0 2.5 3.0
1.0 1.5 2.0 2.5 3.0 3.5 4.0
```

Step-by-Step Breakdown

1. Importing Libraries

```
import numpy as np
```

- **`numpy`** is a popular Python library for numerical computing. Here, we are using it to generate an array of numbers with specific intervals (`np.arange`), which makes it easy to create sequences.

2. Defining the Function `generate_pattern(n)`

```
def generate_pattern(n):
```

- The function `generate_pattern` is defined to take a single argument `n` , which represents the number of rows in the pattern.
- This function will generate the desired pattern of numbers based on `n` .

3. Initializing the Starting Value

```
start = 1.0
```

- We set the starting value to **1.0**. The pattern always begins at 1.0 for each row.

4. Looping Through Rows

```
for i in range(1, n + 1):
```

- This `for` loop iterates through the rows of the pattern.
- The loop runs from `i = 1` to `i = n` (where `n` is the number of lines or rows the user wants).
- The range `range(1, n + 1)` generates numbers from 1 to `n` (inclusive).

5. Generating Each Row with `np.arange`

```
row = np.arange(1.0, 1.0 + 0.5 * (2 * i - 1), 0.5)
```

- **`np.arange(start, stop, step)`** generates an array starting from `start`, ending before `stop`, with steps of size `step`.
- Let's break down the parameters:
  - `1.0` is the start value of the sequence.
  - The second value `1.0 + 0.5 * (2 * i - 1)` defines the **end** value of the sequence:
    - `(2 * i - 1)` calculates the number of elements in the row. For each row `i`, the number of elements increases based on the formula `2 * i - 1`.
    - We multiply this by `0.5` to determine the appropriate step size for the row.
    - The second argument of `np.arange` is the end value of the range. For each row, the number of values increases by 0.5 intervals.
  - `0.5` is the step value, which means the numbers in each row will increment by 0.5.

6. **Printing the Row**

```
print(" ".join(map(str, row)))
```

- `map(str, row)` converts each number in the `row` array to a string so they can be joined together and printed.
- `" ".join(...)` joins the string numbers with a space (`" "`), so the numbers are printed with spaces in between.

For example:

- If the row is `[1.0, 1.5, 2.0]`, it will be printed as: `1.0 1.5 2.0`.

7. **User Input for `n`**

```
n = int(input("Enter the number of lines (n): "))
```

- This line asks the user to input the number of rows (`n`) they want in the pattern.
- The `input()` function captures the user input, and `int()` converts it to an integer value.

8. **Calling the Function**

```
generate_pattern(n)
```

- After receiving the user's input, we call the function `generate_pattern(n)` to generate and print the pattern based on the given number of rows.

Example

Let's look at a practical example:

Input:

```
Enter the number of lines (n): 3
```

Output:

```
1.0
1.0 1.5 2.0
1.0 1.5 2.0 2.5 3.0
```

Explanation:

- **Row 1**: For `i = 1`, the sequence is `[1.0]` (since `2 * 1 - 1 = 1`, and the row length is 1).
- **Row 2**: For `i = 2`, the sequence is `[1.0, 1.5, 2.0]` (since `2 * 2 - 1 = 3`, and the row length is 3).
- **Row 3**: For `i = 3`, the sequence is `[1.0, 1.5, 2.0, 2.5, 3.0]` (since `2 * 3 - 1 = 5`, and the row length is 5).

Summary

- The code uses a loop to generate each row of the pattern.
- The number of elements in each row increases as the row number increases (e.g., Row 1 has 1 element, Row 2 has 3 elements, and Row 3 has 5 elements).
- The numbers in each row increment by 0.5.
- The output is printed in a visually appealing format where each row is printed on a new line with numbers separated by spaces.

This approach ensures that the pattern is generated dynamically based on the user's input.

Q3. Write a function encrypt() that accepts a string and maps each character of the string to the corresponding alphabet in the opposite order of alphabets. E.g. 'a' will be mapped to 'z', 'b' will be mapped to 'y', 'z' will be mapped to 'a' and so on. So, encrypt() will map 'zbc' to 'ayx'. Assume that the input string is in the lower case. Note that ord('a') gives the ASCII value of the character 'a' (i.e. 97) and chr(97) gives the character equivalent of 97 (i.e. 'a').

In [15]:
```
def encrypt(input_string):
    encrypted_string = ""
```

```
    for ch in input_string:
        # Calculate the reversed position
        reversed_char = chr(ord('a') + (25 - (ord(ch) - ord('a'))))
        encrypted_string += reversed_char
    return encrypted_string

string = input("Input your string:") # Example: 'zbc' , output will be 'ayx'

print(encrypt(string))
```

```
ayx
```

Let's break down the code and the approach used to solve the problem step-by-step, explaining each part in detail.

Problem Restatement

We are given a string `input_string` consisting of lowercase letters (from `'a'` to `'z'`). We need to encrypt it by mapping each character to its opposite character in the alphabet. For example, `a` maps to `z`, `b` maps to `y`, and so on.

The goal is to create a function `encrypt()` that will:

1. Read the input string.
2. Encrypt each character by mapping it to its opposite in the alphabet.
3. Return the resulting encrypted string.

Step-by-Step Approach

1. **Understanding the Mapping:**
   - We need to map each letter in the alphabet to its corresponding letter in reverse order.
     - `'a'` (the first letter) should be mapped to `'z'` (the last letter).
     - `'b'` should be mapped to `'y'`.
     - `'c'` should be mapped to `'x'`, and so on.
   - The relationship between the letters can be described mathematically by using their ASCII values.
2. **The ASCII Values:**
   - Each character in Python has an ASCII (or Unicode) value, which can be obtained using the `ord()` function. For example:
     - `ord('a') = 97`
     - `ord('b') = 98`
     - `ord('z') = 122`
   - To reverse the alphabet, we need to compute the position of each character from the end of the alphabet. If `ch` is a character, its position can be calculated as `ord(ch) - ord('a')` (this gives us the position of the character from the start of the alphabet).
3. **Reversing the Position:**
   - For a given character `ch`, we need to find the corresponding character from the end of the alphabet.
   - The position of the character from the end of the alphabet is calculated as:
     `reversed_pos = 25 - (ord(ch) - ord('a'))`
     This formula works because:
     - If the character is `'a'`, its position is `0` (from the start of the alphabet), and the reversed position will be `25` (from the end of the alphabet).
     - If the character is `'b'`, its position is `1`, and the reversed position will be `24`, and so on.
4. **Reversing the Character:**
   - Once we have the reversed position, we can convert it back to a character using the `chr()` function:
     `reversed_char = chr(ord('a') + reversed_pos)`
     - This converts the reversed position back to a character.
     - For example, for the reversed position `25`, this would give `'z'`.
5. **Building the Encrypted String:**
   - The function processes each character in the input string, calculates its reverse, and appends it to the result.
   - Finally, the function returns the encrypted string.

Code Explanation

Here is the code again with the step-by-step explanation:

```
def encrypt(input_string):
    encrypted_string = ""  # Initialize an empty string to hold the encrypted result

    # Loop through each character in the input string
    for ch in input_string:
        # Calculate the position of the character from 'a' (0-based index)
        # ord(ch) - ord('a') gives the position of ch in the alphabet
        # 25 - this position gives the reverse position from 'z'
        reversed_char = chr(ord('a') + (25 - (ord(ch) - ord('a'))))  # Get the corresponding opposite character

        # Append the reversed character to the encrypted string
        encrypted_string += reversed_char
```

```
    # Return the encrypted string
    return encrypted_string
```

Step-by-Step Walkthrough of the Code:

1. **Initialization of `encrypted_string`:**
   - `encrypted_string = ""` initializes an empty string where we will store the final result.
2. **Looping Through Each Character in the Input String:**
   - `for ch in input_string:` starts a loop to process each character `ch` in the `input_string`.
3. **Calculating the Reversed Character:**
   - `ord(ch)` gives the ASCII value of the character `ch`.
   - `ord('a')` gives the ASCII value of `'a'` (which is 97). Subtracting `ord('a')` from `ord(ch)` gives the position of `ch` in the alphabet, starting from 0 for `'a'`.
   - `25 - (ord(ch) - ord('a'))` computes the position of the opposite character in the alphabet. This is because `'a'` is at position 0 (so its reverse is at position 25), `'b'` is at position 1 (so its reverse is at position 24), and so on.
   - `chr(ord('a') + (25 - (ord(ch) - ord('a'))))` then converts the reversed position back to a character.
4. **Appending to the Result:**
   - `encrypted_string += reversed_char` adds the reversed character to the `encrypted_string`.
5. **Returning the Final Encrypted String:**
   - After processing all the characters, `return encrypted_string` returns the resulting encrypted string.

Example Walkthrough:

Let's say we have the input `"zbc"`.

1. **For 'z':**
   - `ord('z') = 122`
   - Position from 'a': `122 - 97 = 25`
   - Reversed position: `25 - 25 = 0`
   - Reversed character: `chr(97 + 0) = 'a'`
2. **For 'b':**
   - `ord('b') = 98`
   - Position from 'a': `98 - 97 = 1`
   - Reversed position: `25 - 1 = 24`
   - Reversed character: `chr(97 + 24) = 'y'`
3. **For 'c':**
   - `ord('c') = 99`
   - Position from 'a': `99 - 97 = 2`
   - Reversed position: `25 - 2 = 23`
   - Reversed character: `chr(97 + 23) = 'x'`

Thus, the encrypted string for the input `"zbc"` is `"ayx"`.

Time Complexity:

- The function processes each character of the string individually.
- The time complexity is **O(n)**, where `n` is the length of the input string.

Space Complexity:

- The space complexity is also **O(n)**, as we store the encrypted string of length `n` in `encrypted_string`.

Conclusion:

The `encrypt()` function works by reversing the position of each character in the alphabet using simple arithmetic with ASCII values. It then constructs the encrypted string by concatenating the reversed characters and returns it. This solution is efficient, processing each character in linear time and space.

Q4. Write a python program to swap the left and right halves of an input string s; the middle element will be unchanged for an odd length string. E.g. ‘kripa’ will be converted to ‘paikr’; ‘aman’ will be converted to ‘anam’.

In [16]:
```
def swap_halves(s):
    n = len(s)
    # Calculate the middle index
    mid = n // 2
```

```python
    # If the string length is odd, the middle character stays in the same place
    if n % 2 == 0:
        left_half = s[:mid]
        right_half = s[mid:]
    else:
        left_half = s[:mid]
        middle = s[mid]
        right_half = s[mid + 1:]

    # Return the swapped halves, considering odd length
    if n % 2 == 0:
        return right_half + left_half
    else:
        return right_half + middle + left_half

# Test cases
print(swap_halves('Abhisek'))  # Output: 'sekiAbh' # odd
print(swap_halves('Mbappé'))   # Output: 'ppéMba' # even

# you can take input() aswell
```

```
sekiAbh
ppéMba
```

Approach:

We want to swap the left and right halves of a string `s`. If the string has an odd length, the middle character should remain in its position, and only the left and right halves should be swapped. If the string has an even length, the left and right halves are simply swapped.

Steps:

1. **Determine the Length of the String (`n`)**: The length of the string `s` is stored in variable `n` using the built-in `len()` function.
2. **Calculate the Middle Index (`mid`)**: The middle index is calculated using integer division: `mid = n // 2`. This will give the index of the middle element if the string length is odd, or the boundary between the two halves if the string length is even.
3. **Check If the String Length is Even or Odd**: We check if the string has an even or odd length by using the modulo operator: `n % 2 == 0`. This will return `True` if the length is even, and `False` if the length is odd.
4. **Handling the Even-Length String**:
   - When the string has an even length, it is split exactly in half. The left half is the substring from index `0` to `mid`, and the right half is the substring from `mid` to the end of the string.
   - The two halves are swapped, and the resulting string is returned.
5. **Handling the Odd-Length String**:
   - When the string has an odd length, the left half is the substring from index `0` to `mid`.
   - The middle character is at index `mid`, so it is stored in the variable `middle`.
   - The right half is the substring from `mid + 1` to the end of the string.
   - The swapped string is then constructed as `right_half + middle + left_half`.

Code Explanation:

```python
def swap_halves(s):
    n = len(s)  # Get the length of the string
    mid = n // 2  # Calculate the middle index

    # If the string length is even, split it into two halves
    if n % 2 == 0:
        left_half = s[:mid]  # Left half from 0 to mid
        right_half = s[mid:]  # Right half from mid to end
    else:
        left_half = s[:mid]  # Left half from 0 to mid (excluding the middle element)
        middle = s[mid]  # Middle character (unchanged)
        right_half = s[mid + 1:]  # Right half from mid+1 to end

    # If the string has an even length, swap the halves
    if n % 2 == 0:
        return right_half + left_half
    else:
        # If the string has an odd length, swap the halves while keeping the middle element in place
        return right_half + middle + left_half

# Test cases
print(swap_halves('Abhisek'))  # Output: 'sekiAbh' (odd-length string)
print(swap_halves('Mbappé'))   # Output: 'ppéMba' (even-length string)
```

Explanation of the Code:

1. **Input:** The function `swap_halves(s)` takes a string `s` as input.
2. **Determine Length and Middle:**
   - `n = len(s)` gets the length of the string.

- `mid = n // 2` finds the middle index of the string. This is crucial to split the string into left and right halves.

3. **Even-Length Case:**

   - If the string has an even length (`n % 2 == 0`), it is divided into two halves.
   - The left half is `s[:mid]` (from index `0` to `mid`).
   - The right half is `s[mid:]` (from index `mid` to the end).
   - These halves are swapped: `right_half + left_half` .

4. **Odd-Length Case:**

   - If the string has an odd length (`n % 2 != 0`), we take:
     - The left half `s[:mid]` (from index `0` to `mid-1`).
     - The middle element `middle = s[mid]` .
     - The right half `s[mid+1:]` (from index `mid+1` to the end).
   - The result is constructed as `right_half + middle + left_half` , keeping the middle element in place.

Example Walkthrough:

Test Case 1: `swap_halves('Abhisek')`

- Length of the string: `n = 7` (odd length).
- Middle index: `mid = 7 // 2 = 3` (the middle element is `'h'`).
- Left half: `'Abh'` (from index `0` to `2`).
- Middle element: `'i'` (at index `3`).
- Right half: `'sek'` (from index `4` to the end).
- The result: `'sek' + 'i' + 'Abh'` → `'sekiAbh'` .

Test Case 2: `swap_halves('Mbappé')`

- Length of the string: `n = 6` (even length).
- Middle index: `mid = 6 // 2 = 3` .
- Left half: `'Mba'` (from index `0` to `2`).
- Right half: `'ppé'` (from index `3` to the end).
- The result: `'ppé' + 'Mba'` → `'ppéMba'` .

Time Complexity:

- The time complexity of this solution is **O(n)**, where `n` is the length of the input string. This is because we are slicing the string and performing constant-time operations.

Space Complexity:

- The space complexity is also **O(n)**, since we create new strings (slices) to store the left half, right half, and middle element.

User Input Version:

You can modify the program to take user input as follows:

```
def swap_halves(s):
    n = len(s)
    mid = n // 2

    if n % 2 == 0:
        left_half = s[:mid]
        right_half = s[mid:]
    else:
        left_half = s[:mid]
        middle = s[mid]
        right_half = s[mid + 1:]

    if n % 2 == 0:
        return right_half + left_half
    else:
        return right_half + middle + left_half

# Taking input from the user
input_str = input("Enter a string: ")
print("Swapped string:", swap_halves(input_str))
```

Now, this code will prompt the user to input a string, and then the function will swap the halves based on the string's length (odd or even).

Q5. Input a list from the user (containing duplicates) and create another list containing only the non-repeating elements. E.g. if the input list is [1, 2, 2, 3, 4, 4, 5], the output list will be [1, 3, 5].

In [17]:
```
def get_non_repeating_elements(input_list):
    result = []
    for item in input_list:
```

```
        if input_list.count(item) == 1:
            result.append(item)
    return result

# Example usage
input_list = [1, 2, 2, 3, 4, 4, 5]
output_list = get_non_repeating_elements(input_list)
print("Non-repeating elements:", output_list)
```

```
Non-repeating elements: [1, 3, 5]
```

Approach for Finding Non-Repeating Elements in a List:

To solve this problem, the goal is to create a new list that contains only the non-repeating elements from the input list. In other words, we need to identify and extract all elements that appear exactly once in the list. Let's break down the steps involved:

Step-by-step Solution:

1. **Input List:**
   - We are given a list `input_list` which may contain duplicate elements.
   - Example input: `[1, 2, 2, 3, 4, 4, 5]` .
2. **Iterating Through the List:**
   - To identify non-repeating elements, we need to iterate through the list and examine each element.
   - For each element, we check how many times it appears in the list.
3. **Counting Occurrences:**
   - We use the `count()` method of the list to count how many times an element appears in the `input_list` .
   - `input_list.count(item)` returns the number of occurrences of `item` in the list.
4. **Filtering Non-Repeating Elements:**
   - If an element appears **exactly once** (`input_list.count(item) == 1`), it is considered a non-repeating element and will be added to a new list called `result` .
5. **Return the Result:**
   - After checking all the elements, the `result` list will contain only the non-repeating elements.
   - This list is then returned as the output.

Code Implementation:

```
def get_non_repeating_elements(input_list):
    result = []  # Initialize an empty list to store non-repeating elements

    # Loop through each item in the input list
    for item in input_list:
        # Count occurrences of each item in the list
        if input_list.count(item) == 1:
            # If the item occurs exactly once, append it to the result list
            result.append(item)

    # Return the final list of non-repeating elements
    return result

# Example usage
input_list = [1, 2, 2, 3, 4, 4, 5]
output_list = get_non_repeating_elements(input_list)
print("Non-repeating elements:", output_list)
```

Theory Behind the Code:

1. **`count()` Method:**
   - The `count()` method is used to determine how many times a specific element appears in the list. It scans the entire list and returns the number of occurrences.
   - Example: `input_list.count(2)` would return `2` if the number `2` appears twice in the list.
2. **Time Complexity:**
   - The `count()` method, when used inside a loop, results in a time complexity of $O(n^2)$ in the worst case because for each item in the list, `count()` scans the entire list again to count occurrences.
   - This can be inefficient for large lists, so alternative approaches (using hashmaps or dictionaries) can optimize the process for better performance.

Example Walkthrough:

Let's go through the example input list `[1, 2, 2, 3, 4, 4, 5]` :

1. **First iteration (item = 1):**

- `count(1)` returns `1` because `1` appears once in the list.
- So, `1` is added to `result`.

2. **Second iteration (item = 2):**
   - `count(2)` returns `2` because `2` appears twice.
   - Since `2` is not non-repeating, it is not added to `result`.
3. **Third iteration (item = 2):**
   - Same result as the previous iteration (`2` is ignored).
4. **Fourth iteration (item = 3):**
   - `count(3)` returns `1` because `3` appears once.
   - So, `3` is added to `result`.
5. **Fifth iteration (item = 4):**
   - `count(4)` returns `2` because `4` appears twice.
   - Since `4` is not non-repeating, it is not added to `result`.
6. **Sixth iteration (item = 4):**
   - Same result as the previous iteration (`4` is ignored).
7. **Seventh iteration (item = 5):**
   - `count(5)` returns `1` because `5` appears once.
   - So, `5` is added to `result`.

Final `result`: `[1, 3, 5]`.

Thus, the output list contains only the elements that appeared exactly once in the input list.

Optimizing the Approach:

The above approach can be inefficient for large lists due to the repeated calls to `count()`. A more optimized solution would involve using a dictionary or `collections.Counter` to track the frequency of each element in a single pass.

Optimized Approach Using `collections.Counter`:

```python
from collections import Counter

def get_non_repeating_elements(input_list):
    # Use Counter to get the frequency of each element
    element_count = Counter(input_list)

    # List comprehension to filter out elements that appear exactly once
    result = [item for item in input_list if element_count[item] == 1]

    return result

# Example usage
input_list = [1, 2, 2, 3, 4, 4, 5]
output_list = get_non_repeating_elements(input_list)
print("Non-repeating elements:", output_list)
```

Explanation of Optimized Code:

- `Counter(input_list)` creates a dictionary-like object where keys are the elements and values are their respective frequencies.
- This allows us to check the frequency of each element in constant time O(1) rather than scanning the entire list repeatedly.
- The overall time complexity is reduced to O(n) with this approach.

Conclusion:

The initial approach using `count()` works for small lists but becomes inefficient for large ones. Using `collections.Counter` is a more efficient solution, as it computes frequencies in a single pass through the list, leading to an O(n) time complexity.

Q6. Print the numpy array [[1,2,3], [4, 5, 6]] column-wise by using slicing.

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

# Creating the array
arr = np.array([[1, 2, 3], [4, 5, 6]])

# Printing column-wise using slicing
for i in range(arr.shape[1]):  # Loop through columns
    print(arr[:, i])  # Print each column by slicing
```

```
[1 4]
[2 5]
[3 6]
```

Theory and Explanation:

In NumPy, we work with arrays (also known as matrices) that can have multiple dimensions. When working with two-dimensional arrays (matrices), the first dimension represents rows, and the second dimension represents columns.

Slicing in NumPy:

Slicing is a way of accessing a specific subset of an array. It follows the syntax `array[start:stop:step]`, where:

- `start` is the index to start the slice.
- `stop` is the index to end the slice (exclusive).
- `step` is the increment between each index in the slice.

For multi-dimensional arrays, we can apply slicing to each dimension separately. For example, to access the first column, we slice the array to include all rows but only the first column. This is done with the syntax `arr[:, i]`:

- The colon `:` represents all rows (i.e., every row from top to bottom).
- `i` is the column index, which allows us to select each column individually.

Step-by-Step Approach to Print Column-wise:

Given the 2D array:

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
```
This is a 2x3 matrix, meaning it has 2 rows and 3 columns:

```
[ [1, 2, 3]
  [4, 5, 6] ]
```

We want to print each column separately. To do this, we can loop through the columns and slice the array to print each one:

Steps:

1. **Determine the shape of the array**:
    - The `shape` attribute of a NumPy array gives us a tuple of the form `(rows, columns)`.
    - For this array, `arr.shape` returns `(2, 3)`, meaning it has 2 rows and 3 columns.
2. **Loop through each column**:
    - We use a `for` loop to iterate over the number of columns, which is given by `arr.shape[1]`.
    - This allows us to access each column index one by one.
3. **Slice the array to get each column**:
    - In the loop, we slice the array using `arr[:, i]`, where:
        - The `:` selects all rows.
        - `i` selects the `i`-th column.

Code Implementation:

```
import numpy as np

# Creating the array
arr = np.array([[1, 2, 3], [4, 5, 6]])

# Printing column-wise using slicing
for i in range(arr.shape[1]):  # Loop through columns
    print(arr[:, i])  # Print each column by slicing
```

How It Works:

1. `arr.shape[1]` gives `3` (the number of columns), so the loop will iterate over indices 0, 1, and 2.
2. `arr[:, i]` slices the array for each column:
    - For `i = 0`: `arr[:, 0]` returns the first column `[1, 4]`.
    - For `i = 1`: `arr[:, 1]` returns the second column `[2, 5]`.
    - For `i = 2`: `arr[:, 2]` returns the third column `[3, 6]`.

Output:

```
[1 4]
[2 5]
[3 6]
```

Each column is printed separately, with each element of the column on the same line. The slicing `arr[:, i]` effectively extracts one column at a time from the array.

Summary:

- **Slicing** is an important concept in NumPy for accessing parts of an array.

- `arr[:, i]` slices all rows and a specific column `i`.
- **Iteration** over the columns using a loop allows us to print the array column-wise.

This method of column-wise access is efficient and takes full advantage of NumPy's slicing capabilities.

Q7. Define a function calc_area (using def) that takes a radius (r) and returns the area of a circle (use math.pi). Take a list radius_list containing five radii values and use calc_area to store the corresponding areas in another list area_list.

Q8. Repeat the problem in (7) using a numpy array radius_np containing five radii values (same as in radius_list) to store the corresponding areas in another numpy array area_np.

In [2]:
```python
#Q7
import math

# Define the function to calculate the area of a circle
def calc_area(r):
    return math.pi * r ** 2

# List of radii
radius_list = [2, 3, 4, 5, 6]

# Using calc_area to get the areas and storing them in area_list
area_list = [calc_area(r) for r in radius_list]

# Display the area_list
print(area_list)
```

```
[12.566370614359172, 28.274333882308138, 50.26548245743669, 78.53981633974483, 113.09733552923255]
```

In [3]:
```python
#Q8
import numpy as np

# Define the function to calculate the area of a circle
def calc_area(r):
    return np.pi * r ** 2

# NumPy array of radii
radius_np = np.array([2, 3, 4, 5, 6])

# Using calc_area to get the areas and storing them in area_np
area_np = calc_area(radius_np)

# Display the area_np
print(area_np)
```

```
[ 12.56637061  28.27433388  50.26548246  78.53981634 113.09733553]
```

Problem Explanation & Approach:

We are tasked with two related problems:

1. **Q7:** Create a function `calc_area` that calculates the area of a circle using the formula (\text{Area} = \pi \times r^2), where `r` is the radius of the circle. Use a list of radii values, `radius_list`, and apply this function to calculate the corresponding areas, storing them in a new list, `area_list`.

2. **Q8:** Repeat the process but use NumPy arrays instead of lists. In this problem, the radii are stored in a NumPy array, `radius_np`, and the areas should be calculated and stored in another NumPy array, `area_np`. NumPy will allow for more efficient element-wise operations.

Q7 Solution (Using Python Lists):

In this solution, we are using Python's built-in `math` library to calculate the area of a circle. We take a list of radii values (`radius_list`), and for each radius, we calculate its corresponding area using the `calc_area` function. The results are stored in the `area_list`.

Code for Q7:

```python
import math

# Define the function to calculate the area of a circle
def calc_area(r):
    return math.pi * r ** 2

# List of radii
radius_list = [2, 3, 4, 5, 6]

# Using calc_area to get the areas and storing them in area_list
area_list = [calc_area(r) for r in radius_list]

# Display the area_list
print(area_list)
```

Step-by-Step Explanation:

1. **Function Definition:** The `calc_area` function accepts a radius `r` as input, and it calculates the area of the circle using the formula (\pi r^2) (where `math.pi` gives the value of (\pi)).
2. **Creating a List of Radii:** We create a list `radius_list` that contains five radii values: `[2, 3, 4, 5, 6]`.
3. **Using List Comprehension:** To apply the `calc_area` function to each radius in the `radius_list`, we use list comprehension. This approach applies `calc_area(r)` to each element in the `radius_list` and stores the results in a new list `area_list`.
4. **Printing the Result:** Finally, the list `area_list` (containing the corresponding areas for each radius) is printed.

Output:

```
[12.566370614359172, 28.274333882308138, 50.26548245743669, 78.53981633974483, 113.09733552923255]
```

These are the areas corresponding to the radii 2, 3, 4, 5, and 6, calculated using the formula (\pi r^2).

Q8 Solution (Using NumPy Arrays):

In this solution, we replace the Python list with a NumPy array (`radius_np`). NumPy allows us to perform element-wise operations on arrays, so we don't need to explicitly iterate over the elements like we did with the list in Q7. Instead, NumPy automatically applies the formula (\pi r^2) to each element in the array.

Code for Q8:

```python
import numpy as np

# Define the function to calculate the area of a circle
def calc_area(r):
    return np.pi * r ** 2

# NumPy array of radii
radius_np = np.array([2, 3, 4, 5, 6])

# Using calc_area to get the areas and storing them in area_np
area_np = calc_area(radius_np)

# Display the area_np
print(area_np)
```

Step-by-Step Explanation:

1. **Importing NumPy:** We import the `numpy` library, which provides powerful array operations.
2. **Function Definition:** The `calc_area` function works similarly to the one in Q7, but now it accepts a NumPy array (`r`). Since NumPy arrays support element-wise operations, the formula (\pi r^2) is applied to each element of the array in a vectorized manner (automatically, without the need for explicit loops).
3. **Creating a NumPy Array of Radii:** We define a NumPy array `radius_np` with the same values as in Q7: `[2, 3, 4, 5, 6]`.
4. **Element-wise Calculation:** The `calc_area` function is applied directly to the `radius_np` array. NumPy computes the areas of all the circles in one step, and the result is stored in the `area_np` array.
5. **Printing the Result:** The `area_np` array is printed, containing the areas for each radius.

Output:

```
[ 12.56637061  28.27433388  50.26548246  78.53981634 113.09733553]
```

These are the areas corresponding to the radii 2, 3, 4, 5, and 6, calculated using the formula (\pi r^2).

Comparison Between Q7 and Q8:

- **Efficiency:** Using NumPy arrays (as in Q8) is more efficient than using Python lists (as in Q7), especially for large datasets. This is because NumPy is optimized for numerical operations and supports vectorized (element-wise) calculations.
- **Simplicity:** In Q8, we don't need to use explicit loops or list comprehensions. The NumPy array automatically handles the element-wise operation, making the code cleaner and more concise.
- **Flexibility:** While lists in Python are versatile and can hold different types of data, NumPy arrays are specifically designed for numerical computations and offer a wide range of functionality for mathematical operations.

Summary:

- **Q7:** We use Python lists to store the radii, and apply the `calc_area` function using list comprehension.
- **Q8:** We use NumPy arrays, which handle element-wise operations automatically, making the code cleaner and more efficient for large datasets.

Both solutions correctly calculate the areas of circles for given radii, but the use of NumPy provides significant advantages when handling arrays of data.

Q9. Input (use input()) the roll number and marks of three courses for 5 students and store it in a dictionary (d) such that the roll number is the key and the marks are stored in a numpy array of type float64 (the type of a numpy array A can be changed to float64 type by A.astype('float64')). For example, for a student with roll number 'ms1901' and marks 70, 80, 90, the dictionary entry will look like 'ms1901': array([70., 80., 90.])

Q.10 Continue with Q9 to create another dictionary 'd_sum' that contains the sum of the marks. So, the corresponding dictionary entry will look like 'ms1901': 240.0. To compute sum use numpy function sum() (Note: The sum of a numpy array A is calculated as numpy.sum(A))

In [5]:
```python
import numpy as np

def create_student_marks_dict(): # Q9
    """
    Create a dictionary to store student roll numbers and their marks.

    Returns:
        dict: Dictionary with roll numbers as keys and marks as numpy arrays.
    """
    d = {}
    for _ in range(5):
        roll_number = input("Enter roll number: ")
        marks = np.array([float(mark) for mark in input("Enter marks of three courses (space-separated): ").split()])
        d[roll_number] = marks.astype('float64')
    return d

def calculate_sum_of_marks(d): # Q10
    """
    Create a dictionary to store the sum of marks for each student.

    Args:
        d (dict): Dictionary with roll numbers as keys and marks as numpy arrays.

    Returns:
        dict: Dictionary with roll numbers as keys and sum of marks as values.
    """
    d_sum = {roll_number: np.sum(marks) for roll_number, marks in d.items()}
    return d_sum

def main():
    d = create_student_marks_dict()
    print("Student Marks Dictionary:")
    for roll_number, marks in d.items():
        print(f"{roll_number}: {marks}")

    d_sum = calculate_sum_of_marks(d)
    print("\nSum of Marks Dictionary:")
    for roll_number, total_marks in d_sum.items():
        print(f"{roll_number}: {total_marks}")

if __name__ == "__main__":
    main()
```

```
Student Marks Dictionary:
20ms091: [99. 98. 85.]
20ms001: [95. 85. 75.]
20ms020: [96. 86. 76.]
20ms110: [94. 84. 74.]
20ms025: [69. 68. 67.]

Sum of Marks Dictionary:
20ms091: 282.0
20ms001: 255.0
20ms020: 258.0
20ms110: 252.0
20ms025: 204.0
```

Step-by-Step Breakdown:

1. **Create a Dictionary to Store Student Roll Numbers and Marks**:

- **Input:** We will ask for user input to get the roll numbers of students and their corresponding marks in three courses.
- **Data Structure:** We use a **dictionary** to store the data, where:
  - The **key** is the student's roll number (a unique identifier for each student).
  - The **value** is a **numpy array** of three elements, each representing marks in one of the three courses.
- **Numpy Array:** We choose numpy arrays because they allow efficient manipulation of numerical data, especially for operations like summing the marks.

Process:

- We create an empty dictionary `d` to store the student data.
- For each student (5 in total as specified), we:
  - Take the student's roll number.
  - Take input for the marks of three courses in a single line, split them, and convert them into a numpy array of float values.
  - Store this data in the dictionary with the roll number as the key and the marks array as the value.

2. **Calculate the Sum of Marks for Each Student**:

- **Objective:** For each student, we need to compute the sum of the marks across the three courses.
- **Data Structure:** We use another dictionary `d_sum`, where:
  - The **key** is the student's roll number.
  - The **value** is the sum of the marks in the three courses.
- **Numpy Array Sum:** We use the `np.sum()` function to calculate the sum of marks for each student. This function operates efficiently on numpy arrays and gives us the total marks directly.
- **Process:**
  - We iterate over the `d` dictionary, applying `np.sum()` to the numpy array of marks for each student.
  - Store the sum of marks in the new dictionary `d_sum`.

3. **Display the Data**:

- **First, display the student marks dictionary** (`d`), where each entry shows the roll number and the corresponding numpy array of marks.
- **Then, display the sum of marks dictionary** (`d_sum`), where each entry shows the roll number and the computed sum of marks.

Explanation of Key Concepts:

1. Python Dictionary:

- A dictionary in Python is a collection of key-value pairs. It allows for fast lookups, insertions, and deletions.
- In our case:
  - Roll numbers are the **keys**.
  - Marks (as numpy arrays) or their sums (as float numbers) are the **values**.

2. Numpy Array:

- Numpy is a powerful library for numerical computing in Python.
- Arrays in numpy allow us to store multiple numerical values efficiently and perform operations like summing, averaging, and mathematical computations easily.
- By converting the marks into numpy arrays, we can make use of vectorized operations like summing the elements without the need for explicit loops, which makes the code more efficient.

3. Numpy's `np.sum()` Function:

- This function takes a numpy array and returns the sum of all its elements.
- For example, if we have an array `[80.0, 90.0, 85.0]`, `np.sum()` will return `255.0` (i.e., the sum of the marks).

Summary of the Flow:

1. **Input Gathering:**
   - Ask the user to input roll numbers and marks for 5 students.
   - Store the roll numbers as dictionary keys and marks as numpy arrays as values.
2. **Processing the Marks:**
   - For each student (through dictionary iteration), calculate the sum of their marks using `np.sum()` and store it in a new dictionary.
3. **Output the Data:**
   - Display the original student marks dictionary.
   - Display the sum of marks dictionary.

Q11. Check if two 1-D numpy arrays are equal or not using a for loop. Verify the result with the numpy function numpy.array_equal(a1, a2) that returns True if the arrays a1, a2 are equal.

In [6]:
```python
import numpy as np

# Create two 1-D numpy arrays for testing
a1 = np.array([1, 2, 3, 4])
a2 = np.array([1, 2, 3, 4])
a3 = np.array([1, 2, 3, 5])

# Check equality using a for loop
def are_arrays_equal_for_loop(arr1, arr2):
    if len(arr1) != len(arr2):  # First check if the arrays have the same length
        return False
```

```
    for i in range(len(arr1)):
        if arr1[i] != arr2[i]:
            return False
    return True

# Using the for loop to check equality
result_for_loop = are_arrays_equal_for_loop(a1, a2)
print(f"Using for loop, a1 and a2 are equal: {result_for_loop}")

# Verifying with numpy's array_equal function
result_numpy_equal = np.array_equal(a1, a2)
print(f"Using numpy.array_equal, a1 and a2 are equal: {result_numpy_equal}")

# Test with arrays that are not equal
result_for_loop_diff = are_arrays_equal_for_loop(a1, a3)
print(f"Using for loop, a1 and a3 are equal: {result_for_loop_diff}")

result_numpy_equal_diff = np.array_equal(a1, a3)
print(f"Using numpy.array_equal, a1 and a3 are equal: {result_numpy_equal_diff}")
```

```
Using for loop, a1 and a2 are equal: True
Using numpy.array_equal, a1 and a2 are equal: True
Using for loop, a1 and a3 are equal: False
Using numpy.array_equal, a1 and a3 are equal: False
```

Problem: Checking Equality Between Two 1-Dimensional NumPy Arrays

The task is to check if two 1-D NumPy arrays are equal, both using a for loop and NumPy's built-in `array_equal()` function. Let's walk through the approach and relevant theory behind the solution.

Approach

We will check if two 1-D NumPy arrays are equal using two different methods:

1. **Using a for loop:**
    - The idea behind using a for loop is to manually compare each element of the two arrays. If all the corresponding elements are the same and the arrays are of the same length, then the arrays are considered equal.
    - If any mismatch is found (either in the element or length), we return `False`.
    - Otherwise, we return `True` when all checks pass.
2. **Using NumPy's `array_equal()` function:**
    - NumPy provides a built-in function `np.array_equal()` to check if two arrays are exactly the same. It is optimized and handles edge cases, such as checking for differences in shape and data types automatically.
    - This function internally compares the shapes, elements, and data types of the arrays and returns `True` if all the conditions are met, otherwise `False`.

Code Implementation

```
import numpy as np

# Create two 1-D numpy arrays for testing
a1 = np.array([1, 2, 3, 4])
a2 = np.array([1, 2, 3, 4])
a3 = np.array([1, 2, 3, 5])

# Check equality using a for loop
def are_arrays_equal_for_loop(arr1, arr2):
    if len(arr1) != len(arr2):  # First check if the arrays have the same length
        return False
    for i in range(len(arr1)):  # Compare each element
        if arr1[i] != arr2[i]:  # If any mismatch, return False
            return False
    return True  # If all elements match, return True

# Using the for loop to check equality
result_for_loop = are_arrays_equal_for_loop(a1, a2)
print(f"Using for loop, a1 and a2 are equal: {result_for_loop}")

# Verifying with numpy's array_equal function
result_numpy_equal = np.array_equal(a1, a2)
print(f"Using numpy.array_equal, a1 and a2 are equal: {result_numpy_equal}")

# Test with arrays that are not equal
result_for_loop_diff = are_arrays_equal_for_loop(a1, a3)
print(f"Using for loop, a1 and a3 are equal: {result_for_loop_diff}")

result_numpy_equal_diff = np.array_equal(a1, a3)
print(f"Using numpy.array_equal, a1 and a3 are equal: {result_numpy_equal_diff}")
```

Explanation of the Code:

1. **Array Creation:**
    - We create three 1-D NumPy arrays (`a1` , `a2` , `a3`) for testing.

- `a1` and `a2` are identical arrays, while `a3` differs from `a1` in the last element.

2. **For Loop Method:**

   - The function `are_arrays_equal_for_loop()` first checks if the arrays have the same length.
   - If the lengths are different, it returns `False` immediately.
   - Then, it loops through each index, comparing the corresponding elements of both arrays.
   - If any element is different, it returns `False`. If no differences are found, it returns `True`.

3. **NumPy `array_equal()` Method:**

   - `np.array_equal()` directly checks if the two arrays are identical. It compares their shape, elements, and data types in a highly optimized manner and returns `True` if everything matches.

4. **Output:**

   - For `a1` and `a2`, both the for loop method and `np.array_equal()` method should return `True`, as the arrays are identical.
   - For `a1` and `a3`, the last element differs, so both methods should return `False`.

Expected Output:

```
Using for loop, a1 and a2 are equal: True
Using numpy.array_equal, a1 and a2 are equal: True
Using for loop, a1 and a3 are equal: False
Using numpy.array_equal, a1 and a3 are equal: False
```

Key Concepts and Theory:

1. **Array Length Comparison:**

   - Before comparing individual elements, it is essential to first check if the arrays have the same length. Arrays of different lengths cannot be equal.

2. **Element-wise Comparison:**

   - If the arrays are of the same length, we compare each element one by one.
   - In the for loop approach, this is done explicitly by iterating through the array indices and comparing the corresponding elements.
   - `np.array_equal()` abstracts this process and handles it efficiently.

3. **NumPy `array_equal()` Function:**

   - This function is specifically designed to check if two arrays are exactly equal, meaning they must have the same shape, dtype, and elements. It handles all the internal details, including shape comparison, element-wise checks, and data type validation.

4. **Time Complexity:**

   - The for loop method has a time complexity of **O(n)**, where **n** is the number of elements in the arrays. This is because it performs a comparison for each element of the arrays.
   - `np.array_equal()` also operates with a time complexity of **O(n)**, but it is typically optimized in C and may perform better for large arrays compared to the Python for loop.

Conclusion:

Both methods, using the for loop and `np.array_equal()`, are valid ways to check if two 1-D NumPy arrays are equal. The for loop gives more control over the process but is slower and more prone to errors when dealing with more complex array structures. On the other hand, `np.array_equal()` is a highly optimized solution that is simpler and faster for large arrays.

Q12. From a list [[1, 2, 3, 4, 11], [100, 55, 80, 33, 22]] create an equivalent numpy array arr. Apply an appropriate slicing operation on arr to print:

[[4 11]

[33 22]]

In [7]:
```python
import numpy as np

# Create the numpy array from the given list
arr = np.array([[1, 2, 3, 4, 11], [100, 55, 80, 33, 22]])

# Apply slicing to extract the desired portion
result = arr[:, -2:]

print(result)
```

```
[[ 4 11]
 [33 22]]
```

Approach and Explanation for the Solution

The task is to create a numpy array from a given list of lists and apply slicing to extract specific elements. Let's break this down step by step:

1. **Creating the Numpy Array**

We are given a list of lists:

```
[[1, 2, 3, 4, 11], [100, 55, 80, 33, 22]]
```

Each sublist represents a row, and each element in the sublist represents an element in the row. To work with this data efficiently, we can convert the list of lists into a **numpy array**.

Numpy arrays are more efficient than regular Python lists when it comes to handling numerical data, and they provide various mathematical and slicing operations that allow for faster computation.

To convert the given list into a numpy array, we use `np.array()`:

```
arr = np.array([[1, 2, 3, 4, 11], [100, 55, 80, 33, 22]])
```

Now, `arr` is a 2x5 numpy array, meaning it has **2 rows and 5 columns**:

```
arr =
[[  1   2   3   4  11]
 [100  55  80  33  22]]
```

2. **Understanding Numpy Array Slicing**

Numpy allows us to slice arrays to extract specific portions of them. The syntax for slicing is:

```
arr[start:stop, start:stop]
```

Where:

- `start:stop` specifies the range of indices you want to include. For example, `arr[:, -2:]` means:
  - `:` for the rows means "select all rows".
  - `-2:` means "select the last two columns".

In Python, negative indices count from the end. So, `-1` refers to the last element, `-2` refers to the second-to-last element, and so on.

3. **Slicing to Extract the Desired Elements**

We need to extract the last two columns for each row from the numpy array. We can do this using the slice `[:, -2:]`:

- `:` before the comma means "select all rows".
- `-2:` after the comma means "select from the second-to-last column to the end of the row".

Thus, the slicing operation `arr[:, -2:]` will return:

```
[[ 4 11]
 [33 22]]
```

Here's how it works:

- For the first row `[1, 2, 3, 4, 11]`, the last two elements are `4` and `11`.
- For the second row `[100, 55, 80, 33, 22]`, the last two elements are `33` and `22`.

4. **The Complete Code**

Now, combining all these steps into a complete code:

```
import numpy as np

# Create the numpy array from the given list
arr = np.array([[1, 2, 3, 4, 11], [100, 55, 80, 33, 22]])

# Apply slicing to extract the desired portion (last two columns)
result = arr[:, -2:]

# Print the result
print(result)
```

Explanation of Code:

- `import numpy as np`: We import the numpy library to work with numpy arrays.
- `arr = np.array([[1, 2, 3, 4, 11], [100, 55, 80, 33, 22]])`: This converts the list of lists into a numpy array.
- `arr[:, -2:]`: This slices the array to extract all rows and the last two columns.
- `print(result)`: Finally, we print the result, which will display the last two columns for each row.

Output:

```
[[ 4 11]
 [33 22]]
```

Summary

1. We started with a list of lists and converted it into a numpy array.
2. We applied numpy array slicing (`[:, -2:]`) to select all rows and the last two columns.

3. The result is the last two elements of each row printed as a 2x2 array.

Numpy's powerful slicing capabilities allow us to easily extract subarrays, making it an essential tool for data manipulation and numerical computation.