# NUMPY. Numerical Computing with Python

## Intro Notes

Python language is an excellent tool for general-purpose programming, with a highly readable syntax, rich and powerful data types and totally Zen.

However, it was not designed specifically for mathematical and scientific computing. In particular, Python lists are very flexible containers, but they are poorly suited to represent efficiently common mathematical constructs like vectors and matrices.

Fortunately, exists the **numpy** package (module) which is a package that provide high-performance vector, matrix and higher-dimensional data structures for Python. It is implemented in C and Fortran so when calculations are vectorized (formulated with vectors and matrices), performance is very good. It is used in almost all numerical computation using Python.

**Why not simply use Python lists for computations instead of creating a new array type?**

There are several reasons:

* Python lists are very general. They can contain any kind of object. They are dynamically typed. They do not support mathematical functions such as matrix and dot multiplications, etc. Implementating such functions for Python lists would not be very efficient because of the dynamic typing.
* Numpy arrays are statically typed and homogeneous. The type of the elements is determined when array is created.
* Numpy arrays are memory efficient.
* Because of the static typing, fast implementation of mathematical functions such as multiplication and addition of numpy arrays can be implemented in a compiled language (C and Fortran is used).

elgif

## What is Numpy?

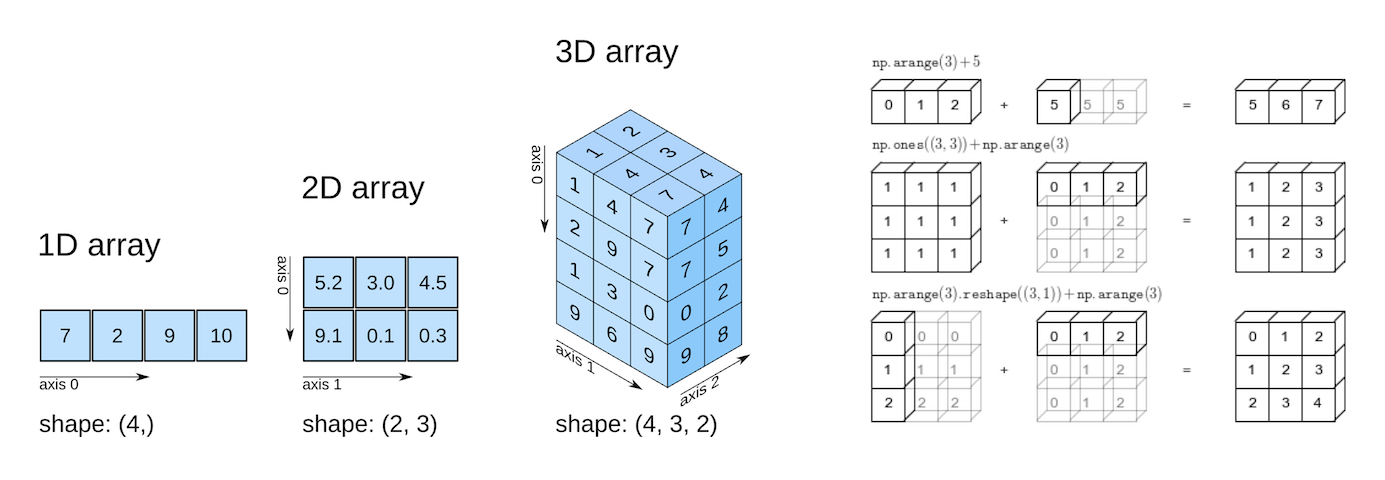
import numpy as np  
  
# Create a function to display information about a NumPy array  
def numpy\_info(arr):  
 print("\nThe dimension is:", arr.ndim)  
 print("The shape is:", arr.shape)  
 print("The size is:", arr.size)

NumPy, short for "Numerical Python," is a powerful library for the Python programming language. It is specifically designed to facilitate the creation and manipulation of large multidimensional arrays and matrices, along with an extensive collection of high-level mathematical functions to operate on these arrays. NumPy is widely used in scientific and data-related applications for its efficient numerical computing capabilities.

The origins of NumPy can be traced back to the "Numeric" library, initially developed by Jim Hugunin, with contributions from several other developers. Over time, NumPy evolved and expanded its capabilities. In 2005, Travis Oliphant played a crucial role in merging features from a competing library called Numarray into Numeric, resulting in the birth of NumPy. Since then, NumPy has gained tremendous popularity and has become an integral part of the Python ecosystem. It is an open-source project with a vibrant community of contributors.

At the heart of the NumPy package lies the ndarray (short for "n-dimensional array") object. This fundamental data structure allows users to create arrays with multiple dimensions, all containing homogeneous data types. One of the key strengths of NumPy is its ability to execute many operations efficiently by using code that is compiled for performance. This makes NumPy well-suited for tasks involving numerical computation and data manipulation.

There are several noteworthy differences between NumPy arrays and standard Python sequences:



## Work with numeric data

**Numpy** is here to make our life easier.

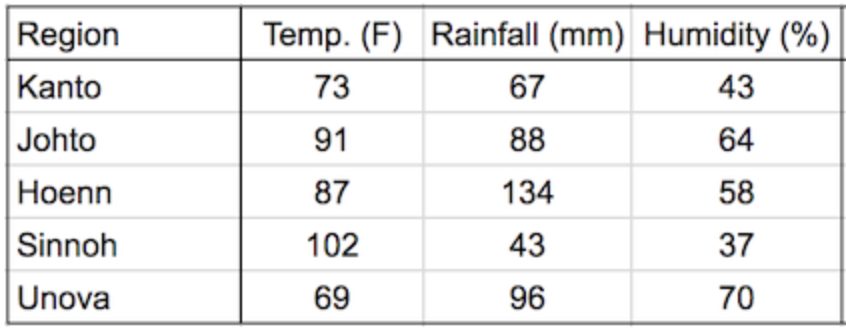
In the realm of data analysis, "data" often implies numerical information, such as stock prices, sales figures, sensor readings, sports scores, database tables, and more. For handling numerical computations in Python, the NumPy library offers specialized data structures, functions, and tools. Let's delve into an example to understand why and how NumPy is instrumental in working with numerical data.

Consider a scenario where we want to assess whether a particular region is suitable for apple cultivation based on climate-related data, including temperature, rainfall, and humidity. A straightforward approach is to establish a relationship between the annual apple yield (measured in tonnes per hectare) and climatic factors like mean temperature (in degrees Fahrenheit), rainfall (in millimeters), and mean relative humidity (in percentage). We can express this relationship as a linear equation:

yield\_of\_apples = w1 \* temperature + w2 \* rainfall + w3 \* humidity

Here, we represent apple yield as a weighted sum of temperature, rainfall, and humidity. While this equation serves as a simplification, the actual relationship may not be strictly linear, and additional factors could come into play. However, a basic linear model often provides practical insights.

By analyzing historical data statistically, we can obtain reasonable values for the coefficients w1, w2, and w3. Here's an example of possible coefficient values:



We can now substitute these variables into the linear equation to predict the yield of the apples.

w1, w2, w3 = 0.3, 0.2, 0.5

yield\_of\_apples = w1 \*temperature + w2\* rainfall + w3 \* humidity

To make the above calculation a bit easier for multiple regions, we can represent the climate data for each region as a vector, i.e. a list of numbers.

yield\_of\_apples = w1 \* 73 + w2 \* 67 + w3 \* 43

The three numbers in each vector represent the temperature, precipitation, and humidity data, respectively. We can also represent the set of weights used in the formula as a vector.

weights = [w1, w2, w3]

How can we calculate the cultivation of each region?

# Define the weights as a vector  
weights = [w1, w2, w3]  
  
# Define the climate data for Kanto as a vector  
kanto\_as\_list = [73, 67, 43]

# Calculate the yield of apples for Kanto using the dot product  
yield\_kanto = sum(w \* x for w, x in zip(weights, kanto\_as\_list))

# Similarly, you can calculate the yield for other regions like Johto  
johto = [91, 88, 64]  
yield\_johto = sum(w \* x for w, x in zip(weights, johto))

## Go from Python lists to Numpy arrays

The operation we performed, which involved elementwise multiplication of two vectors and then summing the results, is commonly known as the dot product. If you'd like to explore the dot product in more detail, you can refer to the official NumPy documentation here: [NumPy Dot Product](https://numpy.org/doc/stable/reference/generated/numpy.dot.html).

Before we can utilize NumPy's built-in function to compute the dot product, we need to convert our Python lists into NumPy arrays. To do this, we'll start by installing the NumPy library using the pip package manager.

Next, we are going to import the numpy module. It is common practice to import numpy with the alias np. This is a convention, all programmers use the same names when importing libraries. It really works if I put any name, in case of numpy we always use np

import numpy as np

Let's start with a list, as before:

kanto\_as\_list = [73, 67, 43]  
type(kanto\_as\_list)

list

Now, we can convert it into a numpy array with the following operation:

kanto\_array = np.array(kanto\_as\_list)  
type(kanto\_array)

numpy.ndarray

Let's do the same for the array:

weights\_array = np.array([0.3, 0.2, 0.5])  
kanto\_array \* weights\_array

array([21.9, 13.4, 21.5])

Now, we can perform previous operation with simply this:

And the final output is:

sum(kanto\_array \* weights\_array)

56.8

## Operating with Numpy Arrays

The \* operator performs an elementwise multiplication of two arrays if they are the same size. The sum method calculates the sum of the numbers in an array.

weights\_array \* kanto\_array

array([21.9, 13.4, 21.5])

This operation (multiplication + sum) is called a dot product.

We can now calculate the dot product of the two vectors using the np.dot function.

np.dot(weights\_array, kanto\_array)

56.8

You can find all mathematical operators for numpy arrays [here](https://numpy.org/doc/stable/reference/routines.math.html).

## Advantages of using Numpy arrays

NumPy arrays offer several advantages over Python lists, especially when working with numeric data in data analysis and scientific computing:

**Ease of Use**

One of the primary advantages of NumPy arrays is their ease of use for performing mathematical operations. You can write concise and intuitive expressions that operate on entire arrays, making your code more readable and reducing the need for explicit loops.

For example, suppose you have two arrays, kanto and weights, representing climate data and corresponding weights. You can easily calculate the yield of apples for the Kanto region as:

yield\_of\_apples = (kanto \* weights).sum()

**Performance**

Another significant advantage of NumPy arrays is their performance. NumPy's operations and functions are implemented internally in C or C++, which makes them considerably faster than using equivalent Python statements and loops that are interpreted at runtime.

import time  
import numpy as np

Now, let's measure the time it takes to calculate the yield using Python lists

# First, let's define the weights and climate data for the Kanto region  
weights = [0.3, 0.2, 0.5] # Weights for temperature, rainfall, and humidity  
kanto\_as\_list = [73, 67, 43] # Climate data for Kanto as a list  
  
# Function to calculate apple yield manually using Python lists  
def apples\_per\_region(lst\_):  
 counter = 0   
 for j, k in zip(weights, lst\_):  
 counter += j \* k  
 return counter  
  
start\_time = time.time() # Record the start time  
for i in range(10000):  
 result = apples\_per\_region(kanto\_as\_list) # Calculate yield using Python lists  
end\_time = time.time() # Record the end time  
  
elapsed\_time = end\_time - start\_time # Calculate the elapsed time  
print(f"Yield calculated using Python lists: {result}")  
print(f"Time taken: {elapsed\_time} seconds")

Yield calculated using Python lists: 56.8  
Time taken: 0.0021948814392089844 seconds

# Now, let's perform the same calculation using NumPy arrays  
kanto\_array = np.array([73, 67, 43]) # Climate data for Kanto as a NumPy array  
weights\_array = np.array([0.3, 0.2, 0.5]) # Weights as a NumPy array  
  
# Measure the time it takes to calculate the yield using NumPy arrays  
start\_time = time.time() # Record the start time  
for i in range(0, 10000):  
 result = np.dot(weights\_array, kanto\_array) # Calculate yield using NumPy arrays  
end\_time = time.time() # Record the end time  
  
elapsed\_time = end\_time - start\_time # Calculate the elapsed time  
print(f"Yield calculated using NumPy arrays: {result}")  
print(f"Time taken: {elapsed\_time} seconds")

Yield calculated using NumPy arrays: 56.8  
Time taken: 0.008521080017089844 seconds

The slight difference in execution time, where NumPy appears to take slightly longer, is primarily due to the additional overhead introduced by NumPy's internal operations for handling arrays. NumPy is optimized for complex mathematical operations on large datasets, which might involve additional setup and memory management. For very simple calculations like this example, the overhead of NumPy can sometimes make it appear slightly slower.

However, in practice, the difference in execution time is negligible for most data-intensive tasks, especially when dealing with large datasets or complex mathematical operations. NumPy truly shines when performing more complex operations or working with larger datasets, where its performance advantages become much more apparent.

So, while in this specific case, NumPy might appear slightly slower, it's essential to consider the broader context where NumPy's efficiency becomes a significant advantage.

NumPy provides a more convenient and efficient way to work with numerical data, enabling you to write cleaner, more expressive code and achieve better performance in data analysis and scientific computing tasks.

## Create Arrays

In data analysis and scientific computing, generating random data is a common task. NumPy provides powerful tools to create arrays filled with random numbers, which can be particularly useful for tasks like simulating experiments or generating synthetic datasets.

### 1. 1darray: vector

# Generate an array of 7 random floating-point numbers between 0 and 1  
vector = np.random.random(7)  
  
# Print the generated vector  
print("\nRandom Vector:")  
print(vector)

Random Vector:  
[0.78109394 0.17074704 0.45933674 0.9311047 0.57176349 0.92492549  
 0.20710658]

# Call the numpy\_info function to display information about the 'vector' array  
print("\nArray Information:")  
numpy\_info(vector)

Array Information:  
  
The dimension is: 1  
The shape is: (7,)  
The size is: 7

### 2. 2darray: matrix

# Generate a random 2D NumPy array with dimensions 3x4  
matrix = np.random.random((3, 4)) # (rows, columns)  
  
# Print the generated vector  
print("\nRandom Matrix:")  
print(matrix)

Random Matrix:  
[[0.96102145 0.24550444 0.18690855 0.9691706 ]  
 [0.06964345 0.15006866 0.01940452 0.90671918]  
 [0.97878937 0.33531014 0.57403958 0.65452511]]

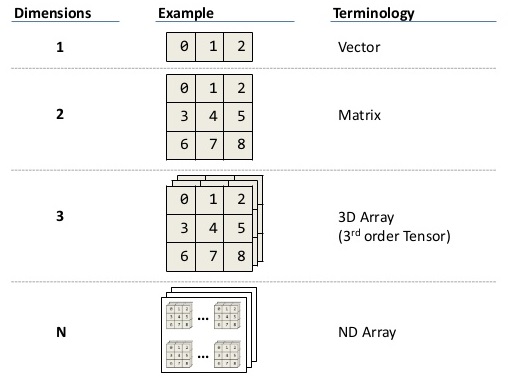
# Call the numpy\_info function to display information about the 'matrix' array  
print("\nArray Information:")  
numpy\_info(matrix)

Array Information:  
  
The dimension is: 2  
The shape is: (3, 4)  
The size is: 12

### 3. 3darray: tensor

# Generate a random 2D NumPy array with dimensions 3 x 4 x 5  
tensor = np.random.randint(3, size=(3, 4, 5)) # (block, rows, columns)  
  
# Print the generated vector  
print("\nRandom tensor:")  
print(tensor)

Random tensor:  
[[[1 1 2 0 0]  
 [2 1 1 2 2]  
 [1 2 2 0 1]  
 [2 1 2 2 2]]  
  
 [[1 0 0 0 2]  
 [1 0 1 2 0]  
 [2 0 1 1 1]  
 [0 2 0 0 1]]  
  
 [[2 1 1 2 2]  
 [2 1 1 2 2]  
 [0 0 1 0 0]  
 [1 0 1 0 2]]]



Numpy also provides a variety of convenient functions for creating arrays with specific shapes and containing either fixed or random values. You can explore these array creation functions further by referring to the official [documentation](https://numpy.org/doc/stable/reference/routines.array-creation.html) or by using the built-in help function for more comprehensive information and examples. These functions simplify the process of initializing arrays to suit your specific data manipulation needs.

### random numbers

**Random Randint vs. Random Random**

In this section, we'll explore two essential functions for generating random numbers in NumPy: [**numpy.random.randint**](https://numpy.org/doc/stable/reference/random/generated/numpy.random.randint.html) and [**numpy.random.random**](https://numpy.org/doc/stable/reference/random/generated/numpy.random.random.html).

* **numpy.random.randint**: This function generates random integers within a specified range. It's useful for creating arrays of whole numbers when simulating scenarios that involve discrete values.
* **numpy.random.random**: This function produces random floating-point numbers between 0 and 1. It's suitable for tasks that require continuous random variables, such as modeling uncertainty in scientific experiments.

By understanding these two functions, you'll gain the capability to generate diverse sets of random data tailored to your specific needs. Whether you're conducting statistical simulations, testing algorithms, or exploring probability distributions, NumPy's random number generation capabilities will prove invaluable.

Let's dive into each of these functions to see how they work and how to use them effectively.

# Import the NumPy library as np  
import numpy as np  
  
# Generate an array of 10 random floating-point numbers between 0 and 1  
random\_numbers = np.random.random(10)  
  
# Print the generated array  
print("Random Numbers Array:")  
print(random\_numbers)

Random Numbers Array:  
[0.91276691 0.27204851 0.61022619 0.86189665 0.55526864 0.50369785  
 0.35797766 0.32030749 0.68200863 0.30369334]

**OFF-TOPIC**

* **Randomness**: Randomness is a concept related to unpredictability and chance in data or events.
* **Time in Seconds**: Measuring time in seconds can be useful for timing events or processes.
* **Coordinates**: Coordinates represent points in space or on a plane, often used in geometry and mapping.

# Create a function to display information about a NumPy array  
def numpy\_info(arr):  
 print("\nThe dimension is:", arr.ndim)  
 print("The shape is:", arr.shape)  
 print("The size is:", arr.size)

The numpy\_info function is designed to provide information about a NumPy array. It takes an input array arr and displays three key characteristics of the array:

1. **Dimension (arr.ndim):** This method returns the number of dimensions in the input array arr. For example, if arr is a 1-dimensional array, it will return 1 (vector). If it's a 2-dimensional array, it will return 2 (matrix), and so on.
2. **Shape (arr.shape):** This method returns a tuple representing the dimensions of the array. The tuple contains the length of the array along each dimension. For example, if arr is a 1-dimensional array with 7 elements, it will return (7,). If it's a 2-dimensional array with dimensions 3x4, it will return (3, 4).
3. **Size (arr.size):** This method returns the total number of elements in the array. It's essentially the product of the lengths of all dimensions. For example, if arr is a 2-dimensional array with dimensions 3x4, it will return 12 because there are a total of 12 elements in the array.

The numpy\_info function's purpose is to quickly summarize the structure and size of a NumPy array for better understanding and analysis.

# Call the numpy\_info function to display information about the 'matrix' array  
print("\nArray Information:")  
numpy\_info(tensor)

Array Information:  
  
The dimension is: 3  
The shape is: (3, 4, 5)  
The size is: 60

### Range

RANGE You can use NumPy to easily create a range of values using the np.arange() function. This function takes three arguments: start, stop, and step, and it returns an array of values starting from start, up to (but not including) stop, with increments of step.

Here's an example:

# Does the same as range() function from build-in python  
range\_of\_values = np.arange(0, 100)

### Linespace

The NumPy linspace function is a versatile tool for creating sequences of evenly spaced values within a specified interval. It allows you to define a start point and an end point for a given range and determine the total number of evenly spaced values you want within that interval. Importantly, this sequence includes both the start and end points.

Here's how you can use the np.linspace() function:

np.linspace(0, 100, 50)

array([ 0. , 2.04081633, 4.08163265, 6.12244898,  
 8.16326531, 10.20408163, 12.24489796, 14.28571429,  
 16.32653061, 18.36734694, 20.40816327, 22.44897959,  
 24.48979592, 26.53061224, 28.57142857, 30.6122449 ,  
 32.65306122, 34.69387755, 36.73469388, 38.7755102 ,  
 40.81632653, 42.85714286, 44.89795918, 46.93877551,  
 48.97959184, 51.02040816, 53.06122449, 55.10204082,  
 57.14285714, 59.18367347, 61.2244898 , 63.26530612,  
 65.30612245, 67.34693878, 69.3877551 , 71.42857143,  
 73.46938776, 75.51020408, 77.55102041, 79.59183673,  
 81.63265306, 83.67346939, 85.71428571, 87.75510204,  
 89.79591837, 91.83673469, 93.87755102, 95.91836735,  
 97.95918367, 100. ])

In this function, start represents the beginning of the interval, end is the endpoint, and num is the total number of values you want in the sequence. NumPy will generate a sequence of values that are evenly distributed between start and end, inclusive of both endpoints.

The linspace function is particularly useful for tasks like defining values along a continuous axis for plotting graphs or creating evenly spaced intervals for numerical calculations.

For more details and options, you can refer to the official [NumPy linspace documentation](https://numpy.org/doc/stable/reference/generated/numpy.linspace.html).

### Around

It allows us to round the arrays to the decimals that we tell it 🙃

np.round(np.linspace(0, 100, 50), 3)

array([ 0. , 2.041, 4.082, 6.122, 8.163, 10.204, 12.245,  
 14.286, 16.327, 18.367, 20.408, 22.449, 24.49 , 26.531,  
 28.571, 30.612, 32.653, 34.694, 36.735, 38.776, 40.816,  
 42.857, 44.898, 46.939, 48.98 , 51.02 , 53.061, 55.102,  
 57.143, 59.184, 61.224, 63.265, 65.306, 67.347, 69.388,  
 71.429, 73.469, 75.51 , 77.551, 79.592, 81.633, 83.673,  
 85.714, 87.755, 89.796, 91.837, 93.878, 95.918, 97.959,  
 100. ])

### Arrays from data

In NumPy, you can easily create arrays from existing data structures like lists. This section explores various ways to convert between Python lists and NumPy arrays.

**Converting a List to a NumPy Array**

Suppose you have a Python list a with some values:

a = [90, 50, 0]  
  
## You can convert this list into a NumPy array using `np.array()`:  
a = np.array(a)  
  
##Now, the variable `a` holds a NumPy array with the same values:

##### **Converting an Array to a List**:

If you have a NumPy array and want to convert it back to a Python list, you have a few options:

1. Using the tolist() method:
2. Using the list() constructor:

b = a.tolist()  
  
## or  
list(a)

[90, 50, 0]

#### **Multi-Dimensional Arrays**:

NumPy can also handle multi-dimensional arrays with ease. For instance, if you want to create a 3D array with random integers:

a = np.random.randint(10, size=(2, 2, 3))  
a

array([[[9, 1, 4],  
 [9, 1, 3]],  
  
 [[5, 9, 0],  
 [5, 5, 7]]])

This will result in a 3D NumPy array. Be cautious when converting such arrays to lists, as the structure can become nested and less intuitive

c\_badly\_done = list(a)  
c\_badly\_done

[array([[9, 1, 4],  
 [9, 1, 3]]),  
 array([[5, 9, 0],  
 [5, 5, 7]])]

In the example above, c\_badly\_done will be a nested list with the structure reflecting the original array's dimensions.

Conclusion Understanding how to convert data between Python lists and NumPy arrays is essential for working with data efficiently. Depending on your needs, you can easily switch between these data structures while taking care to manage the dimensions properly.

Always be mindful of the data structure you are working with, especially when handling multi-dimensional arrays, as converting them to lists may result in nested structures that require additional handling.

### Array of ZEROs

np.zeros is used to create the array where all elements are 0. Its syntax is: np.zeros(shape, dtype=float)

Where:

* The shape is the size of the array, and can be 1-D, 2-D or multiple dimensions.
* The dtype is float64 by default, but can be set to any data type in NumPy.

np.zeros((2, 5, 10))

array([[[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],  
 [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],  
 [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],  
 [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],  
 [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]],  
  
 [[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],  
 [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],  
 [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],  
 [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],  
 [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]]])

### Array of ONES

Similar to creating arrays of zeros, you can create an array where all elements are set to 1 using np.ones(). The syntax and parameters of np.ones() are identical to those of np.zeros():

np.ones((2, 5, 10))

array([[[1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],  
 [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],  
 [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],  
 [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],  
 [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]],  
  
 [[1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],  
 [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],  
 [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],  
 [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],  
 [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]]])

### Diagonal Array

(Who remembers Linear Algebra? Why "eye"?)

The np.eye() function in NumPy is used to generate a two-dimensional array with ones along the main diagonal and zeros elsewhere. This can be especially useful when you need to create identity matrices, which are square matrices with ones on the diagonal and zeros everywhere else.

Here's how you can use np.eye() with its parameters:

* **N**: An integer that specifies the number of rows in the resulting array. This determines the size of the square matrix.
* **M** (optional): An integer that specifies the number of columns in the array. By default, it is set to None, which means it will be equal to N, resulting in a square matrix. However, you can set M to a different value to create a rectangular matrix.
* **k** (optional): An integer, the default value is 0. It determines the position of the diagonal. When k is 0, it places ones on the main diagonal. If k is positive, it shifts the diagonal upwards, creating an upper diagonal with the offset of k. Conversely, if k is negative, it shifts the diagonal downwards, forming a lower diagonal with the offset of -k.
* **dtype** (optional): The data type of the array elements. By default, it is set to float, but you can specify other data types as needed.

np.eye(5, 5, dtype=int)

array([[1, 0, 0, 0, 0],  
 [0, 1, 0, 0, 0],  
 [0, 0, 1, 0, 0],  
 [0, 0, 0, 1, 0],  
 [0, 0, 0, 0, 1]])

np.eye(5, 5, 2) # Return a 2-D array with ones on the diagonal and zeros elsewhere.

array([[0., 0., 1., 0., 0.],  
 [0., 0., 0., 1., 0.],  
 [0., 0., 0., 0., 1.],  
 [0., 0., 0., 0., 0.],  
 [0., 0., 0., 0., 0.]])

## Dtypes

DTypes

NumPy provides various data types (dtypes) to represent and work with different kinds of data efficiently. These data types are essential for specifying the type of data an array can hold, and they are a crucial aspect of NumPy's functionality.

Here's a list of common NumPy data types:

* **int8, int16, int32, int64**: Signed integer types with different bit sizes.
* **uint8, uint16, uint32, uint64**: Unsigned (non-negative) integer types with different bit sizes.
* **float16, float32, float64**: Floating-point types with varying precision.
* **complex64, complex128**: Complex number types.
* **bool**: Boolean type.
* **object**: Generic Python object type.
* **string\_**: String type (fixed-size ASCII strings).
* **unicode\_**: Unicode string type (fixed-size Unicode strings).
* **datetime64**: Date and time type.
* **timedelta64**: Differences between two datetime values.
* **void**: Raw data type (for structured arrays).
* **structured**: User-defined structured data types.
* **unicodechar**: A single Unicode character.

These data types allow you to work with a wide range of data efficiently, making NumPy a powerful library for numerical and scientific computing. You can explore the [official NumPy documentation](https://numpy.org/doc/stable/user/basics.types.html) for more detailed information about these data types and their usage.

np.eye(5, 5) # type: float

array([[1., 0., 0., 0., 0.],  
 [0., 1., 0., 0., 0.],  
 [0., 0., 1., 0., 0.],  
 [0., 0., 0., 1., 0.],  
 [0., 0., 0., 0., 1.]])

np.eye(5, 5, dtype="int") # type: int

array([[1, 0, 0, 0, 0],  
 [0, 1, 0, 0, 0],  
 [0, 0, 1, 0, 0],  
 [0, 0, 0, 1, 0],  
 [0, 0, 0, 0, 1]])

np.eye(5, 5, dtype="str") # type: str

array([['1', '', '', '', ''],  
 ['', '1', '', '', ''],  
 ['', '', '1', '', ''],  
 ['', '', '', '1', ''],  
 ['', '', '', '', '1']], dtype='<U1')

# if a dataframe has dtype=object -> it's a string  
np.eye(5, 5, dtype="object") # type: object

array([[1, 0, 0, 0, 0],  
 [0, 1, 0, 0, 0],  
 [0, 0, 1, 0, 0],  
 [0, 0, 0, 1, 0],  
 [0, 0, 0, 0, 1]], dtype=object)

## Business challenge: Forex Data Analysis Challenge

Business challenge: Forex Data Analysis Challenge

In this exercise, we'll embark on a Forex data analysis challenge, simulating a scenario where you need to analyze historical foreign exchange (Forex) rate data for a currency pair. You've been provided with a dataset containing daily Forex rates for a specific currency pair over a month. Your objective is to leverage NumPy to extract valuable insights from this financial data.

### Dataset Description

The dataset comprises two NumPy arrays:

* dates: An array containing date values representing each trading day of the month.
* exchange\_rates: An array of exchange rate values for a specific currency pair corresponding to each date.

Your tasks will involve:

1. Calculating the average exchange rate for the entire month.
2. Identifying the date with the highest exchange rate.
3. Determining the total trading volume for the month.

By the end of this exercise, you'll gain hands-on experience in utilizing NumPy for financial data analysis, a valuable skill for professionals in the world of Forex trading and finance.

Let's kick things off by calculating the average exchange rate for the entire month!

**Hint!**: check the documentation [here](https://numpy.org/doc/stable/reference/routines.html).

import numpy as np  
  
# Sample Forex data (you can replace this with your dataset)  
dates = np.array(['2023-09-01', '2023-09-02', '2023-09-03', '2023-09-04', '2023-09-05'])  
exchange\_rates = np.array([1.15, 1.16, 1.14, 1.17, 1.15])  
  
# 1. Calculating the average exchange rate for the entire month  
# [your code here]  
print(f"Average Exchange Rate for the Month: {average\_rate:.2f}")  
  
# 2. Identifying the date with the highest exchange rate  
# [your code here]  
print(f"Highest Exchange Rate Date: {max\_rate\_date}, Rate: {max\_rate:.2f}")  
  
# 3. Determining the total trading volume for the month (assuming equal volume each day)  
# [your code here]  
print(f"Total Trading Volume for the Month: {total\_volume:.2f}")

---------------------------------------------------------------------------  
NameError Traceback (most recent call last)  
Cell In[41], line 9  
 5 exchange\_rates = np.array([1.15, 1.16, 1.14, 1.17, 1.15])  
 7 # 1. Calculating the average exchange rate for the entire month  
 8 # [your code here]  
----> 9 print(f"Average Exchange Rate for the Month: {average\_rate:.2f}")  
 11 # 2. Identifying the date with the highest exchange rate  
 12 # [your code here]  
 13 print(f"Highest Exchange Rate Date: {max\_rate\_date}, Rate: {max\_rate:.2f}")  
  
NameError: name 'average\_rate' is not defined

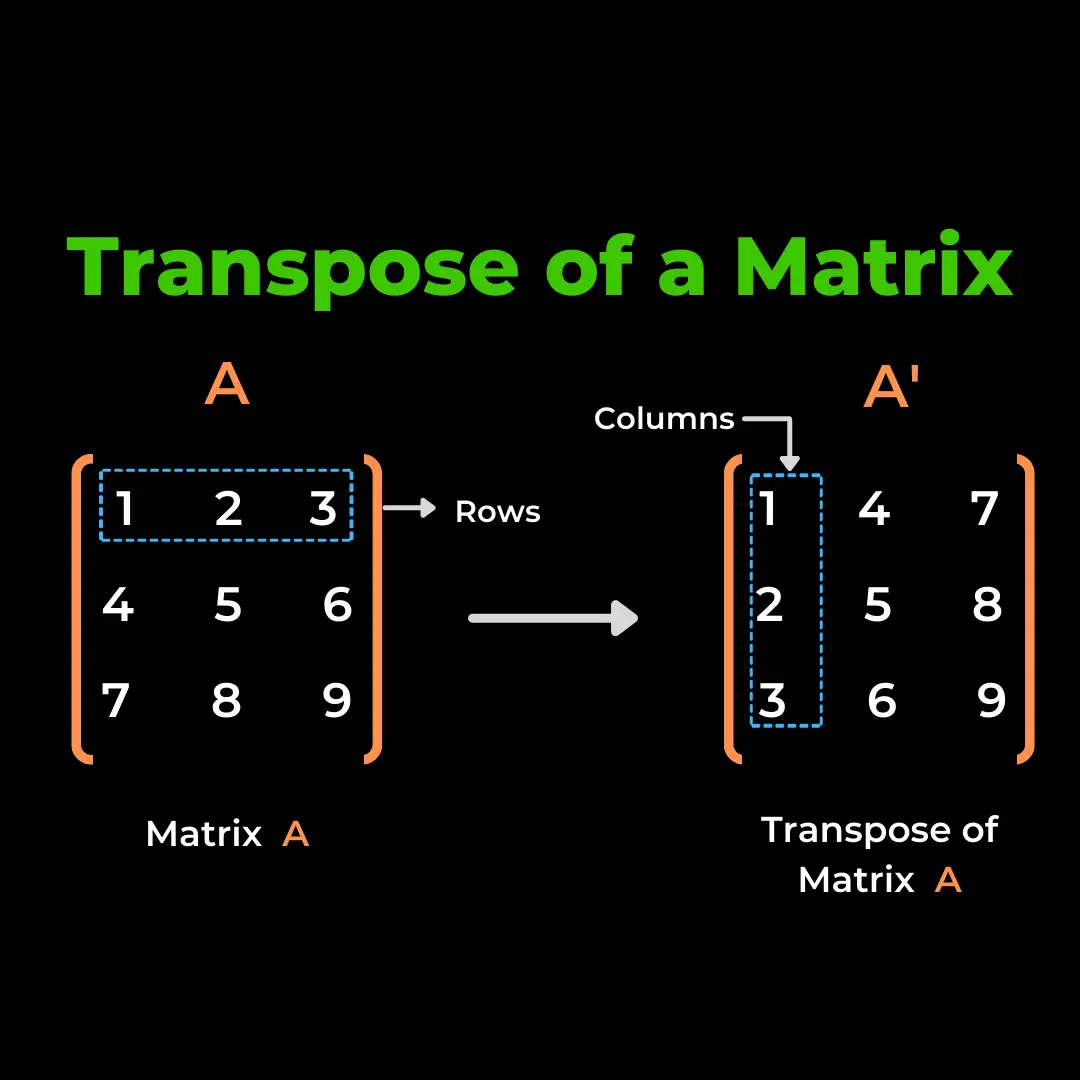
## Transpose

[Transpose](https://numpy.org/doc/stable/reference/generated/numpy.transpose.html)

In data analysis and linear algebra, the transpose of a matrix is a fundamental operation that flips the matrix over its diagonal. This operation swaps the rows and columns of the matrix, effectively changing the orientation of the data. Transposition is commonly used in various mathematical operations and transformations, making it a crucial concept in numerical computing.

You can think of the transpose as a way to represent the same data from a different perspective, especially when dealing with multi-dimensional arrays or matrices. It is often used to align data correctly for mathematical operations, including matrix multiplication, solving linear equations, and more.

In NumPy, the numpy.transpose() function allows you to easily transpose arrays and matrices, providing a versatile tool for data manipulation and analysis.



Please note that the image above is a simplified representation of matrix transposition, showing the change in orientation from rows to columns and vice versa. In practice, the transpose operation can involve complex data transformations, but it serves as a critical building block for various mathematical computations and data manipulations.

# Create a sample 2D array (matrix)  
matrix = np.array([[1, 2, 3],  
 [4, 5, 6],  
 [7, 8, 9]])  
  
# Display the original matrix  
print("Original Matrix:")  
print(matrix)

Original Matrix:  
[[1 2 3]  
 [4 5 6]  
 [7 8 9]]

# Use numpy.transpose() to transpose the matrix  
transposed\_matrix = np.transpose(matrix)  
transposed\_matrix\_alt = matrix.T  
  
# Display the transposed matrix  
print("\nTransposed Matrix:")  
print(transposed\_matrix)  
  
print("\nAlternative Transposed Matrix:")  
print(transposed\_matrix\_alt)  
  
print(np.array\_equal(transposed\_matrix, transposed\_matrix\_alt))

Transposed Matrix:  
[[1 4 7]  
 [2 5 8]  
 [3 6 9]]  
  
Alternative Transposed Matrix:  
[[1 4 7]  
 [2 5 8]  
 [3 6 9]]  
True

## Reshaping Arrays in NumPy

Reshaping arrays is a fundamental operation in data manipulation and analysis. It involves changing the dimensions or shape of an array while preserving its original data. NumPy, a powerful library for numerical computing in Python, provides a versatile function called numpy.reshape() for this purpose.

Reshaping allows you to convert 1D arrays into 2D arrays (and vice versa), rearrange the dimensions of multi-dimensional arrays, and efficiently prepare data for various data analysis and machine learning tasks.

In this section, we'll explore how to reshape arrays in NumPy, including transforming arrays between different dimensions and examining the properties of reshaped arrays.

Let's dive into the world of array reshaping and discover how it can be a valuable tool in your data analysis toolkit.

reshape

# Create a 1D array  
array\_1d = np.array([1, 2, 3, 4, 5, 6])  
print(array\_1d)

[1 2 3 4 5 6]

# Reshape the 1D array into a 2D array with 2 rows and 3 columns  
array\_2d = np.reshape(array\_1d, (2, 3))  
print(array\_2d)  
  
# Check the shape of the reshaped array  
print("\nShape of Reshaped Array:", array\_2d.shape)

[[1 2 3]  
 [4 5 6]]  
  
Shape of Reshaped Array: (2, 3)

# Reshape the 2D array back to a 1D array  
array\_1d\_reshaped = np.reshape(array\_2d, (6, )) # we are telling the away we want a 1D array, with size 6  
print(array\_1d\_reshaped)  
  
## This would also work  
# array\_1d\_reshaped = np.reshape(array\_2d, -1) # we are telling the away we want a 1D array, with size adjusted to the amount of elements

[1 2 3 4 5 6]

## Updating the value of a ndim array

In data analysis and numerical computing, it's often necessary to modify the values of individual elements within multi-dimensional arrays. NumPy, a powerful library for array manipulation in Python, provides various techniques to update the values of ndimensional arrays efficiently.

In this section, we'll explore methods and techniques for updating the values of ndim arrays in NumPy. Whether you're working with data cleaning, transformation, or any data manipulation task, knowing how to update array values is an essential skill.

Let's delve into the world of array value updates and discover how to perform these operations effectively using NumPy.

#### 1. overwriting

# Create a sample 2D array  
original\_array = np.array([[1, 2, 3],  
 [4, 5, 6],  
 [7, 8, 9]])  
  
print(original\_array)  
  
original\_array[1, 1] = 99  
  
print("\nUpdated Array:")  
print(original\_array)

[[1 2 3]  
 [4 5 6]  
 [7 8 9]]  
  
Updated Array:  
[[ 1 2 3]  
 [ 4 99 6]  
 [ 7 8 9]]

#### 2. using its index

# Update a specific element using its index  
original\_array[0][-1] = 60  
  
print("Updated Array:")  
print(original\_array)

Updated Array:  
[[ 1 2 60]  
 [ 4 99 6]  
 [ 7 8 9]]

#### 3. another syntax

# Create a 2D NumPy array  
array\_new = np.array(([1, 2, 3], [2, 2, 0]))  
array\_new

array([[1, 2, 3],  
 [2, 2, 0]])

# Create a boolean Matrix  
# Why? boolean matrix can be used as a filter  
print(array\_new)  
print(array\_new==2)  
  
## And we can index array\_new using the boolean matrix  
array\_new[array\_new==2]

[[1 2 3]  
 [2 2 0]]  
[[False True False]  
 [ True True False]]

array([2, 2, 2])

We update the elements in array\_new where the condition array\_new == 2 is true. In this case, we set those elements to 20.

array\_new[array\_new==2] = 20  
array\_new

array([[ 1, 20, 3],  
 [20, 20, 0]])

## Array comparison

NumPy arrays not only store data efficiently but also support a wide range of comparison operations. These operations allow you to compare elements within arrays, resulting in new arrays of Boolean values. These Boolean arrays serve as masks, enabling you to extract or manipulate elements based on specific conditions.

# Create two NumPy arrays  
array1 = np.array([1, 2, 3, 4, 5])  
array2 = np.array([3, 4, 1, 6, 5])

# Equality comparison  
equal\_result = (array1 == array2)  
print("Equality Comparison Result:")  
print(equal\_result)

Equality Comparison Result:  
[False False False False True]

# Inequality comparison  
not\_equal\_result = (array1 != array2)  
print("\nInequality Comparison Result:")  
print(not\_equal\_result)

Inequality Comparison Result:  
[ True True True True False]

# Greater than comparison  
greater\_than\_result = (array1 > array2)  
print("\nGreater Than Comparison Result:")  
print(greater\_than\_result)

Greater Than Comparison Result:  
[False False True False False]

# Less than comparison  
less\_than\_result = (array1 < array2)  
print("\nLess Than Comparison Result:")  
print(less\_than\_result)

Less Than Comparison Result:  
[ True True False True False]

## Array indexing and slicing

NumPy extends Python's list indexing notation using [] to multiple dimensions in an intuitive way. With NumPy, you can provide a comma-separated list of indices or ranges to select specific elements or subarrays, also known as slices, from a NumPy array. This powerful feature allows you to access and manipulate data within multi-dimensional arrays efficiently.

# Create a 2D NumPy array  
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])  
arr

array([[1, 2, 3],  
 [4, 5, 6],  
 [7, 8, 9]])

# Accessing individual elements  
element = arr[1, 2] # Accesses the element in the second row and third column  
print("Individual Element:", element)

Individual Element: 6

# Slicing subarrays  
subarray = arr[0:2, 1:3] # Selects a subarray consisting of rows 0 to 1 and columns 1 to 2  
print("\nSubarray:")  
print(subarray)

Subarray:  
[[2 3]  
 [5 6]]

# Using step in slicing  
step\_array = arr[:2, :2] # Selects rows from 0 to 1 and columns from 0 to 1  
print("\nStep Slicing:")  
print(step\_array)

Step Slicing:  
[[1 2]  
 [4 5]]

## Methods of np.arrays

NumPy arrays come equipped with a wide range of built-in methods that make performing various operations on arrays more convenient and efficient. These methods provide functionality for tasks such as mathematical operations, aggregation, statistics, and data manipulation.

To explore the extensive list of methods available for NumPy arrays, you can refer to the official [NumPy documentation](https://numpy.org/doc/stable/reference/arrays.ndarray.html). This documentation provides detailed information about each method along with examples of their usage.

# Creating a NumPy array  
arr = np.array([3, 1, 2, 5, 4])  
print(arr)  
  
# Method 1: Sorting the array  
sorted\_arr = np.sort(arr)  
print("Sorted Array:", sorted\_arr)  
  
# Method 2: Reshaping the array  
reshaped\_arr = arr.reshape(1, 5)  
print("Reshaped Array:", reshaped\_arr)

[3 1 2 5 4]  
Sorted Array: [1 2 3 4 5]  
Reshaped Array: [[3 1 2 5 4]]

# Method 3: Finding the maximum and minimum values  
max\_value = np.max(arr); print("Max Value:", max\_value);  
min\_value = np.min(arr); print("Min Value:", min\_value)  
  
# Method 4: Calculating the sum and mean  
sum\_values = np.sum(arr); print("Sum of Values:", sum\_values);  
mean\_value = np.mean(arr); print("Mean of Values:", mean\_value)  
  
# Method 5: Calculating the square root  
sqrt\_arr = np.sqrt(arr); print("Square Root of Array:", sqrt\_arr)  
  
# Method 6: Applying a mathematical function element-wise  
exp\_arr = np.exp(arr); print("Exponential of Array:", exp\_arr)  
  
# Method 7: Standard Deviation  
std = np.std(arr); print("Standard Deviation:", std)

Max Value: 5  
Min Value: 1  
Sum of Values: 15  
Mean of Values: 3.0  
Square Root of Array: [1.73205081 1. 1.41421356 2.23606798 2. ]  
Exponential of Array: [ 20.08553692 2.71828183 7.3890561 148.4131591 54.59815003]  
Standard Deviation: 1.4142135623730951

# Method 8: Finding unique values and their counts  
unique\_values, counts = np.unique(arr, return\_counts=True)  
print("Unique Values:", unique\_values)  
print("Counts:", counts)

Unique Values: [1 2 3 4 5]  
Counts: [1 1 1 1 1]

## Business challenge: E-commerce Customer Data Analysis

You work for an e-commerce company, and your manager has asked you to analyze customer data to gain insights into customer behavior. Your task is to calculate some key metrics and generate reports based on the provided data.

### Customer Data

You are provided with a NumPy array named customer\_data, which contains information about each customer. Each row in the array represents a customer, and the columns contain the following information:

1. Customer ID
2. Total Orders Placed
3. Total Amount Spent (in dollars)
4. Days Since Last Purchase
5. Email Subscribed (1 for subscribed, 0 for not subscribed)

Your goal is to create a set of functions to analyze this data and generate reports for your manager.

### Functions to Implement

You need to implement the following functions:

1. **average\_order\_value(data)**: Calculate the average order value (AOV) for all customers. AOV is calculated as the total amount spent divided by the total number of orders.
2. **customer\_lifetime\_value(data)**: Calculate the customer lifetime value (CLV) for each customer. CLV is calculated as the product of the average order value and the number of days since the last purchase.
3. **high\_value\_customers(data, threshold)**: Identify high-value customers whose CLV exceeds a given threshold.
4. **email\_subscription\_rate(data)**: Calculate the email subscription rate, which is the percentage of customers who have subscribed to emails.
5. **inactive\_customers(data, days)**: Identify customers who have not made a purchase in a specified number of days.

### Instructions

You are provided with a NumPy array named customer\_data containing customer information. Your task is to implement the functions mentioned above to perform the following tasks:

* Calculate the average order value (AOV) for all customers.
* Calculate the customer lifetime value (CLV) for each customer.
* Identify high-value customers whose CLV exceeds a given threshold. (Take 300 as threshold)
* Calculate the email subscription rate.
* Identify customers who have not made a purchase in a specified number of days. (take 30 days)

You should test your functions with the provided data and display the results in a clear and organized manner.

Feel free to use NumPy, lambda functions, list comprehension, and any other Python techniques you find suitable for this analysis.

**Note**: You can assume that the customer\_data array is already loaded with data.

import numpy as np  
  
# Sample customer data (replace with your actual data)  
customer\_data = np.array([  
 [101, 5, 250.00, 30, 1],  
 [102, 3, 150.00, 15, 0],  
 [103, 7, 350.00, 45, 1],  
 [104, 2, 100.00, 60, 0],  
 [105, 4, 200.00, 10, 1]  
])  
  
# Function to calculate the average order value (AOV)  
def average\_order\_value(data):  
 #[your code here]  
 return aov  
  
# Function to calculate the customer lifetime value (CLV)  
def customer\_lifetime\_value(data):  
 # Explanation of the CLV formula:  
 # - data[:, 2] retrieves the column of total amount spent by each customer.  
 # - data[:, 3] retrieves the column of days since the last purchase for each customer.  
 # - data[:, 1] retrieves the column of the number of orders made by each customer.  
 # - The formula (data[:, 2] / data[:, 3]) \* data[:, 1] calculates the average amount spent per day  
 # and multiplies it by the number of orders to estimate the CLV for each customer.  
 # - The result is an array containing the CLV for each customer in the same order as the input data.  
 #[your code here]  
 return clv  
  
# Function to identify high-value customers above a threshold CLV  
def high\_value\_customers(data, threshold):  
 #[your code here]  
 return high\_value  
  
# Function to calculate the email subscription rate  
def email\_subscription\_rate(data):  
 #[your code here]  
 return subscription\_rate  
  
# Function to identify inactive customers based on days since last purchase  
def inactive\_customers(data, days):  
 #[your code here]  
 return inactive  
  
#[more code here]

TOMORROW'S RITUAL

## Fun things that Numpy allows us to do

Numpy is a versatile library that offers a wide range of functionalities beyond numerical computing. It opens up opportunities for various data manipulation tasks, including image processing. In this section, we'll explore some exciting things that you can do with Numpy, from analyzing images to other fun applications.

### Exploring Numpy's Versatility

To discover the myriad of functionalities that Numpy offers, you can use the dir(object) function, which displays all possible methods that can be applied to a Numpy object. This allows you to explore the extensive capabilities of Numpy for different data manipulation tasks.

### Image Processing with Pillow and scikit-image

In addition to numerical data, Numpy can be used to work with images through libraries like Pillow and scikit-image:

* [Pillow](https://pillow.readthedocs.io/en/stable/): Pillow, the Python Imaging Library, provides comprehensive image processing capabilities for your Python interpreter. You can open and manipulate images with ease using functions like Image.open("path").
* [scikit-image](https://scikit-image.org/docs/stable/): Scikit-image extends your image processing capabilities in Python. With functions like io.imread("path"), you can read and process images efficiently for various applications.

In the upcoming sections, we'll delve into practical examples and explore the fun side of Numpy, demonstrating its versatility and usefulness in diverse data-related tasks.

We can process images with Numpy!

#!pip install scikit-image  
from skimage import io  
photo = io.imread("images/lisbon.png")  
numpy\_info(photo) # you might need to re run the notebook  
  
photo

The dimension is: 3  
The shape is: (407, 612, 3)  
The size is: 747252

array([[[206, 239, 244],  
 [206, 239, 244],  
 [205, 240, 244],  
 ...,  
 [ 86, 99, 69],  
 [ 34, 78, 45],  
 [ 12, 86, 51]],  
  
 [[206, 239, 244],  
 [206, 239, 244],  
 [205, 240, 244],  
 ...,  
 [ 56, 77, 60],  
 [ 48, 76, 62],  
 [ 9, 40, 34]],  
  
 [[206, 239, 244],  
 [206, 239, 244],  
 [205, 240, 244],  
 ...,  
 [ 5, 34, 3],  
 [ 0, 17, 0],  
 [ 21, 30, 13]],  
  
 ...,  
  
 [[225, 172, 132],  
 [209, 160, 119],  
 [220, 177, 134],  
 ...,  
 [202, 101, 49],  
 [210, 109, 57],  
 [218, 118, 66]],  
  
 [[202, 144, 106],  
 [212, 160, 120],  
 [241, 196, 154],  
 ...,  
 [187, 80, 64],  
 [184, 78, 62],  
 [184, 78, 62]],  
  
 [[188, 130, 92],  
 [136, 83, 43],  
 [131, 86, 44],  
 ...,  
 [193, 89, 86],  
 [179, 77, 73],  
 [170, 68, 64]]], dtype=uint8)

# pixel: square w/color  
# color: [r, g, b]  
# [0, 255, 89]

RGB cube

exploring the picture numerically

photo.shape[0] # FIRST DIMENSION: 800, height, y axis  
photo.shape[1] # SECOND DIMENSION: 1200, length, x axis  
photo.shape[2] # THIRD DIMENSION: b3, color, (r, g, b) -> depth  
  
print(f'First dimension is {photo.shape[0]}, the second dimension is {photo.shape[1]} and the third dimension is {photo.shape[2]}')

First dimension is 407, the second dimension is 612 and the third dimension is 3

# Color grading  
photo[0][0]   
  
# RED: photo[0][0][0] is 227  
# GREEN: photo[0][0][1] is 208  
# BLUE: photo[0][0][2] is 201

array([206, 239, 244], dtype=uint8)

pic = io.imshow(photo)



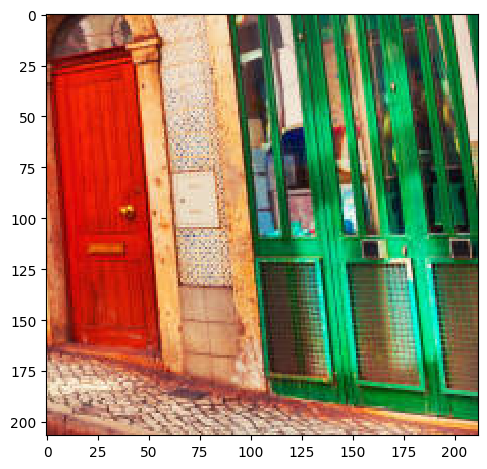
### 1. reversing a picture

# Do you remember we can reverse an array with array[::-1]? Let's try it  
reversed\_pic = io.imshow(photo[::-1])



1. cropping a picture

# We can also select a specific part  
cathedral = io.imshow(photo[200:500, 400:800])



1. changing color

# All dimensions but only get the blue   
 # Every element in Y axis  
 # Every element in X axis  
 # Only blue element in Blue   
io.imshow(photo[:, :, 2]);



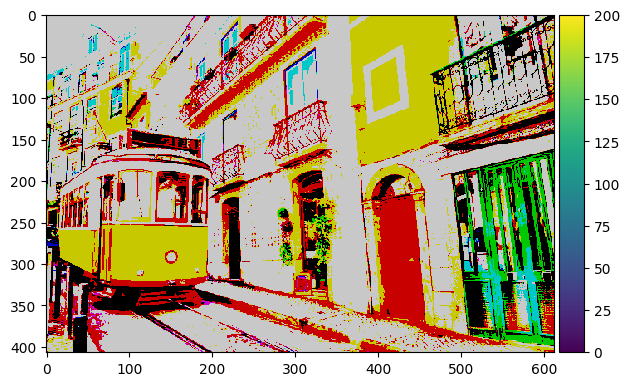
1. np.where

# np.where: how it works  
# https://numpy.org/doc/stable/reference/generated/numpy.where.html  
  
# a new array (filter\_2)  
# will be photo with updated values (np.where)  
  
  
# if the value is greater than 100  
 # that value will be changed int0 255  
# if not greater:   
 # the value will be zero  
  
# filter\_2 the result of updating based on the np.where condition  
 # np.where access every single element

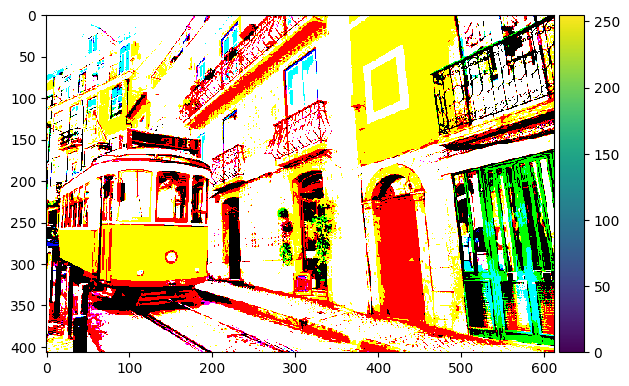
* 1. brightness: high

# Let's test with 200  
brightness\_high = np.where(photo > 100, 200, 0) # brightness  
io.imshow(brightness\_high);

/Users/joaorochaemelo/code/IH/venv\_ironhack/lib/python3.12/site-packages/skimage/io/\_plugins/matplotlib\_plugin.py:158: UserWarning: Low image data range; displaying image with stretched contrast.  
 lo, hi, cmap = \_get\_display\_range(image)

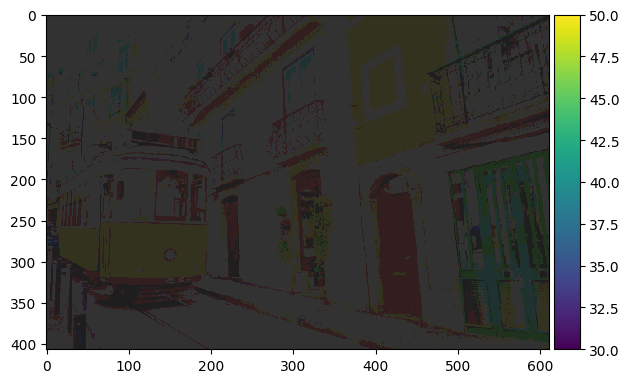


# Let's test with 255  
brightness\_high = np.where(photo > 100, 255, 0) # brightness  
io.imshow(brightness\_high);



* 1. brightness: low

# Let's test with 50  
brightness\_low = np.where(photo > 70, 50, 30)  
io.imshow(brightness\_low);



RGB cube

A digital image is decomposed on our computer as a 3-dimensional array where each pixel is a value from 0 to 255, with 0 being black and 255 being white.

## Some methods:

* [Multiply](https://numpy.org/doc/stable/reference/generated/numpy.multiply.html)
* [Reshape](https://numpy.org/doc/stable/reference/generated/numpy.reshape.html)
* [Transpose](https://numpy.org/doc/stable/reference/generated/numpy.transpose.html)

## Summary

It's your turn. What have we learned today?

## Further materials

* [NumPy Cheatsheet](https://s3.amazonaws.com/assets.datacamp.com/blog_assets/Numpy_Python_Cheat_Sheet.pdf)
* [Master Numpy](https://medium.com/analytics-vidhya/master-numpy-in-45-minutes-74b2460ecb00)
* [Numpy Tricks](https://github.com/patidarparas13/Numpy-Tricks/blob/master/Numpy%2BTricks(Zero%2Bto%2BHero).ipynb)
* [101 Numpy exercises](https://www.machinelearningplus.com/python/101-numpy-exercises-python/)

### Solutions

# Business challenge: Forex Data Analysis challenge  
import numpy as np  
  
# Sample Forex data (you can replace this with your dataset)  
dates = np.array(['2023-09-01', '2023-09-02', '2023-09-03', '2023-09-04', '2023-09-05'])  
exchange\_rates = np.array([1.15, 1.16, 1.14, 1.17, 1.15])  
  
# 1. Calculating the average exchange rate for the entire month  
average\_rate = np.mean(exchange\_rates)  
print(f"Average Exchange Rate for the Month: {average\_rate:.2f}")  
  
# 2. Identifying the date with the highest exchange rate  
max\_rate\_index = np.argmax(exchange\_rates)  
max\_rate\_date = dates[max\_rate\_index]  
max\_rate = exchange\_rates[max\_rate\_index]  
print(f"Highest Exchange Rate Date: {max\_rate\_date}, Rate: {max\_rate:.2f}")  
  
# 3. Determining the total trading volume for the month (assuming equal volume each day)  
total\_volume = np.sum(exchange\_rates) # Replace this with actual trading volume data if available  
print(f"Total Trading Volume for the Month: {total\_volume:.2f}")  
  
# 4. Detecting significant fluctuations or anomalies in the exchange rate (custom logic)  
# You can implement custom logic here to detect fluctuations or anomalies based on your criteria.  
  
# For example, identifying days where the rate is significantly lower than the average:  
anomalies = dates[exchange\_rates < (average\_rate - 0.1)] # Adjust the threshold as needed  
print("Dates with Significant Rate Anomalies:")  
for date in anomalies:  
 print(date)

Average Exchange Rate for the Month: 1.15  
Highest Exchange Rate Date: 2023-09-04, Rate: 1.17  
Total Trading Volume for the Month: 5.77  
Dates with Significant Rate Anomalies:

# Business challenge: E-commerce Customer Data Analysis  
import numpy as np  
  
# Sample customer data (replace with your actual data)  
customer\_data = np.array([  
 [101, 5, 250.00, 30, 1],  
 [102, 3, 150.00, 15, 0],  
 [103, 7, 350.00, 45, 1],  
 [104, 2, 100.00, 60, 0],  
 [105, 4, 200.00, 10, 1]  
])  
  
# Function to calculate the average order value (AOV)  
def average\_order\_value(data):  
 total\_orders = data[:, 1].sum()  
 total\_amount\_spent = data[:, 2].sum()  
 aov = total\_amount\_spent / total\_orders  
 return aov  
  
# Function to calculate the customer lifetime value (CLV)  
def customer\_lifetime\_value(data):  
 clv = (data[:, 2] / data[:, 3]) \* data[:, 1]  
 return clv  
  
# Function to identify high-value customers above a threshold CLV  
def high\_value\_customers(data, threshold):  
 clv = customer\_lifetime\_value(data)  
 high\_value = data[clv > threshold]  
 return high\_value  
  
# Function to calculate the email subscription rate  
def email\_subscription\_rate(data):  
 subscribed\_count = data[:, 4].sum()  
 total\_customers = len(data)  
 subscription\_rate = (subscribed\_count / total\_customers) \* 100  
 return subscription\_rate  
  
# Function to identify inactive customers based on days since last purchase  
def inactive\_customers(data, days):  
 inactive = data[data[:, 3] > days]  
 return inactive  
  
# Test the functions and display results  
print("Average Order Value (AOV): $", average\_order\_value(customer\_data))  
print("\nCustomer Lifetime Value (CLV):")  
print(customer\_lifetime\_value(customer\_data))  
print("\nHigh-Value Customers (CLV > $300):")  
print(high\_value\_customers(customer\_data, 300))  
print("\nEmail Subscription Rate: %", email\_subscription\_rate(customer\_data))  
print("\nInactive Customers (No purchase in > 30 days):")  
print(inactive\_customers(customer\_data, 30))

Average Order Value (AOV): $ 50.0  
  
Customer Lifetime Value (CLV):  
[41.66666667 30. 54.44444444 3.33333333 80. ]  
  
High-Value Customers (CLV > $300):  
[]  
  
Email Subscription Rate: % 60.0  
  
Inactive Customers (No purchase in > 30 days):  
[[103. 7. 350. 45. 1.]  
 [104. 2. 100. 60. 0.]]