University of Sulaimani
College of Science
Computer Department
4th Stage

Data Science Management

Mathematics and Statistics for Data Science

Class 2 and 3

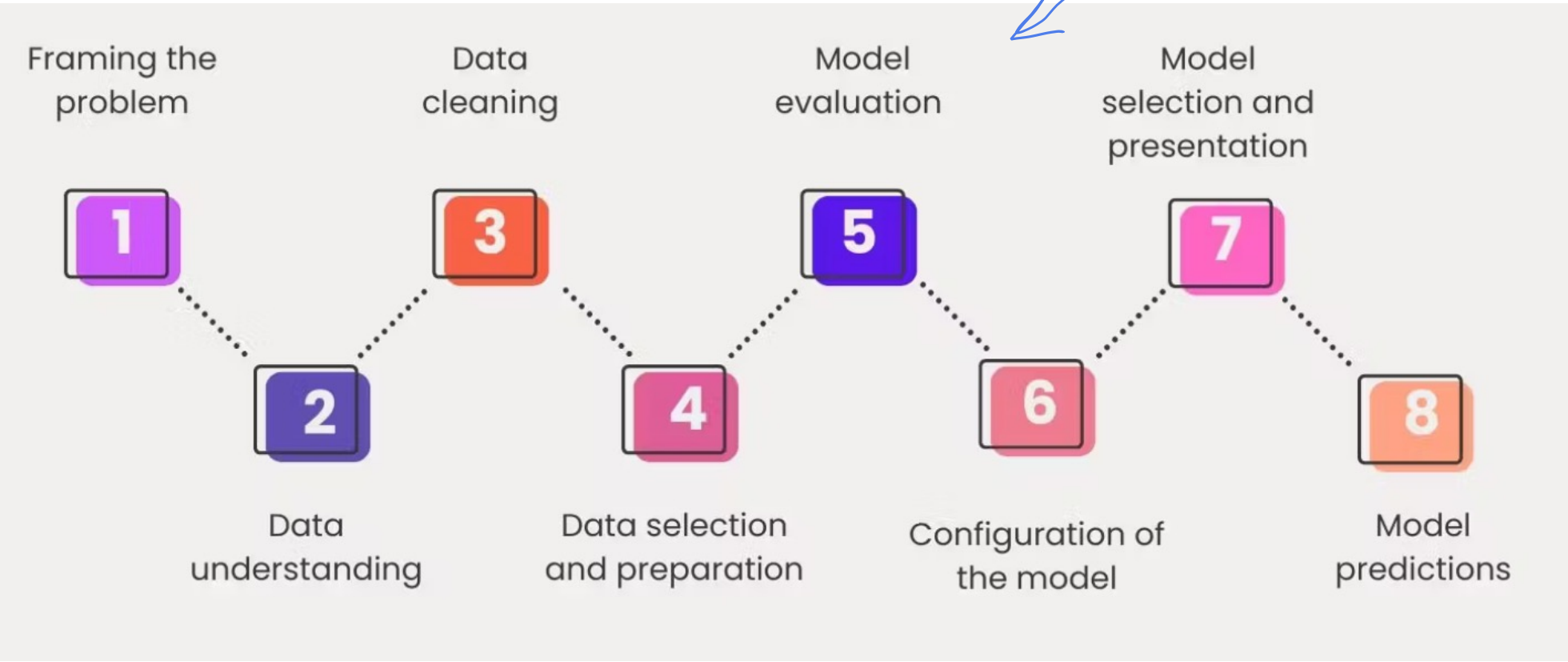
Theoretical and practical lectures

Assist. Prof. Dr. Miran Taha Abdullah
2025-2026

Class agenda

- **Introduction to Mathematics for Data Science**
    - Importance of mathematical concepts in analyzing and modeling data
- **Statistics for Data Science**
    - **Role of Statistics in Data Science**
        - Descriptive vs Inferential Statistics
        - Importance of statistical analysis in drawing insights from data
- **Vectors, Arrays, and Matrices**
    - **Introduction to Vectors and Arrays**

A Data Scientist must find **patterns** within the **data**. **Before find the patterns**, Data Scientist must organize the data in a **standard format**.



why ?

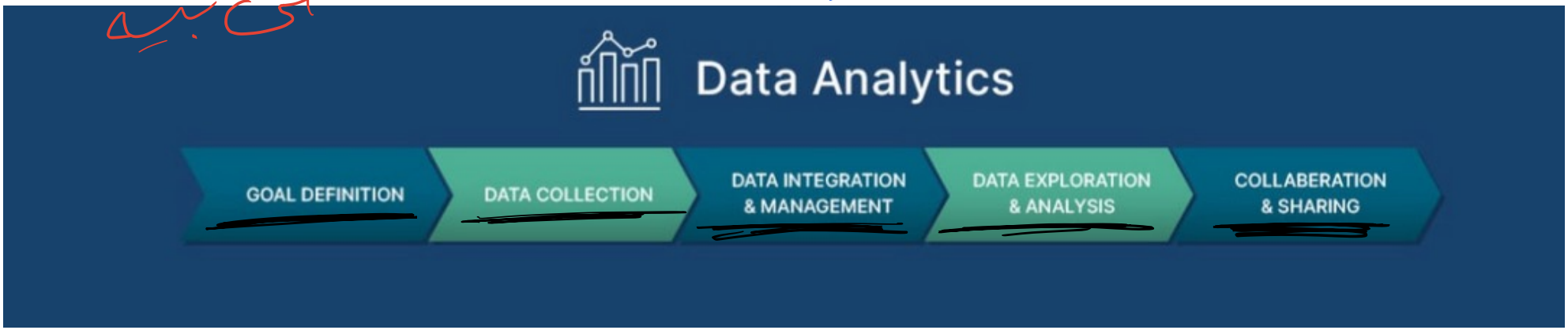
This step (**standard format**) is crucial because raw or unstructured data can introduce noise and inconsistencies that hinder analysis

- How do mathematics and statistics contribute to the field of data science?
- How can statistics and mathematical functions be used to make predictions in data science?
- What are some examples of methods or algorithms that rely on these techniques?

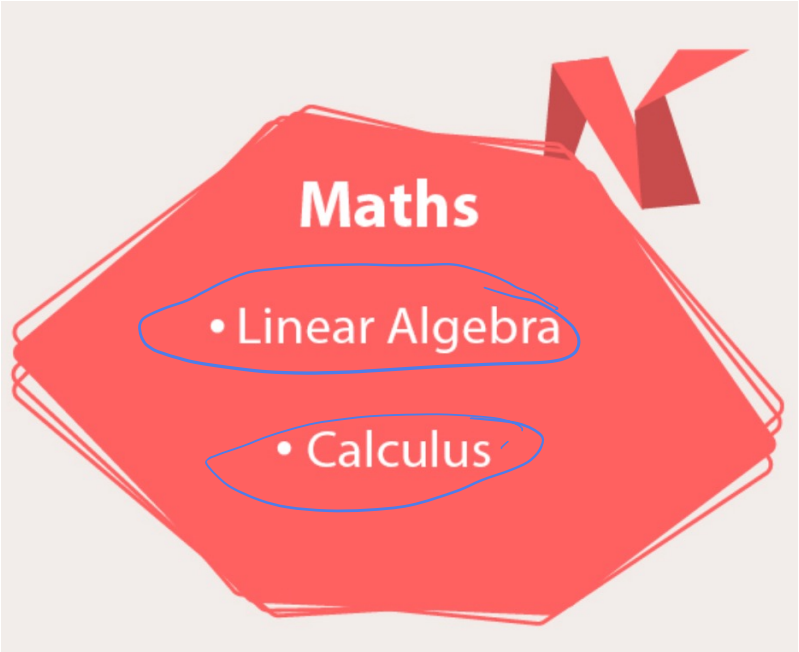
مهم يأتي في الامتحان ما الفرق بين

الفرق هو كل واحد كنده مودل خاص بيه

كلهما يجمع المعلومات ويعالجها لكن الاختلاف في الفانكشن المستخدمة.



Mathematics and **Statistics** are two of the most important concepts of data science. Data Science revolves around these two fields and draws their concepts to operate on the data.



1) Mathematics for Data Science

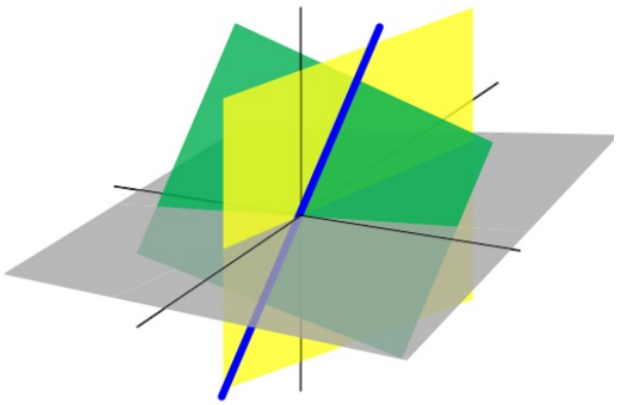
Mathematics has created an impact on every discipline. The magnitude of the usage of mathematics varies according to the disciplines.
There are two main components of mathematics that contribute to Data Science namely – ***Linear Algebra and Calculus.***

1.1. Linear Algebra

Linear Algebra is designed to solve problems of linear equations. These equations can sometimes contain higher dimension variables.
Linear algebra that deals with vectors and matrices and, more generally, with vector spaces and linear transformations.

Example

$$a_1 x_1 + \cdots + a_n x_n = b,$$



(optimaze) و (Index) السرعة و organize كلمة matrix في لماذا نضع البيانات

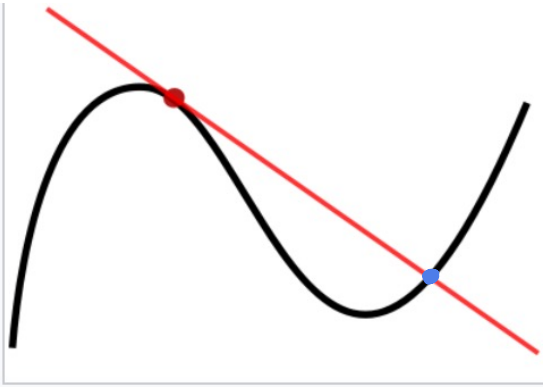
matrix و vecters ما الفرق بين matrix و vecters و loop ؟

1.2 Calculus

Another important requirement of Maths for Data Science is calculus. Calculus is used essentially in **optimization techniques**.

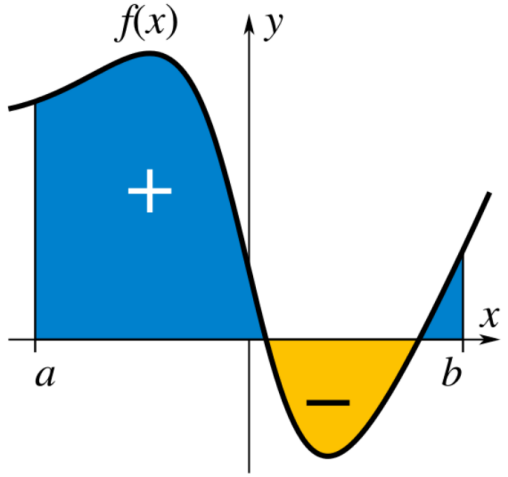
حساب التفاضل و التكامل

1.2.1 Differential Calculus



Differential Calculus studies the rate at which the quantities change. Derivates are most widely used for finding the maxima and minima of the functions. Derivates are used in optimization techniques where we have to find the minima in order to minimize the error function.

1.2.2 Integral Calculus



Integral Calculus is the mathematical study of the accumulation of quantities and for finding the area under the curve. Integrals are further divided into definite integrals and indefinite integrals. Integration is most widely used in computing probability density functions and variance of the random variable.

Example on linear Algebra and Calculas

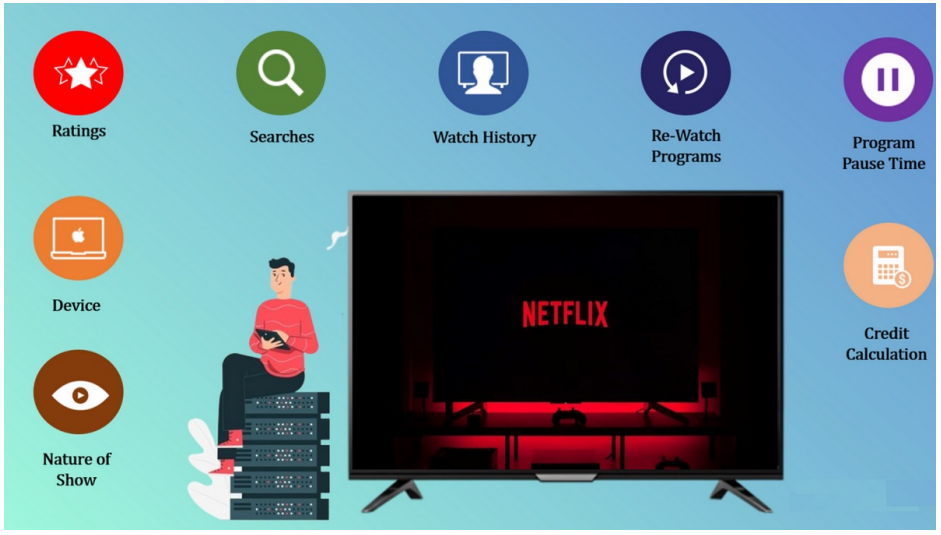
In recommender systems, especially in platforms like Netflix or Amazon, linear algebra is heavily used for **matrix factorization** techniques to *predict user preferences.*

Problem: Given a user-item interaction matrix (e.g., users and movies), where each entry represents how much a user likes a movie (rating), the goal is to predict missing values (i.e., unrated movies) based on existing ratings.

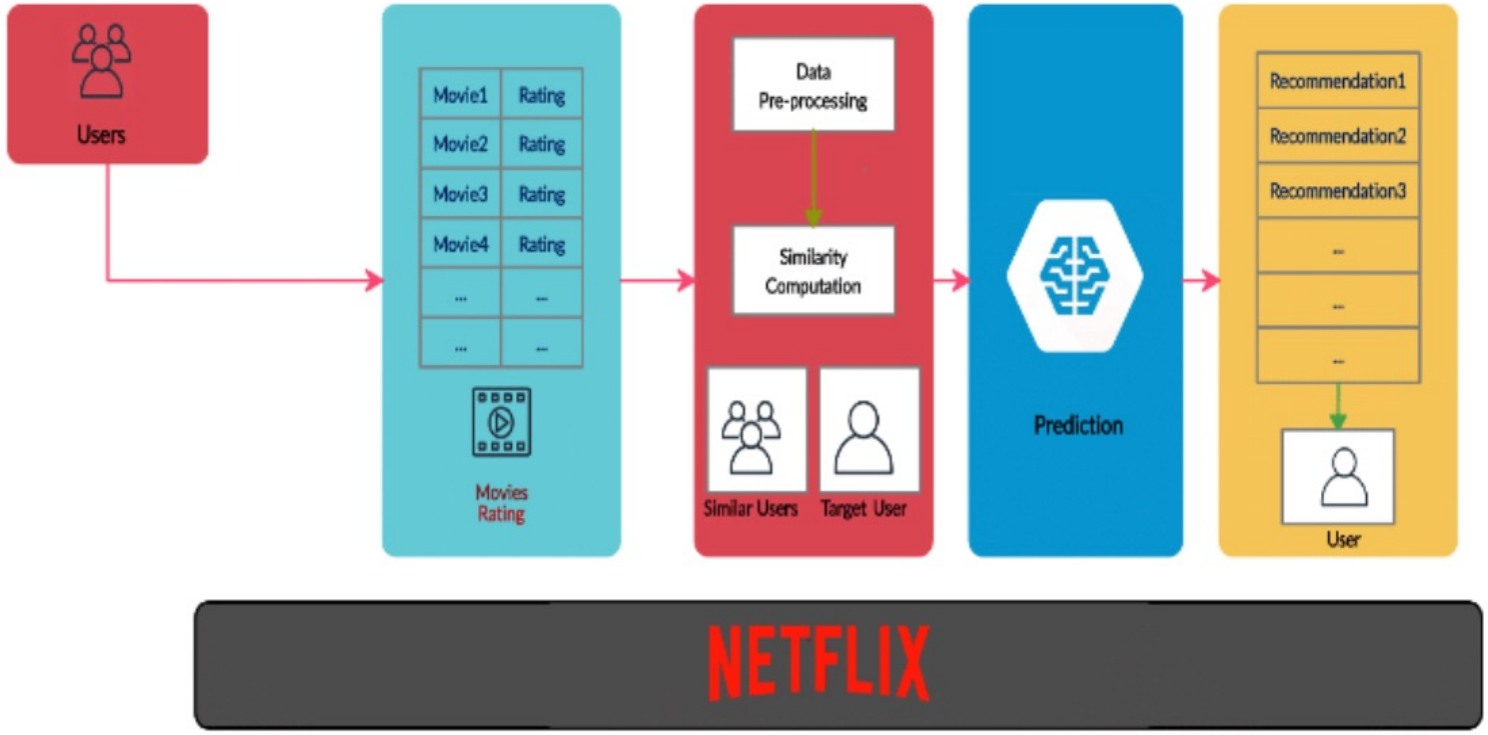
Rating Matrix $R \approx U \cdot V^T$

U (user matrix): This represents characteristics of the users, like preferences or tastes.

V (item matrix): This represents characteristics of the items (movies) based on how users rate them.



if User A has rated 5 movies but hasn't rated a particular movie, the recommender system will predict what their rating for that movie might be.



In a recommender system, **matrix factorization** is used to predict missing ratings in a user-item matrix. By breaking the matrix into two smaller matrices that represent users' preferences and items' features, the system can efficiently predict what movies a user might like

Calculus plays a critical role in optimizing the model by minimizing the error between the predicted values and actual outcomes. This process is done using **gradient descent**.

Problem: You want to fit a line to data points to predict a continuous target variable (e.g., predicting house prices based on features like *square footage, number of rooms, etc.*).
The objective is to find the best-fitting line that minimizes the difference between the **predicted and actual values.**

$$\text{Price} = m \times \text{Size} + b$$

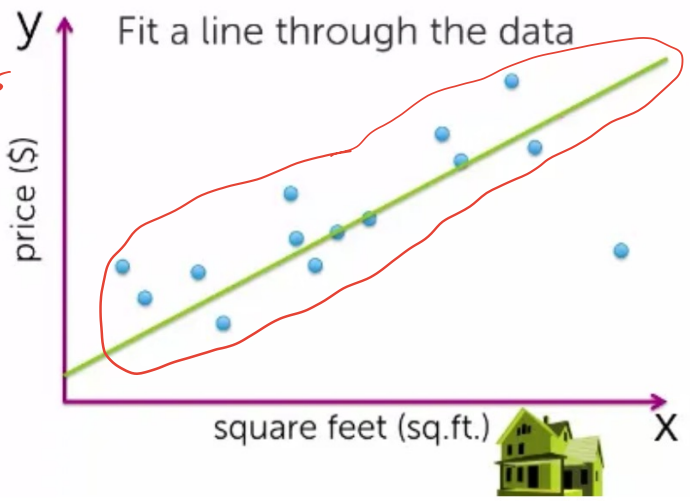
Price is the price of the house.

Size is the size of the house (in square feet).

m is the **slope** of the line (how much the price changes with each square foot).

b is the **intercept** (the price of the house when the size is 0).

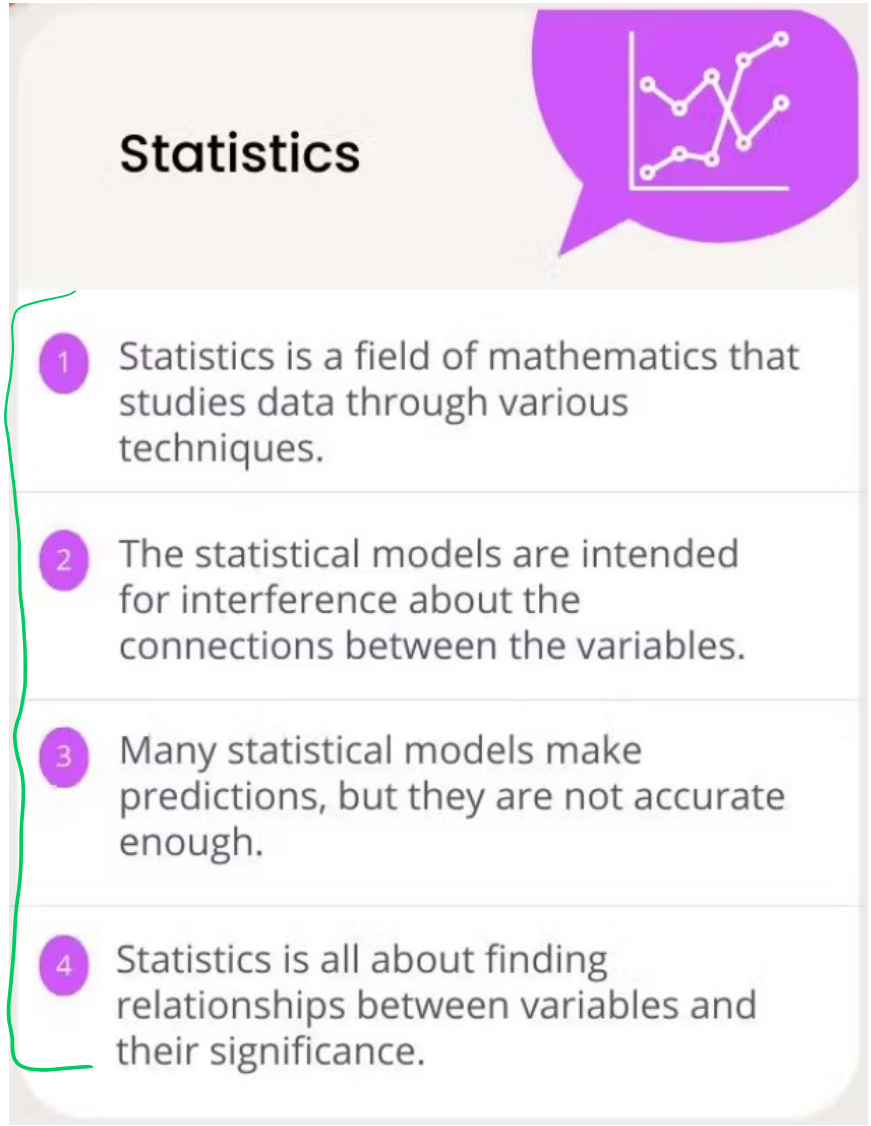
يسألك بالأمتحان
كيشنو العلاقة
بهذا الرسم



2) Statistic for Data Science

Statistics is a set of mathematical methods and tools that enable us to answer important questions about data.

Statistics are used in all scientific disciplines such as the physical and social sciences, as well as in business, the humanities, government, and manufacturing.



Statistic is divided into two categories:

2.1. Descriptive Statistics - this offers methods to summarise data by transforming **raw observations** into **meaningful information** that is easy to **interpret** and **share**.

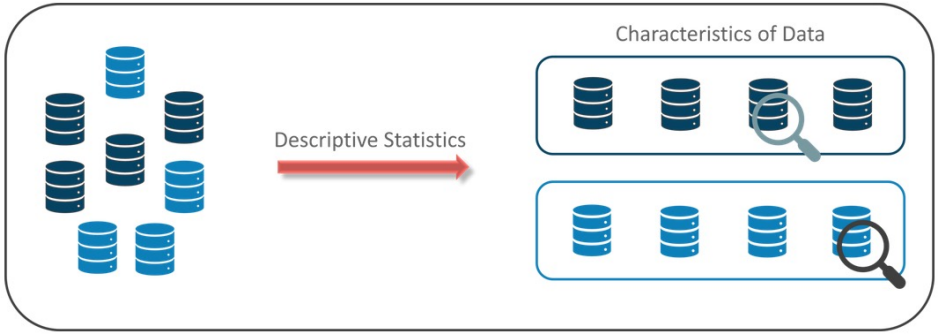
Techniques : **Normal Distribution, Central Tendency, Skewness & Kurtosis, Variability**

2.2 Inferential Statistics - this offers methods to **study experiments** done on small samples of data and chalk out (Apply) the inferences (conclusion) to the entire population (entire domain).

Techniques : **Central Limit Theorem, Hypothesis Testing, ANOVA, Qualitative Data Analysis**

Descriptive Statistics

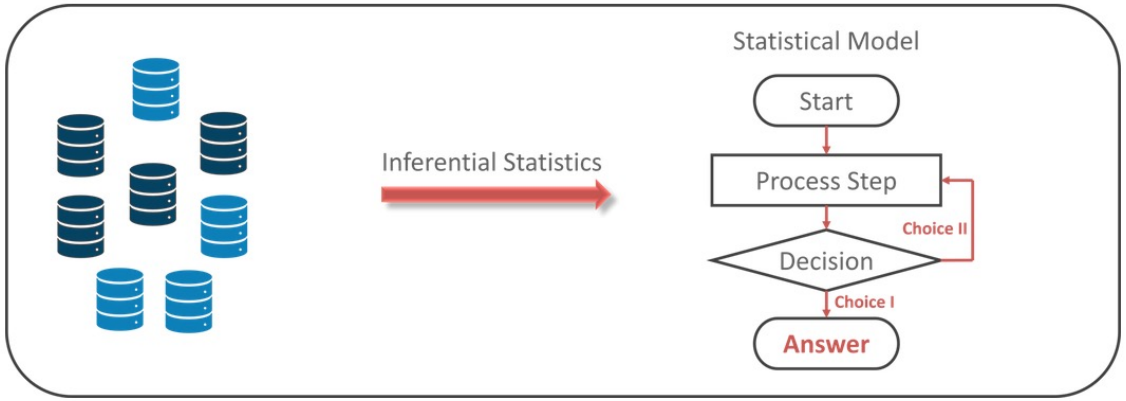
Descriptive Statistics uses the data to provide descriptions of the population, either through numerical calculations or graphs or tables. Descriptive Statistics helps organize data and focuses on the characteristics of data providing parameters. Descriptive statistics such as mean, median, and mode.

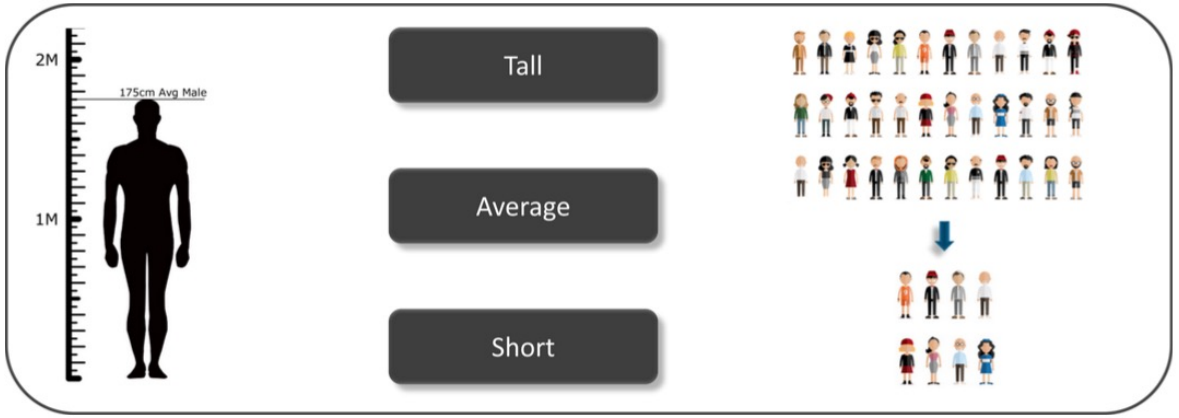


Example: Suppose you want to study the average height of students in a classroom, in descriptive statistics you would record the heights of all students in the class and then you would find out the maximum, minimum and average height of the class.

Inferential Statistics

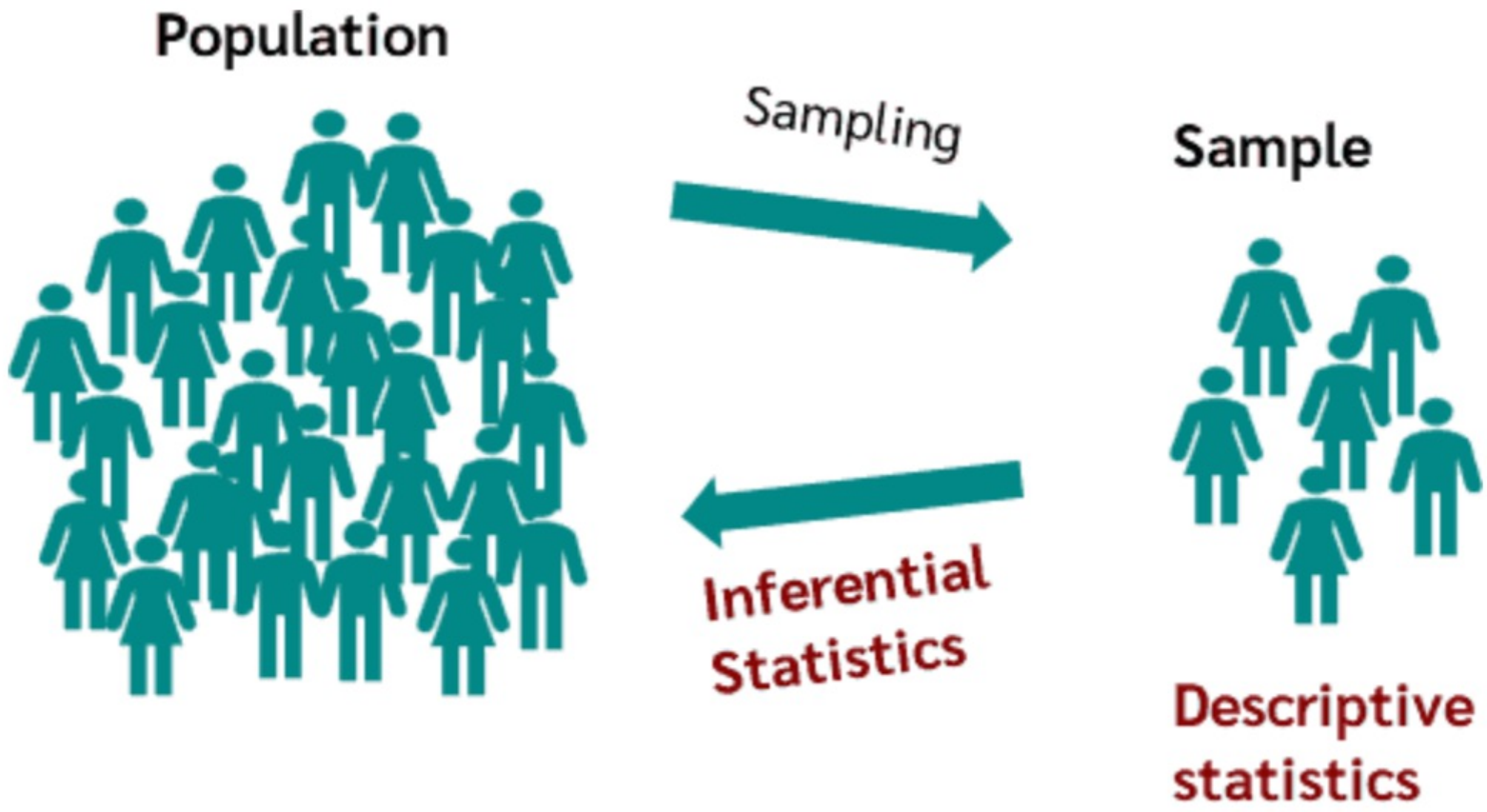
Inferential Statistics makes inferences and predictions about a population based on a sample of data taken from the population in question. Inferential statistics generalizes a large data set and applies probability to arrive at a conclusion. It allows you to infer parameters of the population based on sample stats and build models on it.





Example:

So, if we consider the same example of finding the average height of students in a class, in Inferential Statistics, you will take a sample set of the class, which is basically a few people from the entire class. You already have had grouped the class into tall, average and short. In this method, you basically build a statistical model and expand it for the entire population in the class.

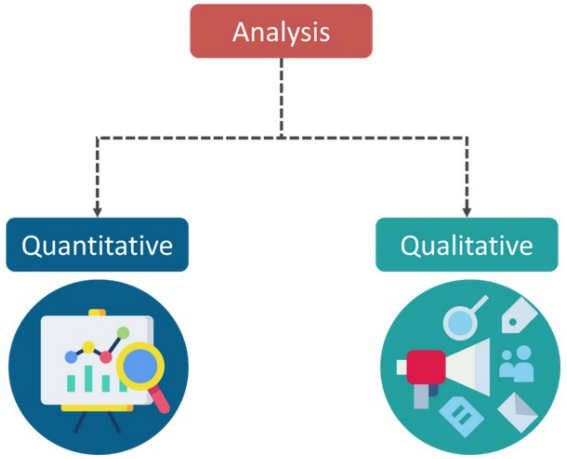

Population
Sampling
Sample
Inferential Statistics
Descriptive statistics

Types Of Analysis

An analysis of any event can be done in one of two ways

1. **Quantitative Analysis:** Quantitative Analysis or the Statistical Analysis is the science of collecting and interpreting data with numbers and graphs to identify patterns and trends.

2. **Qualitative Analysis:** Qualitative or Non-Statistical Analysis gives generic information and uses text, sound and other forms of media to do so.



For example, if I want a purchase a coffee from Starbucks, it is available in Short, Tall and Grande. This is an example of Qualitative Analysis. But if a store sells 70 regular coffees a week, it is Quantitative Analysis because we have a number representing the coffees sold per week.

Descriptive Statistics

1) Mean, Median, Mode
2) IQR, percentiles
3) Std deviation and Variance
4) Normal Distribution
5) Z-statistics and T-statistics
6) correlation and linear regression

Inferential Statistics:

1) Sampling distributions
2) confidence interval
3) chi-square test
4) Advanced regression
5) ANOVA

Vectors, Matrices, and Multidimensional Arrays

Class outline

1. **Importing the Modules**
2. The NumPy Array Object
3. Creating Arrays
4. Indexing and Slicing
5. Reshaping and Resizing
6. Vectorized Expressions
7. Matrix and Vector Operations

What is Vector?

- Vectors, matrices, and arrays of higher dimensions are essential tools in numerical computing.
- When a computation must be repeated for a set of input values, it is natural and advantageous to represent the data as arrays and the computation in terms of array operations.
- Computations that are formulated this way are said to be vectorized.
- Vectorized computing eliminates the need for many explicit loops over the array elements by applying batch operations on the array data.
- The result is concise and more maintainable code, and it enables delegating the implementation of array operations to more efficient low-level libraries.
- Vectorized computations can therefore be significantly faster than sequential element-by-element computations.

1. Importing the Modules

The Python programming language comes with a variety of built-in functions. Among these are several common functions, including:

print() which prints expressions out
abs() which returns the absolute value of a number
int() which converts another data type to an integer
len() which returns the length of a sequence or collection

Modules are Python .py files that consist of Python code. Any Python file can be referenced as a module. A Python file called hello.py has the module name of hello that can be imported into other Python files or used on the Python command line interpreter.

Modules can define functions, classes, and variables that you can reference in other Python .py files or via the Python command line interpret.

Importing the Modules:

In order to use the NumPy library, we need to import it in our program. By convention, the numPy module imported under the alias np.

Example : How To Import Modules in Python

```
# Python Module example

def add(a, b):
   """This program adds two
   numbers and return the result"""

   result = a + b
   return result
```

save it as **example.py**

```
>>> import example
```

```
>>> example.add(4,5.5)
9.5
```

```
# import all names from the standard module math

from math import *
print("The value of pi is", pi)
```

Example : How To Import Modules in Python

```
import random
import math


for i in range(5):
    print(random.randint(1, 25))

print(math.pi)
```

```
import [module] as [another_name]
```

```
import random


for i in range(10):
    print(random.randint(1, 25))
```

```
import math as m


print(m.pi)
print(m.e)
```

Class outline

1. Importing the Modules
2. **The NumPy Array Object**
3. **Creating Arrays**
4. Indexing and Slicing
5. Reshaping and Resizing
6. Vectorized Expressions
7. Matrix and Vector Operations

2. The NumPy Array Object

3. Creating Arrays

The NumPy library provides data structures for representing a rich variety of arrays and methods and functions for operating on such arrays.

NumPy provides the numerical backend for nearly every scientific or technical library for Python. It is therefore a very important part of the scientific Python ecosystem

Array creation

First, we must import the module "numpy"

```
import numpy as np
```

np is the alias used for accessing to the routines of the package 'numpy '.

Converting Python array_like objects (e.g. list)

```
a = np.array([1.2,2.5,3.2,1.8])
```

[ ] is a list of values (float)

Information about the structure

```
#object type
print(type(a))  #<class 'numpy.ndarray'>
#data type
print(a.dtype)  #float64
#number of dimensions
print(a.ndim)  #1 (we have 2 if it is a matrix, etc.)
#number of rows and columns
print(a.shape)  #(4,) → tuple! 4 elements for the 1st  dim (n°0)
#total number of elements
print(a.size)  #4, nb.rows  x nb.columns if a matrix
```

Attributes of the ndarray Class

Attribute	Description
`Shape`	A tuple that contains the number of elements (i.e., the length) for each dimension (axis) of the array.
`Size`	The total number elements in the array.
`Ndim`	Number of dimensions (axes).
`nbytes`	Number of bytes used to store the data.
`dtype`	The data type of the elements in the array.

Numerical Data Types Available in NumPy

dtype	Variants	Description
`int`	`int8, int16, int32, int64`	Integers
`uint`	`uint8, uint16, uint32, uint64`	Unsigned (nonnegative) integers
`bool`	`Bool`	Boolean (True or False)
`float`	`float16, float32, float64, float128`	Floating-point numbers
`complex`	`complex64, complex128, complex256`	Complex-valued floating-point numbers

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type (same as C `long`; normally either `int64` or `int32`)
`intc`	Identical to C `int` (normally `int32` or `int64`)
`intp`	Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`)
`int8`	Byte (–128 to 127)
`int16`	Integer (–32768 to 32767)
`int32`	Integer (–2147483648 to 2147483647)
`int64`	Integer (–9223372036854775808 to 9223372036854775807)
`uint8`	Unsigned integer (0 to 255)
`uint16`	Unsigned integer (0 to 65535)
`uint32`	Unsigned integer (0 to 4294967295)
`uint64`	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64`
`float16`	Half-precision float: sign bit, 5 bits exponent, 10 bits mantissa
`float32`	Single-precision float: sign bit, 8 bits exponent, 23 bits mantissa
`float64`	Double-precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Shorthand for `complex128`
`complex64`	Complex number, represented by two 32-bit floats
`complex128`	Complex number, represented by two 64-bit floats

Setting the data type

Specifying the data type can be implicit or explicit

```
#creating a vector – implicit typing
a = np.array([1,2,4])
print(a.dtype)  #int32

#creating a vector – explicit typing – preferable !
a = np.array([1,2,4],dtype=float)
print(a.dtype)  #float64
print(a)  #[1.  2.  4.]

#a vector of Boolean values is possible
b = np.array([True,False,True,True], dtype=bool)
print(b) #[True  False  True True]
```

Creating an array with objects of non-standard type is possible

```
# the array value may be an object
a = np.array([{"Toto":(45,2000)},{"Tata":(34,1500)}])
print(a.dtype) #object
```

NumPy Functions for Generating Arrays

Function Name	Type of Array
`np.array`	Creates an array for which the elements are given by an array-like object, which, for example, can be a (nested) Python list, a tuple, an iterable sequence, or another `ndarray` instance.
`np.zeros`	Creates an array with the specified dimensions and data type that is filled with zeros.
`np.ones`	Creates an array with the specified dimensions and data type that is filled with ones.
`np.diag`	Creates a diagonal array with specified values along the diagonal and zeros elsewhere.
`np.arange`	Creates an array with evenly spaced values between the specified start, end, and increment values.
`np.linspace`	Creates an array with evenly spaced values between specified start and end values, using a specified number of elements.

```
>>> x = np.arange(9).reshape((3,3))
>>> x
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```
>>> np.diag(x)
array([0, 4, 8])
>>> np.diag(x, k=1)
array([1, 5])
>>> np.diag(x, k=-1)
array([3, 7])
```

```
>>> np.diag(np.diag(x))
array([[0, 0, 0],
       [0, 4, 0],
       [0, 0, 8]])
```

`np.logspace`	Creates an array with values that are logarithmically spaced between the given start and end values.
`np.meshgrid`	Generates coordinate matrices (and higher-dimensional coordinate arrays) from one-dimensional coordinate vectors.
`np.fromfunction`	Creates an array and fills it with values specified by a given function, which is evaluated for each combination of indices for the given array size.
`np.fromfile`	Creates an array with the data from a binary (or text) file. NumPy also provides a corresponding function `np.tofile` with which NumPy arrays can be stored to disk and later read back using `np.fromfile`.
`np.genfromtxt`, `np.loadtxt`	Create an array from data read from a text file, for example, a comma-separated value (CSV) file. The function `np.genfromtxt` also supports data files with missing values.
`np.random.rand`	Generates an array with random numbers that are uniformly distributed between 0 and 1. Other types of distributions are also available in the `np.random` module.

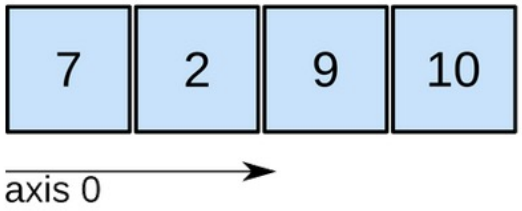
```
import numpy as np

arr = np.array(42)

print(arr)
```

```
import numpy as np

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

print(arr)
```

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr)
```

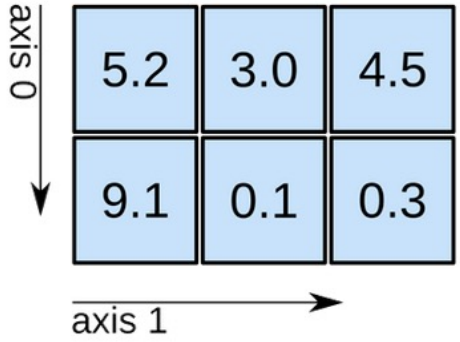
```
import numpy as np

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

print(arr)
```

1D array



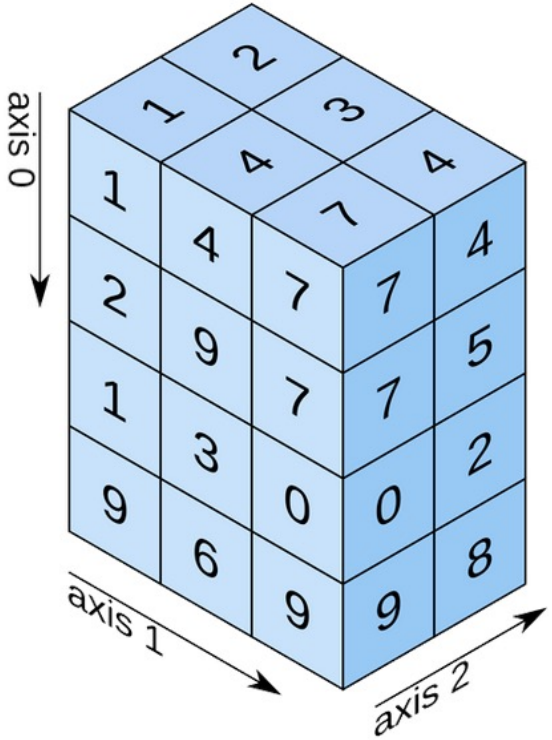
shape: (4,)

2D array



shape: (2, 3)

3D array



shape: (4, 3, 2)

Creating sequence of numbers

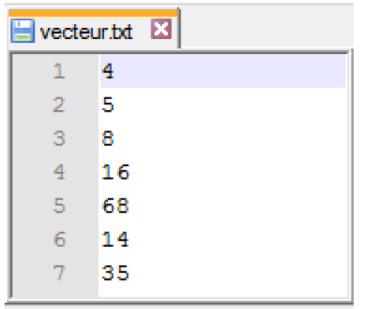
```
#evenly spaced values within a given interval (step = 1 here)
a = np.arange(start=0,stop=10)
print(a) #[0  1  2  3  4  5  6  7  8  9], the last value is excluded

#specifying the step property
a = np.arange(start=0,stop=10,step=2)
print(a) #[0  2  4  6  8]

#evenly spaced value, specify the number of elements
a = np.linspace(start=0,stop=10,num=5)
print(a) #[0.  2.5  5.  7.5  10.], the last value is included here

#repeating 5 times the value 1 – number  of values = 5 (1 dimension)
a = np.ones(shape=5)
print(a) # [1.  1.  1.  1.  1.]

#repeating 5 times (1 dimension) the value 3.2
a = np.full(shape=5,fill_value=3.2)
print(a) #[3.2  3.2  3.2  3.2  3.2]
```

Loading a vector from a data file

The values can be stored in a text file (loadtxt for reading, savetxt for writing)

```
#loading from a text file
#we can set the type of the data
a = np.loadtxt("vecteur.txt",dtype=float)
print(a) #[4.  5.  8.  16.  68.  14.  35.]
```

Only 1 column here



Note: If necessary, we change the default directory with the function **chdir()** from the **os** module (that must be imported)

We can convert a Python sequence type in a "numpy" array

```
#lst is a list of values (float)
lst = [1.2,3.1,4.5]
print(type(lst)) #<class 'list'>
#converting the list
a = np.asarray(lst,dtype=float)
print(type(a)) #<class 'numpy.ndarray'>
print(a) #[1.2  3.1  4.5]
```

Adding and removing elements

Add a value in last position

```
#a is a vector
a = np.array([1.2,2.5,3.2,1.8])
#append the value 10 into the vector a
a = np.append(a,10)
print(a) #[1.2  2.5  3.2  1.8  10.]
```

Remove a value from its index

```
#remove the value n°2
b = np.delete(a,2) #a range of indices can be used
print(b) #[1.2  2.5  1.8  10.]
```

Modify the size of a vector

```
a = np.array([1,2,3])
#adding two cells
#fills zero for the new cell
a.resize(new_shape=5)
print(a) #[1  2  3  0  0]
```

Concatenation of vectors

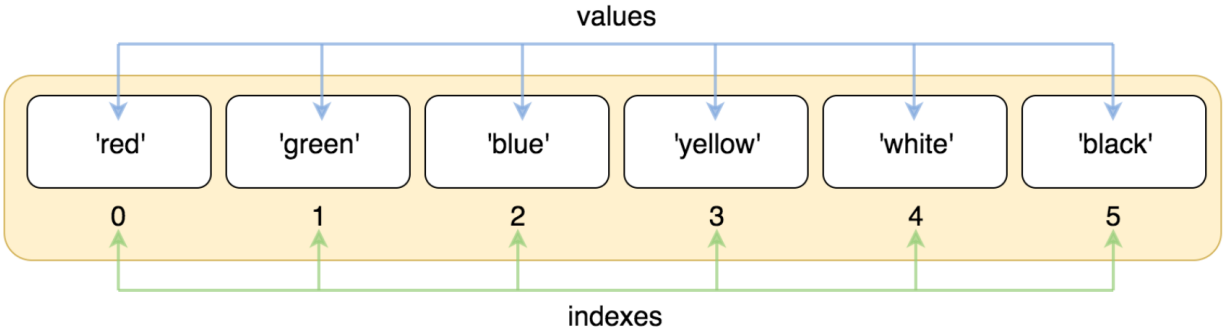
```
#concatenate 2 vectors
x = np.array([1,2,5,6])
y = np.array([2,1,7,4])
z = np.append(x,y)
print(z) #[1  2  5  6  2  1  7  4]
```

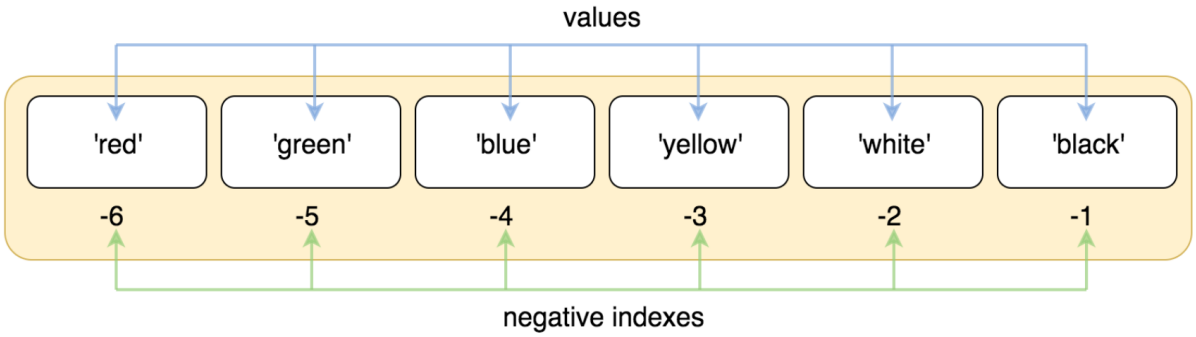
Class outline

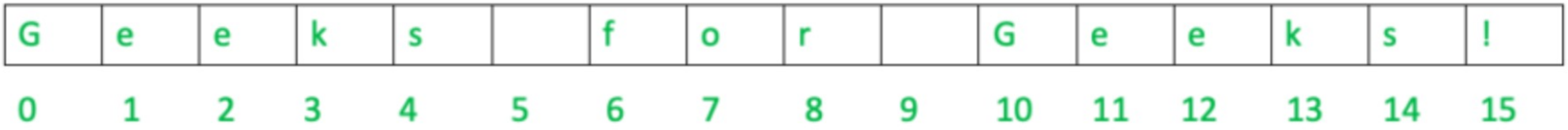
1. Importing the Modules
2. The NumPy Array Object
3. Creating Arrays
4. **Indexing and Slicing**
5. Reshaping and Resizing
6. Vectorized Expressions
7. Matrix and Vector Operations

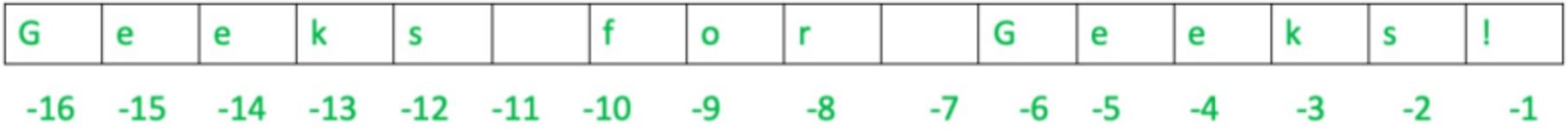
4. Indexing and Slicing

- **Indexing** means referring to an element of an iterable by its position within the iterable. Each of a string's characters corresponds to an index number and each character can be accessed using their index number.

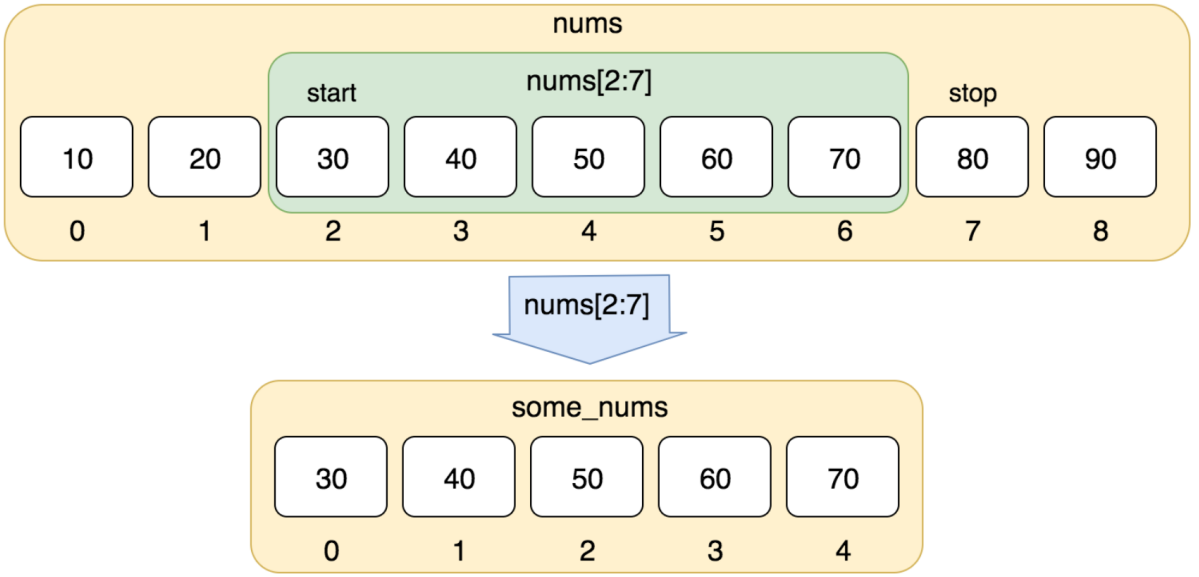





G e e k s f o r G e e k s !
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


G e e k s f o r G e e k s !
-16 -15 -14 -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1

Slicing in Python is a feature that enables accessing parts of sequence. In slicing string (or number), we create a substring (or sub-number)., which is essentially a string (or a number) that exists within another string (or number). We use slicing when we require a part of string (or number) and not the complete string.



Examples of Array Indexing and Slicing Expressions

Expression	Description
`a[m]`	Select element at index *m*, where *m* is an integer (start counting form 0).
`a[-m]`	Select the *n*th element from the end of the list, where *n* is an integer. The last element in the list is addressed as –1, the second to last element as –2, and so on.
`a[m:n]`	Select elements with index starting at *m* and ending at *n* – 1 (*m* and *n* are integers).
`a[:]` or `a[0:-1]`	Select all elements in the given axis.
`a[:n]`	Select elements starting with index 0 and going up to index *n* – 1 (integer).
`a[m:]` or `a[m:-1]`	Select elements starting with index *m* (integer) and going up to the last element in the array.
`a[m:n:p]`	Select elements with index *m* through *n* (exclusive), with increment *p*.
`a[::-1]`	Select all the elements, in reverse order.

Indexed access – v = np.array([1.2,7.4,4.2,8.5,6.3])

```
#printing all the values
print(v)
#or
print(v[:]) # note the role of :  ; here, from start to end

#indexed access - first value
print(v[0]) # 1.2 – the first index is 0 (zero)

#last value
print(v[v.size-1]) #6.3, v.size is okay because v is a vector

#contiguous indices
print(v[1:3]) # [7.4  4.2]

#extreme values, start to 3 (not included)
print(v[:3]) # [1.2  7.4  4.2]

#extreme values, 2 to end
print(v[2:]) # [4.2  8.5  6.3]

#negative indices
print(v[-1]) # 6.3, last value

#negative indices
print(v[-3:]) # [4.2  8.5  6.3], 3 last values
```

Note : Apart from singletons, the generated vectors are of type numpy.ndarray

Class outline

1. Importing the Modules
2. The NumPy Array Object
3. Creating Arrays
4. Indexing and Slicing
5. **Reshaping and Resizing**
6. Vectorized Expressions
7. Matrix and Vector Operations

5. Reshaping and Resizing

The reshape() function is used to give a new shape to an array without changing its data.

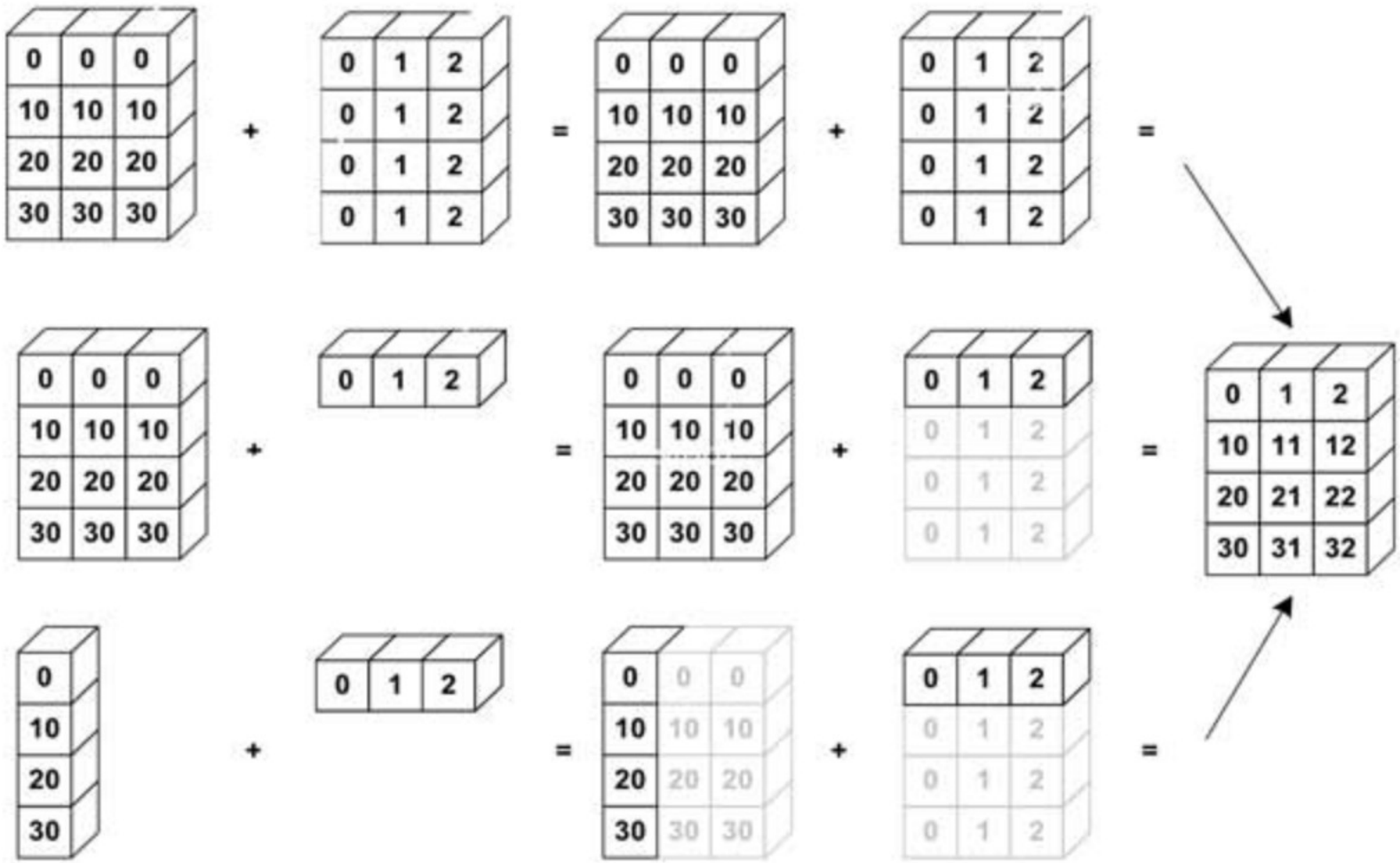
numpy.reshape()

New_Array = np.reshape(x,new_resize) → X= curent Array

The resize() function is used to create a new array with the specified shape. If the new array is larger than the original array, then the new array is filled with repeated copies of X.

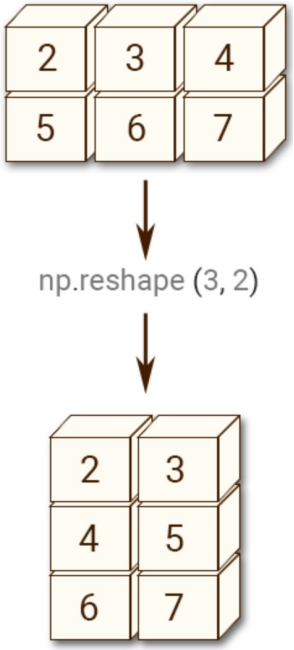
numpy.resize()

New_Array = np.resize(x,new_shape) → X= curent Array

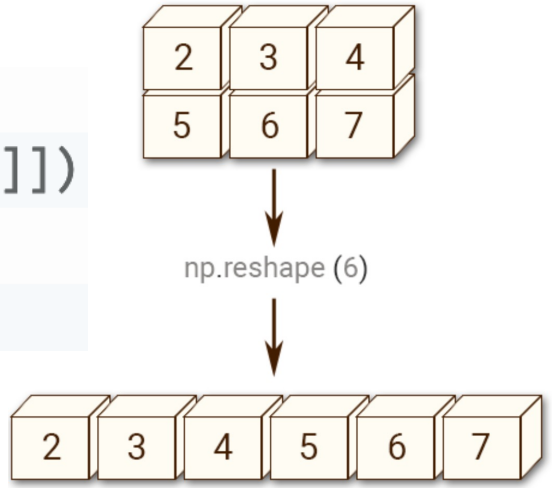

0 0 0
10 10 10
20 20 20
30 30 30
+
0 1 2
0 1 2
0 1 2
0 1 2
=
0 0 0
10 10 10
20 20 20
30 30 30
+
0 1 2
0 1 2
0 1 2
0 1 2
=
0 0 0
10 10 10
20 20 20
30 30 30
+
0 1 2
=
0 0 0
10 10 10
20 20 20
30 30 30
+
0 1 2
0 1 2
0 1 2
0 1 2
=
0 1 2
10 11 12
20 21 22
30 31 32
0
10
20
30
+
0 1 2
=
0 0 0
10 10 10
20 20 20
30 30 30
+
0 1 2
0 1 2
0 1 2
0 1 2
=

np.reshape()

```
>>> import numpy as np
>>> x = np.array([[2,3,4], [5,6,7]])
>> np.reshape(x, (3, 2))
array([[2, 3],
       [4, 5],
       [6, 7]])
```




```
>>> import numpy as np
>>> x = np.array([[2,3,4], [5,6,7]])
>>> np.reshape(x, 6)
array([2, 3, 4, 5, 6, 7])
```

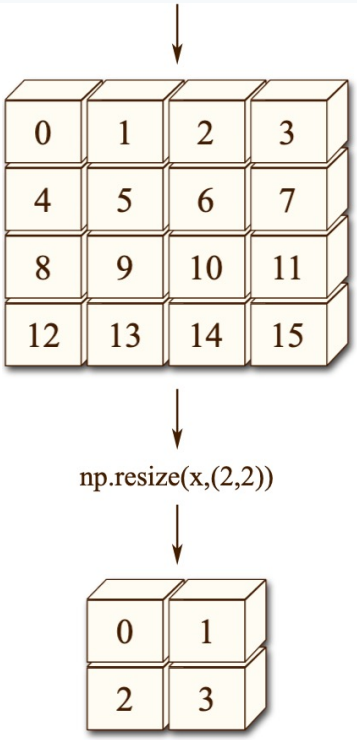


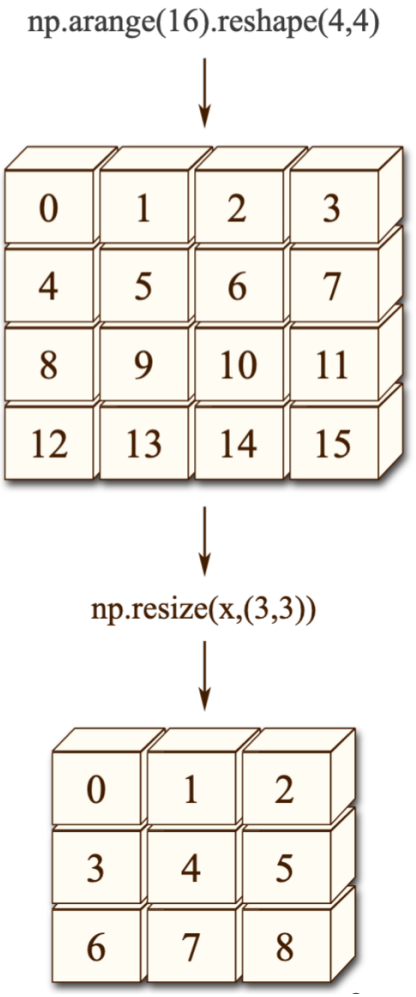
np.resize()

```
1  import numpy as np
2  x = np.arange(16).reshape(4,4)
3  print("Original arrays:")
4  print(x)
5  print("\nArray with size 2x2 from the said array:")
6  new_array1 = np.resize(x,(2,2))
7  print(new_array1)
```

```
Original arrays:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
```

```
Array with size 2x2 from the said array:
[[0 1]
 [2 3]]
```





How?

Function/Method	Description
`np.reshape`, `np.ndarray.reshape`	Reshape an N-dimensional array. The total number of elements must remain the same.
`np.ndarray.flatten`	Creates a copy of an N-dimensional array, and reinterpret it as a one-dimensional array (i.e., all dimensions are collapsed into one).
`np.ravel`, `np.ndarray.ravel`	Create a view (if possible, otherwise a copy) of an N-dimensional array in which it is interpreted as a one-dimensional array.
`np.squeeze`	Removes axes with length 1.
`np.expand_dims`, `np.newaxis`	Add a new axis (dimension) of length 1 to an array, where `np.newaxis` is used with array indexing.
`np.transpose`, `np.ndarray.transpose`, `np.ndarray.T`	Transpose the array. The transpose operation corresponds to reversing (or more generally, permuting) the axes of the array.
`np.hstack`	Stacks a list of arrays horizontally (along axis 1): for example, given a list of column vectors, appends the columns to form a matrix.
`np.vstack`	Stacks a list of arrays vertically (along axis 0): for example, given a list of row vectors, appends the rows to form a matrix.
`np.dstack`	Stacks arrays depth-wise (along axis 2).
`np.concatenate`	Creates a new array by appending arrays after each other, along a given axis.

Function/Method	Description
`np.resize`	Resizes an array. Creates a new copy of the original array, with the requested size. If necessary, the original array will be repeated to fill up the new array.
`np.append`	Appends an element to an array. Creates a new copy of the array.
`np.insert`	Inserts a new element at a given position. Creates a new copy of the array.
`np.delete`	Deletes an element at a given position. Creates a new copy of the array.

Student Workload (4 hours)

Exercise the different parameters of **NumPy**

Use this URL (https://www.javatpoint.com/numpy-array**)**

numpy.appned() in Python

1.import numpy as np

2.a=np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])

3.b=np.array([[11, 21, 31], [42, 52, 62], [73, 83, 93]])

4.c=np.append(a,b,axis=0)

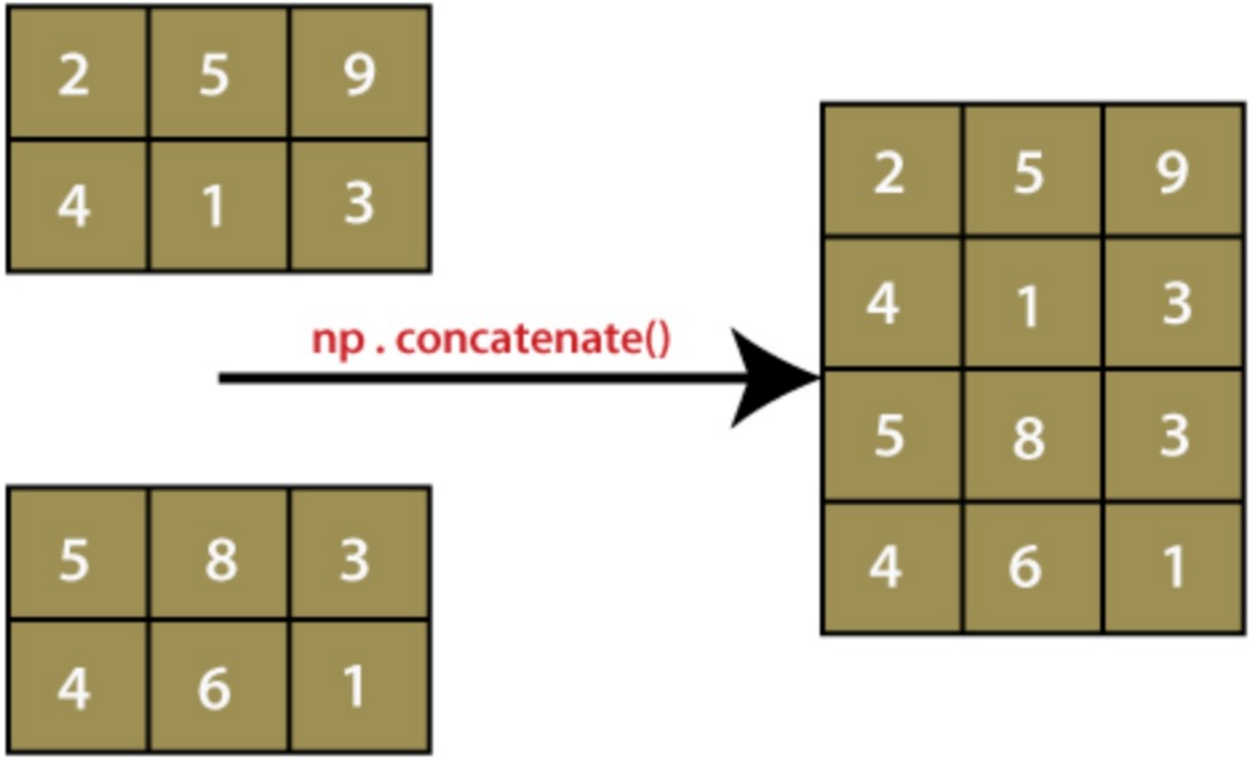
5.c

```
array([[ 10,  20,  30],
       [ 40,  50,  60],
       [ 70,  80,  90],
       [11, 21, 31],
       [42, 52, 62],
       [73, 83, 93]])
```

```
1.import numpy as np
2.a=np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])
3.b=np.array([[11, 21, 31], [42, 52, 62], [73, 83, 93]])
4.c=np.append(a,b,axis=1)
5.c
```

```
array([[ 10,  20,  30, 11, 21, 31],
       [ 40,  50,  60, 42, 52, 62],
       [ 70,  80,  90, 73, 83, 93]])
```

numpy.concatenate() in Python



1.import numpy as np

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

3.y=np.array([[12,30]])

4.z=np.concatenate((x,y), axis=0)

5.z

```
array([[ 1,  2],
       [ 3,  4],
       [12, 30]])
```

```
1.import numpy as np
2.x=np.array([[1,2],[3,4]])
3.y=np.array([[12,30]])
4.z=np.concatenate((x,y.T), axis=1)
5.z
```

T' used to change the rows into columns and columns into rows.

```
array([[ 1,  2, 12],
       [ 3,  4, 30]])
```

Class outline

1. Importing the Modules
2. The NumPy Array Object
3. Creating Arrays
4. Indexing and Slicing
5. Reshaping and Resizing
6. **Vectorized Expressions**
7. Matrix and Vector Operations

6. Vectorized Expressions

The purpose of storing numerical data in arrays is to be able to process the data with concise vectorized expressions that represent batch operations that are applied to all elements in the arrays.

Efficient use of vectorized expressions eliminates the need of many explicit for loops.

This results in less verbose code, better maintainability, and higher-performing code.

NumPy implements functions and vectorized operations corresponding to most fundamental mathematical functions and operators.

Vectorization in Python

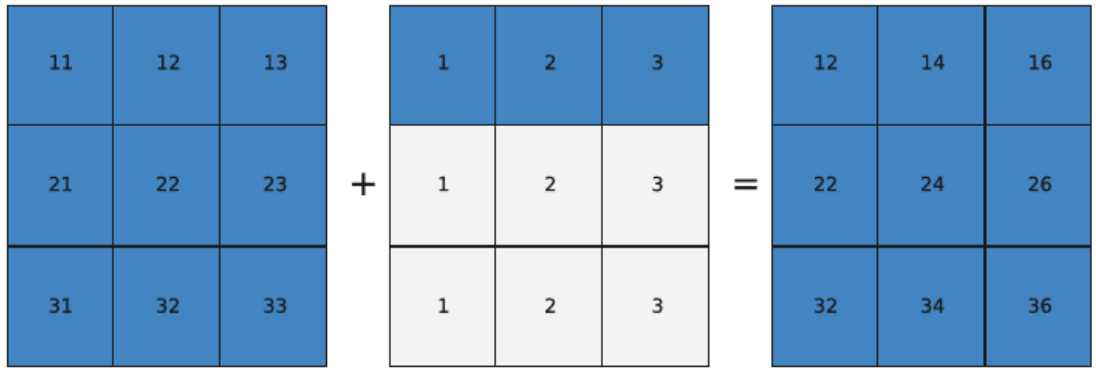
- Vectorization is a technique of implementing array operations without using for loops. Instead, we use functions defined by various modules which are highly optimized that reduces the running and execution time of code. Vectorized array operations will be faster than their pure Python equivalents, with the biggest impact in any kind of numerical computations.

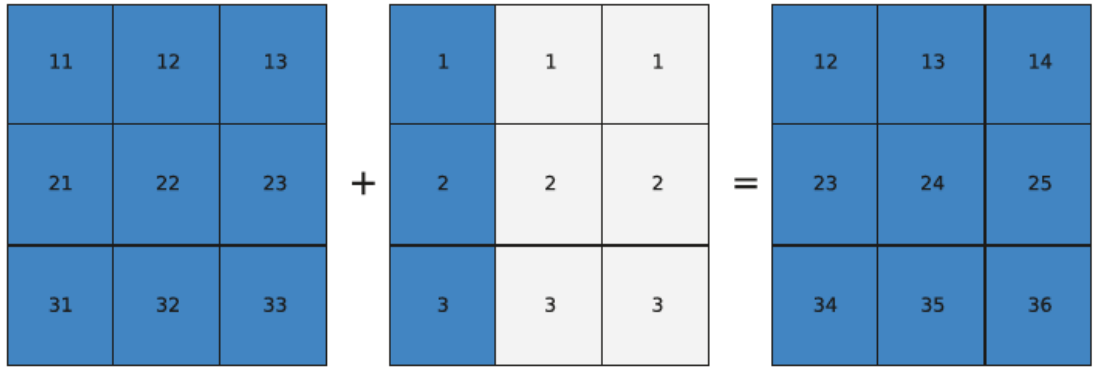
Vectorized Operations using NumPy

```python
import numpy as np
from timeit import Timer

# Creating a large array of size 10**6
array = np.random.randint(1000, size=10**6)

# method that adds elements using for loop
def add_forloop():
    new_array = [element + 1 for element in array]

# method that adds elements using vectorization
def add_vectorized():
    new_array = array + 1

# Finding execution time using timeit
computation_time_forloop = Timer(add_forloop).timeit(1)
computation_time_vectorized = Timer(add_vectorized).timeit(1)

print("Computation time is %0.9f using for-loop"%computation_time_forloop)
print("Computation time is %0.9f using vectorization"%computation_time_vectorized)
```

Computation time is 0.001202600 using for-loop
Computation time is 0.000236700 using vectorization

Visualization of broadcasting of row and column vectors into the shape of a matrix.





Arithmetic Operations

```
In [132]: x = np.array([[1, 2], [3, 4]])
In [133]: y = np.array([[5, 6], [7, 8]])
In [134]: x + y
Out[134]: array([[ 6,  8],
                 [10, 12]])

In [135]: y - x
Out[135]: array([[4, 4],
                 [4, 4]])


In [136]: x * y
Out[136]: array([[ 5, 12],
                 [21, 32]])


In [137]: y / x
Out[137]: array([[ 5.        ,  3.        ],
                 [ 2.33333333,  2.        ]])
```

```
In [138]: x * 2
Out[138]: array([[2, 4],
                 [6, 8]])
In [139]: 2 ** x
Out[139]: array([[ 2,  4],
                 [ 8, 16]])
In [140]: y / 2
Out[140]: array([[ 2.5,  3. ],
                 [ 3.5,  4. ]])
In [141]: (y / 2).dtype
Out[141]: dtype('float64')
```

Elementwise Functions

NumPy Function	Description
`np.cos, np.sin, np.tan`	Trigonometric functions.
`np.arccos, np.arcsin, np.arctan`	Inverse trigonometric functions.
`np.cosh, np.sinh, np.tanh`	Hyperbolic trigonometric functions.
`np.arccosh, np.arcsinh, np.arctanh`	Inverse hyperbolic trigonometric functions.
`np.sqrt`	Square root.
`np.exp`	Exponential.
`np.log, np.log2, np.log10`	Logarithms of base e, 2, and 10, respectively.

np.sin function (which takes only one argument) is used to compute the sine function for all values in the array:

```
>>> import numpy as np
>>> n = np.linspace(-1,1,11)
>>> n
array([-1. , -0.8, -0.6, -0.4, -0.2,  0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
>>> m = np.sin(np.pi*n)
>>> m
array([ -1.22464680e-16,  -5.87785252e-01,  -9.51056516e-01,
       -9.51056516e-01,  -5.87785252e-01,   0.00000000e+00,
        5.87785252e-01,   9.51056516e-01,   9.51056516e-01,
        5.87785252e-01,   1.22464680e-16])
>>> np.round(m,decimals=4)
array([-0.    , -0.5878, -0.9511, -0.9511, -0.5878,  0.    ,  0.5878,
       0.9511,  0.9511,  0.5878,  0.    ])
```

Boolean Arrays and Conditional Expressions

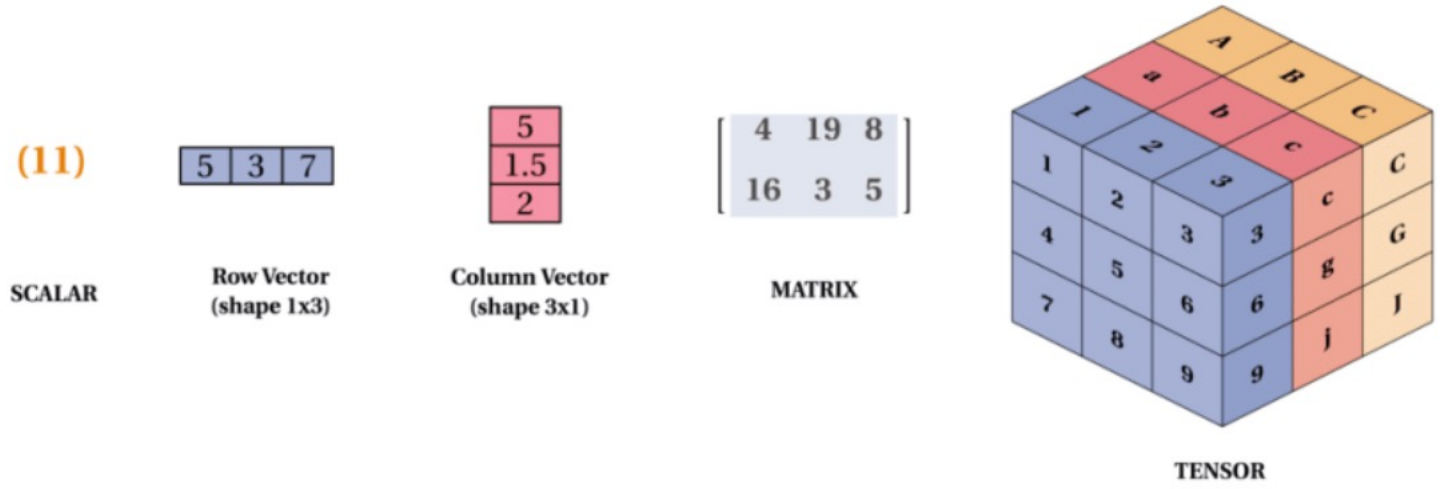
```
>>> a = np.array([1,2,3,4])
>>> a
array([1, 2, 3, 4])
>>> b = np.array([4,3,2,1])
>>> b
array([4, 3, 2, 1])
>>> a>b
array([False, False,  True,  True], dtype=bool)

>>> b>a
array([ True,  True, False, False], dtype=bool)
```

Class outline

1. Importing the Modules
2. The NumPy Array Object
3. Creating Arrays
4. Indexing and Slicing
5. Reshaping and Resizing
6. Vectorized Expressions
7. **Matrix and Vector Operations**

7. Matrix and Vector Operations

We have so far discussed general N-dimensional arrays. One of the main applications of such arrays is to represent the mathematical concepts of vectors, matrices, and tensors, and in this use-case, we also frequently need to calculate vector and matrix operations such as scalar (inner) products, dot (matrix) products, and tensor (outer) products.



NumPy Functions for Matrix Operations

NumPy Function	Description
`np.dot`	Matrix multiplication (dot product) between two given arrays representing vectors, arrays, or tensors.
`np.inner`	Scalar multiplication (inner product) between two arrays representing vectors.
`np.cross`	The cross product between two arrays that represent vectors.
`np.tensordot`	Dot product along specified axes of multidimensional arrays.
`np.outer`	Outer product (tensor product of vectors) between two arrays representing vectors.
`np.kron`	Kronecker product (tensor product of matrices) between arrays representing matrices and higher-dimensional arrays.
`np.einsum`	Evaluates Einstein's summation convention for multidimensional arrays.

NumPy.dot in Python

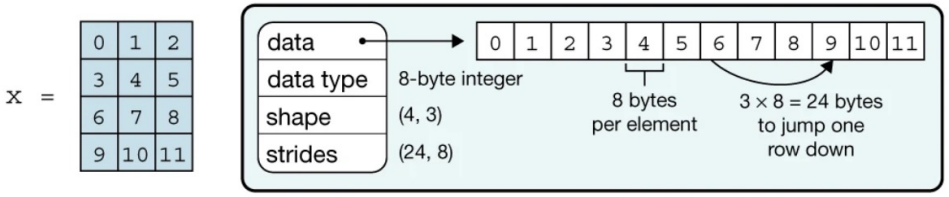
```
import numpy as np
a = np.array([[1,2],[3,4]])
b = np.array([[11,12],[13,14]])
np.dot(a,b)
```

```
[[37  40]
 [85  92]]
```

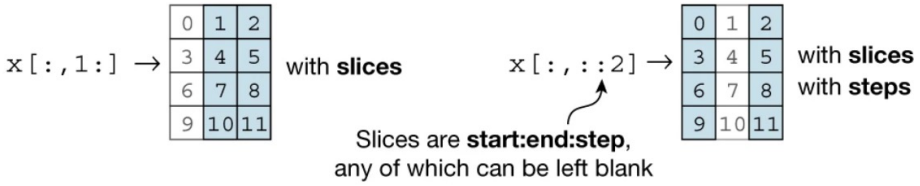
[[1*11+2*13, 1*12+2*14],[3*11+4*13, 3*12+4*14]]

Array Vector Matrix

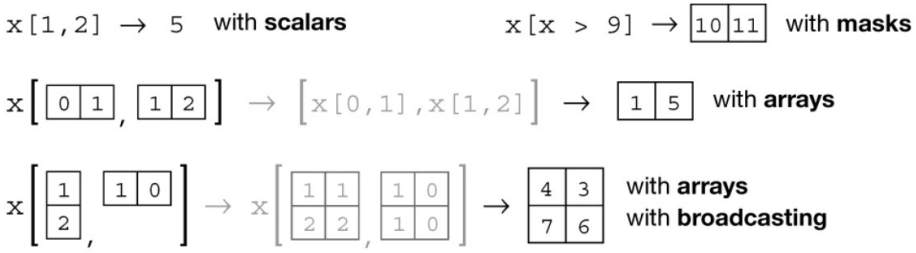
a Data structure



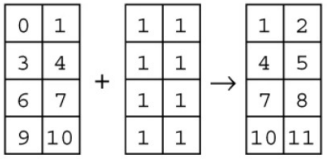
b Indexing (view)



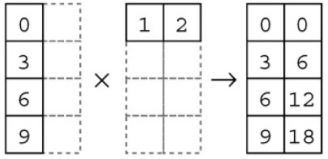
c Indexing (copy)



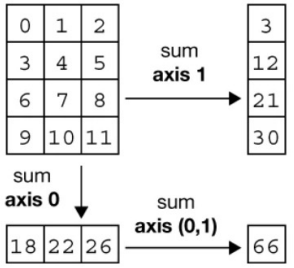
d Vectorization



e Broadcasting



f Reduction

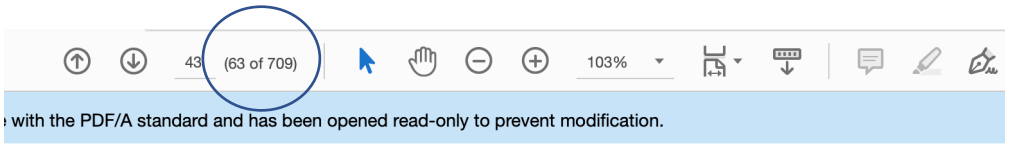


g Example

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

In [2]: x = np.arange(12)

In [3]: x = x.reshape(4, 3)

In [4]: x
Out[4]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])

In [5]: np.mean(x, axis=0)
Out[5]: array([4.5, 5.5, 6.5])

In [6]: x = x - np.mean(x, axis=0)

In [7]: x
Out[7]:
array([[-4.5, -4.5, -4.5],
       [-1.5, -1.5, -1.5],
       [ 1.5,  1.5,  1.5],
       [ 4.5,  4.5,  4.5]])
```

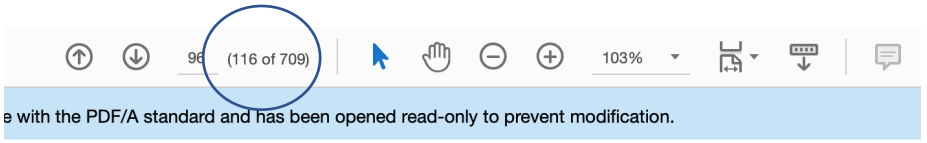
Student should read pages 63 – 116 on the following book to obtain extra knowledge.

Numerical Python Scientific Computing and Data Science Applications with Numpy, SciPy and Matplotlib -- Second Edition
— Robert Johansson



CHAPTER 2

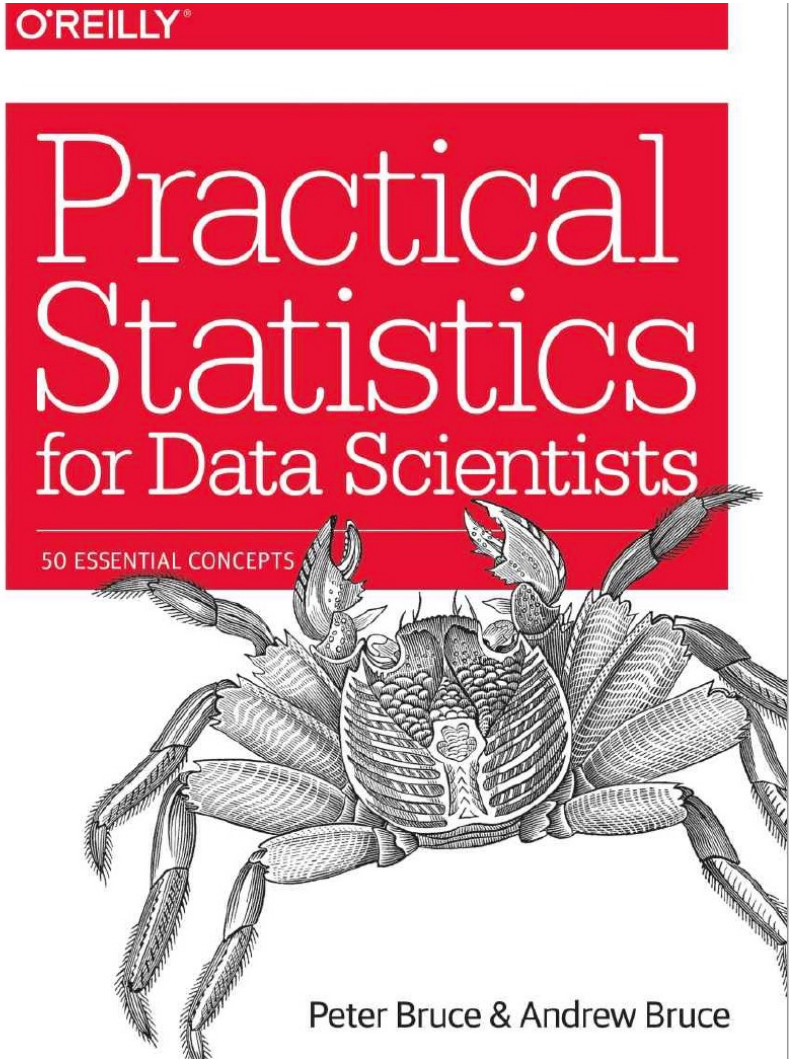
Vectors, Matrices, and Multidimensional Arrays



References

Idris, I. *Learning NumPy Array*. Mumbai: Packt, 2014.
—. *Numpy Beginner's Guide*. 3rd. Mumbai: Packt, 2015.
—. *NumPy Cookbook*. Mumbai: Packt, 2012.
McKinney, Wes. *Python for Data Analysis*. Sebastopol: O'Reilly, 2013.

Student can read the following book to get more knowledge.



End of Class 2 & 3