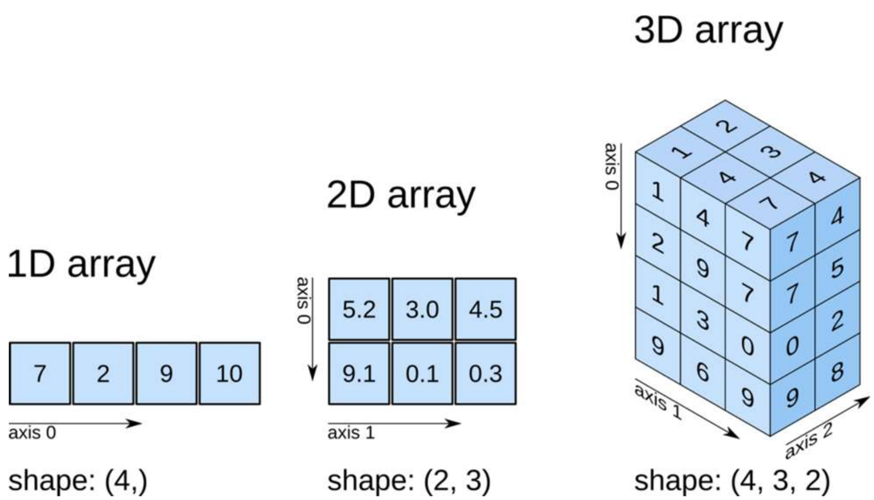
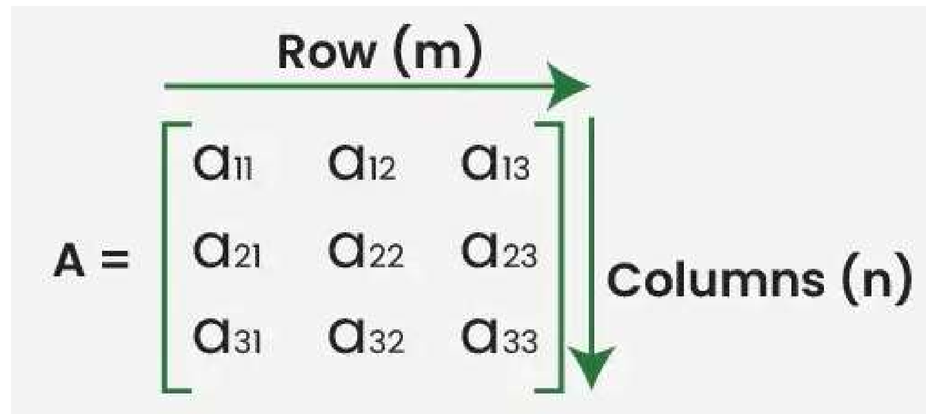
Cognitive Computing (UCS420)
NumPy – Numerical Python

NumPy

- **Fundamental package and library for scientific computing in Python**
- **Acts as basis for Pandas**
- **NumPy provides**
  - ***Multidimensional Array Object***
  - ***Derived Objects (such as masked arrays and matrices)***
  - ***Routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.***

Arrays vs Matrices





- **Can be n-dimensional (1-D,2-D, 3-D,…n-D)**
- **Designed for data storage and computation**

- **Always 2-D**
- **Specifically designed for linear algebra and its operations**
- **Deprecated in new versions of Numpy (use Array instead)**

Numpy Array (ND Array)

A = [“0” ,1 , “Two”, “3”, 4] ← **Python List**

0	1	2	3	4

A[0] :“0” **A[1] : 1** **A[2] : “Two”** **A[3] : “3”**

- **A Numpy array or ND array is similar to a list.**
- **It's usually fixed in size and each element is of the same type (here, integers)**

```
import numpy as np
a = np.array( [0, 1, 2, 3, 4] )
>> type(a)
>> numpy.ndarray
```

ND Array - Attributes

```
import numpy as np
a = np.array([0,1,2,3,4])

>> a.dtype          //Datatype of individual elements of array
>> dtype('int64')

>> a.size
>> 5
>> a.ndim : 1       // no. of array dimension or rank of array
>> a.shape : (5,)   //size of array in each dimension
```

ND Array – Indexing & Slicing

c=np.array([20,1,2,3, 4])
c:array([20,1,2,3, 4])

c[0]=100

c:array([100,1,2,3,4])

c[4]=0

c:array([100,1,2,3,0])

c:array([100,1,2,3,0])

0	1	2	3	4

d=c[1:4]

d:array([1, 2, 3])

c:array([100,1,2,3,0])

0	1	2	3	4

c[3:5]=300,400

c:array([100,1,2,300,400])

Numpy array vs Other Python Sequences

- Have a fixed size at creation
  - Changing the size of an ndarray will create a new array and delete the original
- Elements same datatype, and thus will be the same size in memory*
- Vectorized Operations
  - Executed more efficiently and with less code

*Exception – Array of objects

- Python list can grow dynamically
- Elements of different data type
- Looping overhead
  - Slower because each operation is interpreted and executed one-by-one

Example - Multiplying each element in a 1-D sequence with the corresponding element in another sequence of the same length

```
c = []
for i in range(len(a)):
    c.append(a[i]*b[i])
```

What if a and b each contain millions of numbers; we will pay the price for the inefficiencies of looping in Python.

Python is an interpreted language, which makes it slower for tasks like loops or element-wise operations due to:

1. ***Type checking for every element***
2. ***High-level abstractions***
3. ***Interpreted execution (not compiled)***

1. Dynamic Type Checking

- Type of a variable is determined at runtime.
- In operations like loops or element-wise calculations, Python performs type checks for every single element to ensure the operation is valid.

```
data = [1, 2, 3, 4, 5]
result = []
for i in data:
   result.append(i + 2)  # Python checks the type of `i` and `2` on every iteration
```

In contrast, C:

- C variables have fixed types (e.g., int, float) determined at compile time, so no type checks are needed during execution.

Why is it slow?

- For every addition (i + 2), Python:
  - Checks the type of i (e.g., is it an int, float, or str?).
  - Checks the type of 2.
  - Decides how to perform the operation (int + int, float + int, etc.).

2. High Level Abstractions (List, Tuple,..)

- Lists, dictionaries, and objects that are easy to use but have additional overhead.
- For example, a Python list can store elements of different types, requiring Python to handle memory allocation dynamically.

```
data = [1, 2.5, '3', 4, 5]
```

Why is it slow?

- Each element in the list is a Python object, not a raw integer.
  - Every element comes with additional metadata (e.g., type, size).
  - Python dynamically allocates and manages memory for the list and its elements.

In contrast, C:
Arrays in C are simple and store raw data without additional metadata or memory management overhead.

3. Interpreted Execution (Not Compiled)

- Translates Python code to machine code during execution, adding overhead
- Evaluates each line, figures out the meaning of data, i, *, and append, and executes the operations.
- This is repeated for each iteration of the loop.

```
data = [1, 2, 3, 4, 5]
result = []
for i in data:
    result.append(i * 2) # Each iteration involves interpretation
```

In contrast, C:
Compiled languages like C translate code into machine instructions ahead of time. During execution, the CPU directly executes those instructions without interpreting the code line by line.

Numpy:

Gives us the best of both worlds: element-by-element operations are the "default mode" when an ndarray is involved, but the element-by-element operation is speedily executed by pre-compiled C code.

In Python:

```python
c = []
for i in range(len(a)):
    c.append(a[i]*b[i])
```

```python
c = a * b
```

In C:

```c
for (i = 0; i < rows; i++) {
  c[i] = a[i]*b[i];
}
```

```c
for (i = 0; i < rows; i++) {
  for (j = 0; j < columns; j++) {
    c[i][j] = a[i][j]*b[i][j];
  }
}
```

Same operation

Near C-speed

Numpy has C backend

Why is NumPy fast?

- NumPy - Handles array operations in vectorized manner

$$c = a * b$$

<u>Advantages of Vectorized Code</u>

- Enhanced Speed
- Conciseness and Readability
- Fewer Bugs

- Process of applying operations **directly to entire arrays** or large blocks of data, rather than iterating over individual elements.
- Absence of any explicit looping, indexing, etc., in the code - these things are taking place, of course, just "behind the scenes" in **optimized, pre-compiled C code.**

Applications: *Machine Learning & Data Science, Simulations, Image Processing etc.*

Comparison of Python, NumPy, and C

Task	Pure Python	NumPy (Vectorized)	C
Looping over arrays	Slow due to Python loops	Very fast due to C backend	Fastest (manual optimization possible)
Matrix operations	Slow and complex	Near-C speed	Fast
Memory efficiency	High memory usage	Optimized for arrays	Most efficient

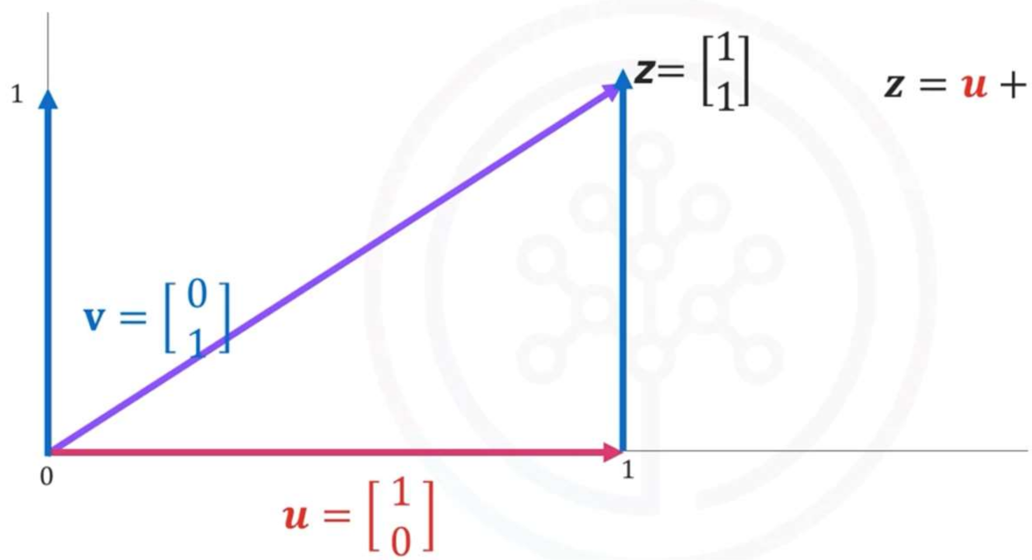
Basic Operations

Operations are usually computationally faster and require less memory in NumPy compared to regular Python

Why ? - Vectorization

Vector Addition

$$\boldsymbol{u} = \begin{bmatrix} \boxed{1} \\ 0 \end{bmatrix} \quad \boldsymbol{v} = \begin{bmatrix} \boxed{0} \\ 1 \end{bmatrix} \qquad \boldsymbol{z} = \boldsymbol{u} + \boldsymbol{v} = \begin{bmatrix} 1+0 \\ 0+1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$



$$\boldsymbol{z} = \boldsymbol{u} + \boldsymbol{v} = \begin{bmatrix} 1 \\ \boxed{1+0} \end{bmatrix}$$

Vector Addition

```
u=np.array([1,0])
v=np.array([0,1])

z=u+v

z:array([1, 1])
```

```
u=[1,0]

v=[0,1]

z=[ ]

for n, m in zip(u,v):
z.append(n+m)
```

Vector Subtraction

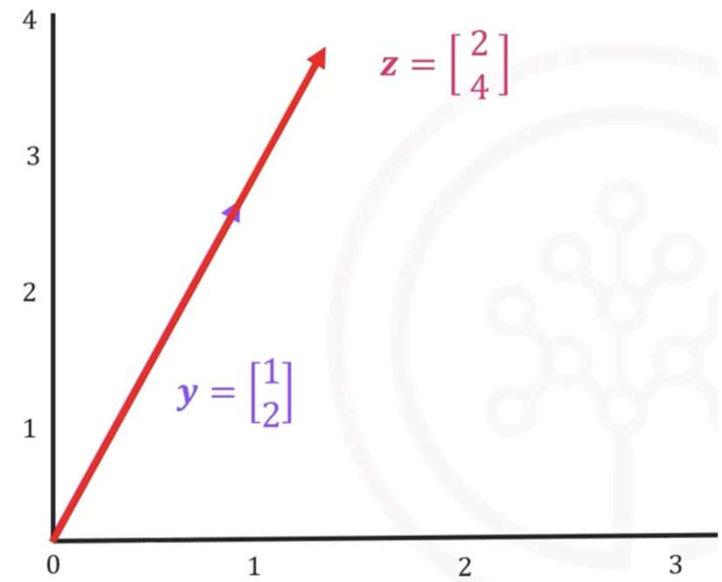
```
u=np.array([1,0])
v=np.array([0,1])

z=u-v

z:array([1, -1])
```

```
u=[1,0]

v=[0,1]

z=[ ]

for n, m in zip(u,v):
z.append(n-m)
```

Array Multiplication with Scalar

$$\boldsymbol{y} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

$$\boldsymbol{z} = 2\boldsymbol{y} = \begin{bmatrix} 2(1) \\ 2(2) \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$



Array Multiplication with Scalar

```
y=np.array([1,2])

z=2*y

z:array([2,4])
```

```
y=[1,2]

z=[ ]

for n in y:
z.append(2*n)
```