Kalinga University

Faculty of Computer Science & Information Technology

Course- BCAAIML
Subject- R Programming
Subject Code – BCAAIML401 **Sem- IV**

Unit 2

Creating matrices – Matrix Operations – Applying Functions to Matrix Rows and Columns – Adding and deleting rows and columns - Vector/Matrix Distinction – Avoiding Dimension Reduction – Higher Dimensional arrays – lists – Creating lists – General list operations – Accessing list components and values – applying functions to lists – recursive lists.
Introduction to R and Data Structures.

Creating Matrices

A matrix in Python is typically represented as a 2D list or a NumPy array.

Using Lists: You can create a matrix using nested lists in Python.

```
# 2x2 matrix (list of lists)
matrix = [[1, 2], [3, 4]]
```

Using NumPy: NumPy provides more efficient ways to create matrices, especially for larger datasets.

```
import numpy as np
# 3x3 matrix filled with zeros
matrix = np.zeros((3, 3))
# 4x5 matrix filled with random values between 0 and 1
matrix_random = np.random.rand(4, 5)
```

Matrix Operations

Matrix Addition/Subtraction: Matrices can be added or subtracted element-wise.

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
result_add = A + B  # Matrix addition
result_sub = A - B  # Matrix subtraction
```

Matrix Multiplication: Matrix multiplication can be performed using the np.dot() or @ operator in Python.

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
result_mul = np.dot(A, B)  # or A @ B
```

Element-wise Multiplication: If you want element-wise multiplication, use * operator.

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
result_elementwise = A * B
```

Transpose of a Matrix: You can transpose a matrix using .T in NumPy.

```
A = np.array([[1, 2], [3, 4]])
A_T = A.T  # Transpose of A
```

Applying Functions to Matrix Rows and Columns

Applying a Function to Each Element: You can apply a function (like squaring each element) to each element of a matrix.

```
A = np.array([[1, 2], [3, 4]])
result = np.vectorize(lambda x: x ** 2)(A)
```

Row and Column Operations: You can apply operations to entire rows or columns using axis in NumPy functions.

```
A = np.array([[1, 2], [3, 4]])
row_sum = np.sum(A, axis=1)  # Sum of each row
col_sum = np.sum(A, axis=0)  # Sum of each column
```

Adding and Deleting Rows and Columns

Adding a Row or Column: You can use np.vstack() to add rows or np.hstack() to add columns.

```
A = np.array([[1, 2], [3, 4]])
new_row = np.array([[5, 6]])
A_new_row = np.vstack([A, new_row])  # Add row

new_col = np.array([[7], [8]])
A_new_col = np.hstack([A, new_col])  # Add column
```

Deleting a Row or Column: Use np.delete() to remove rows or columns.

```
A = np.array([[1, 2], [3, 4], [5, 6]])
A_no_row = np.delete(A, 1, axis=0)  # Remove second row
A_no_col = np.delete(A, 1, axis=1)  # Remove second column
```

Vector/Matrix Distinction

Vector: A vector is a 1D array, either a row or a column matrix. It has a single dimension, e.g., 1xN or Nx1.

```
# Row vector
row_vector = np.array([1, 2, 3])
```

```
# Column vector
col_vector = np.array([[1], [2], [3]])
```

Matrix: A matrix is a 2D array with more than one row and column. It has two dimensions, e.g., MxN.

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

Avoiding Dimension Reduction

When slicing arrays, Python may reduce dimensions (e.g., extracting a single row or column might return a 1D array instead of a 2D matrix). To avoid this, use np.newaxis or reshape() to maintain the 2D structure.

```
A = np.array([[1, 2], [3, 4]])
row = A[0, np.newaxis, :]  # Keeps the row as a 2D array
```

Higher Dimensional Arrays

3D Arrays: A 3D array is essentially an array of matrices, which can be created using NumPy.

```
array_3D = np.random.rand(2, 3, 4)  # 2x3x4 3D array
```

Slicing Higher Dimensional Arrays: You can slice higher-dimensional arrays similarly to 2D arrays.

```
# Slice from a 3D array
sliced_array = array_3D[1, :, :]
```

Lists in Python

Creating Lists: Lists in Python are ordered collections of elements. You can create a list using square brackets.

```
numbers = [1, 2, 3, 4, 5]
fruits = ['apple', 'banana', 'cherry']
```

Accessing List Components: Use indexing to access elements of the list. Remember that Python indexing starts from 0.

```
first_element = numbers[0]  # 1
last_element = numbers[-1]  # 5
```

List Operations: Lists support various operations such as appending, removing, or modifying elements.

```
numbers.append(6)  # Add 6 to the end of the list
numbers.remove(3)  # Remove first occurrence of 3
numbers.insert(2, 10)  # Insert 10 at index 2
```

General List Operations

List Comprehensions: You can use list comprehensions to create new lists based on existing lists.

```
squares = [x**2 for x in numbers]  # [1, 4, 9, 16, 25]
```

Map Function: Apply a function to each element in a list using map().

```
squares = list(map(lambda x: x**2, numbers))
```

Filtering Lists: Use filter() to filter a list based on a condition.

```
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
```

Applying Functions to Lists

Applying Functions: You can apply functions to lists using map(), or directly use a loop to modify or process list elements.

```
numbers = [1, 2, 3, 4]
result = list(map(lambda x: x * 2, numbers))  # Apply lambda to double each element
```

Recursive Lists

Recursive Function for Sum: You can use recursion to process lists, such as calculating the sum of elements.

```
def recursive_sum(lst):
   if not lst:
      return 0
   return lst[0] + recursive_sum(lst[1:])

numbers = [1, 2, 3, 4]
print(recursive_sum(numbers))  # Output: 10
```

Recursive List Flattening: Use recursion to flatten nested lists.

```
def flatten(lst):
   result = []
   for item in lst:
      if isinstance(item, list):
         result.extend(flatten(item))  # Recursively flatten sublists
      else:
         result.append(item)
   return result
```

```
nested_list = [[1, 2], [3, [4, 5]]]
print(flatten(nested_list))  # Output: [1, 2, 3, 4, 5]
```