Tecnológico de Monterrey

Contents

NumPy Tutorial

Version 1.0.0 - February, 2026. Monterrey

Author: Flavio F. Contreras-Torres. Tecnológico de Monterrey.

INTRO

This notebook is a **hands-on introduction** to numerical computing with NumPy and to some essential programming practices in Python. It is designed for beginners and for learners with prior experience who want to refresh or strengthen their understanding in how to work with arrays, perform basic linear algebra operations, and write more reliable code.

The notebook is divided into three main parts:

- **Part 1: Introduction to NumPy.** You will learn how to create and manipulate NumPy arrays, and how to use them for tasks such as vector and matrix operations, linear algebra, and polynomial multiplication.

- **Part 2: Introduction to Assert Statements and Testing.** You will learn how to use assert statements to test your code, check that your functions work as expected, and catch errors early.
- **Part 3: Debugging Your Code.** You will learn basic strategies for finding and fixing bugs in your programs, and how to use error messages and test cases to improve your code.

How to use this tutorial

This tutorial is designed to be completed in approximately **6 to 7 hours**, depending on your background and how much time you spend experimenting with the code and solving the exercises.

To reach the full training goal, you are encouraged to work through the material step-by-step. The practical core of this tutorial is **composed of two essential workbooks** that you will run directly in **Google Colab**:

1. `numpy_python_basics.ipynb`: A foundational review to ensure your Python skills are ready for data science.
2. `numpy_tutorial_exercises.ipynb`: A deep dive into specialized NumPy operations and array logic.

   Submission and Grading: Please note that these two notebooks will be evaluated as **official assignments**. Once you have completed all the exercises, you must submit your final versions through the **Canvas** platform for grading.

By using Google Colab, you can run these files in your browser **without any local setup**. Read the explanations, run the code cells, and try to modify the examples to see what happens. When you reach an exercise, take your time to think about the problem and attempt a solution before looking at any hints or solutions.

You are expected to **solve these exercises yourself** by writing Python and NumPy code. The explanations provided earlier will give you the tools you need, but the real learning happens when you **try**, **test**, **make mistakes**, and **fix them**.

If you get stuck, use the testing and debugging sections to help you understand what went wrong rather than skipping ahead. The goal is not just to finish the notebook, but to build confidence in using NumPy and in writing, testing, and debugging your own Python code.

Sources and Learning Materials

This tutorial is not meant to be the first or the last resource you will use to learn NumPy and scientific computing in Python. Instead, it is a **curated learning path** built from several tutorials, lecture notes, exercises, and official documentation.

Many of the ideas, examples, and exercises presented here are **inspired by or adapted from** existing educational materials, including the NumPy documentation and common teaching resources used in courses and online tutorials. For this reason, you may notice that some exercises or examples look **similar to ones you have seen elsewhere**. This is intentional: these problems are standard, well-tested ways of learning core concepts.

The goal of this tutorial is not to present completely new material, but to **organize and connect** these concepts in a coherent, progressive way, with explanations, practice exercises, testing, and debugging techniques all in one place.

You are encouraged to complement this tutorial with other resources, such as:

- The official NumPy documentation
- Free course notes, books, and lecture materials Mathematics for Machine Learning, Wolfram MathWorld
- Online tutorials 3blue1brown

Learning works best when you see the same ideas explained in **multiple ways and from multiple sources**.

PART 1. Introduction to NumPy

NumPy and SciPy are two core libraries for scientific computing in Python, but they serve different roles. In particular, NumPy (**Numerical Python**) is the core library for numerical and scientific computing in Python. It provides the fundamental data structure *the array* and the basic numerical operations needed to work efficiently with numerical data. This way, Numpy provides an efficient multidimensional array structure that enables fast, vectorized numerical operations, forming the foundation of most scientific and data-driven workflows in the Python ecosystem. In addition, NumPy is built for performance: it allows large datasets to be processed without explicit Python loops, achieving speeds comparable to compiled languages such as C and Fortran. For this reason, NumPy also serves as a foundation for many other libraries, such as pandas (data analysis), scikit-learn (machine learning), and frameworks like TensorFlow and PyTorch, which rely on NumPy-style arrays for their internal computations.

On the other hand, SciPy (**Scientific Python**) is built on top of NumPy and extends it with a large collection of ready-to-use scientific algorithms, including tools for optimization, numerical integration, statistics, signal processing, and linear algebra. Because SciPy operates directly on NumPy arrays, it is essential to understand NumPy first before using SciPy effectively.

In this tutorial, we will focus exclusively on **NumPy** and use it in the simplest possible way, without relying on higher-level libraries such as pandas, scikit-learn, TensorFlow, or PyTorch. This approach allows us to concentrate on the core concepts—arrays, basic operations, and numerical computing—so you can build a solid foundation before moving on to more advanced tools.

In particular, this tutorial will focus on the basic concepts of **linear algebra**, such as vectors, matrices, and the operations between them. In this context, NumPy provides reliable implementations of common matrix operations, as well as routines for eigenvalue and eigenvector calculations, covariance matrices, and affine transformations. These tools are widely used in areas such as machine learning, dimensionality reduction, and mathematical modeling.

The documentation of **NumPy** is extensively referenced and is available at the official webpage.

1.1 Python Basics

The tokens `+`, `-`, and `*`, and the use of parentheses for grouping, mean in Python what they mean in mathematics. The asterisk (`*`) is the token for multiplication, and `**` is the token for exponentiation. Addition, subtraction, multiplication, and exponentiation all do what you expect.

The Python interpreter can compute new values with function calls. There you might define a function `a` or `f` by specifying how it transforms an input into an output, for instance, `f(x) = 3x + 2`. Functions are like factories that take in some material, do some operation, and then send out the resulting object.

Additionally, if we want to see the results of the computation, the program needs to specify that with the function `print()`.

```
# Step 1. Define input data
a = 20 + 30
b = 5 ** 2
c = (5 + 9) * (15 - 7)
d = 16 - 2 * 5 // 3 + 1

# Step 2. Show the results
print(a)
print(a+b)
print(a-b+c)
print(d)
```

```
52
77
139
14
```

Remember that when more than one operator appears in an expression, the order of evaluation depends on the rules of precedence. Python follows the same precedence rules for its mathematical operators that mathematics does. For the last example, there is:

```
print((16 - ((2 * 5) // 3)) + 1)
```

```
14
```

In Python 3, the division operator `/` produces a floating point result (*even if the result is an integer*; `4/2` is `2.0`). If you want truncated division, which ignores the remainder, you can use the `//` operator (for example, `5//2` is 2).

```
# Step 1. Define input data
a = 9 / 5
b = 5 / 9
c = 9 // 5
d = 18.0 // 4

# Step 2. Show the results
print(a)
print(b)
print(c)
```

```
1.8
0.5555555555555556
1
4.0
```

The modulus operator, sometimes also called the remainder operator or integer remainder operator works on integers (and integer expressions) and yields the remainder when the first operand is divided by the second. In Python, the modulus operator is a percent sign (%). The syntax is the same as for other operators.

```
# Step 1. Define input data
a = 7 // 3
b = 7 % 3

# Step 2. Show the results
print(a)
print(b)
```

```
2
1
```

```
print(square(3))
print(sub(6, 4))
print(sub(5, 9))
print(square(3) + 2)
print(sub(square(3), square(1+1)))
```

```
9
2
-4
11
5
```

Notice that when a function takes more than one input parameter, the inputs are separated by a comma. However, when you type a large integer, you might be tempted to use commas between groups of three digits, as in `42,000`. This is **not a legal** integer in Python, but it does mean something else, which is legal:

```
print(42500)
print(42,500)
```

```
42500
42 500
```

Remember not to put commas or spaces in your integers, no matter how big they are. Additionally, if you are not sure what class (data type) a value falls into, Python has a function called type which can tell you.

```
print(type("Hello, World!"))
print("Hello, World")
print(type(17))
print(type(3.2))
print(type("17"))
print(type("3.2"))
```

```
<class 'str'>
Hello, World
<class 'int'>
<class 'float'>
<class 'str'>
<class 'str'>
```

Notice that strings in Python can be enclosed in either single quotes (') or double quotes ("), or three of each (''' or """)

```
print(type('This is a string.'))
print(type("And so is this."))
print(type("""and this."""))
print(type('''and even this...'''))
```

```
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
```

Triple quoted strings can even span multiple lines, such as:

```
print("""This message will span
several lines
of the text.""")
```

```
This message will span
several lines
of the text.
```

Sometimes it is necessary to convert values from one type to another. Python provides a few simple functions that will allow us to do that. The functions **`int`**, **`float`** and **`str`** will convert their arguments into types **`int`**, **`float`** and **`str`** respectively.

The **`int`** function can take a floating point number or a string, and turn it into an **`int`**. For example:

```
print(3.14, int(3.14))
print(3.0, int(3.0))
print("2345", int("2345"))
```

```
3.14 3
3.0 3
2345 2345
```

The type converter **`float`** can turn an integer, a float, or a syntactically legal string into a float:

```
print(float("123.45"))
print(type(float("123.45")))
```

```
123.45
<class 'float'>
```

The type converter **str** turns its argument into a string. Remember that when we print a string, the quotes are removed in the output window. However, if we print the type, we can see that it is definitely **str**.

```
print(str(17))
print(str(123.45))
print(type(str(123.45)))
```

```
17
123.45
<class 'str'>
```

1.2 Using Google Colab for This Tutorial

If you've ever wanted to write code without the headache of software installation, **Google Colab** is the perfect place to start. It's a cloud-based platform that lets you execute Python directly in your browser. We'll use it throughout this tutorial so you can focus entirely on mastering NumPy instead of troubleshooting your computer setup. Think of it as **Google Docs, but for programmers**.

What is Google Colab?

Beyond just avoiding installation, Google Colab provides an interactive environment called a **Notebook**. These notebooks are powerful because they allow you to combine:

- Executable Code: Write and run Python instantly.
- Rich Documentation: Add explanations, notes, and instructions using Markdown.
- Visuals: Embed images, HTML, and even complex mathematical formulas using LaTeX.

Because it's integrated with the Google ecosystem, your progress saves automatically to **Google Drive**. You can share your work as easily as a link, allowing others to view, comment, or collaborate in real-time. All you need is a browser and an internet connection to get started.

Why Use Colab?

Since we've already seen that Colab is cloud-based and requires zero setup, here are the specific reasons it's the best choice for learning **NumPy**:

- **Pre-installed Libraries**: You don't need to worry about pip install. Essential libraries like NumPy, Pandas, and Matplotlib are already configured and ready to use.
- **Google's Computing Power**: Your code runs on Google's high-performance servers. This means your computer's battery and RAM stay fresh while the cloud handles the heavy lifting.
- **Hardware Acceleration**: If our data tasks get complex, you can toggle on a GPU for free to speed up calculations—something that usually requires expensive hardware.
- **Cell-Based Learning**: Unlike a traditional script, the Interactive Format allows you to run code in small "cells." You can test one line of NumPy at a time, see the result immediately, and move on.

- **Seamless Collaboration**: Just like any other Google Workspace tool, you can share your notebooks with a link to get help or show off your progress.

Colab and Jupyter Notebooks

If you've heard of **Jupyter Notebooks**, you're already one step ahead. Google Colab is essentially a cloud-hosted version of the Jupyter technology.

- **Same DNA**: Colab uses the standard .ipynb file format. This means any notebook you create here can be downloaded and opened in a local Jupyter environment later.
- **The Main Difference**: While traditional Jupyter requires you to manage your own Python "environments" and local installations, Colab provides a pre-configured workspace on Google's infrastructure.
- **Why start here?** We are using Colab today to remove technical barriers, allowing you to focus entirely on NumPy logic.

In **Table 1**, a comparison is shown between these two environments to help you decide which setup best suits your future projects.

Table 1: Key Differences Between Google Colab and Local Jupyter Notebooks

Feature	Google Colab	Jupyter Notebook
Setup	No installation requiered (cloud-based)	Manual Python and Jupyter installation (local)
Internet	Required connection	Works offline
Hardware	Uses Google's servers	Uses your own computer
Environment	Pre-installed popular libraries	Customizable environments
Collaboration	Live sharing and commenting (like Google Docs)	Shared via files (*.ipynb)

Looking Ahead: In future tutorials, we will explore how to run Jupyter locally on your own machine. This will give you more control over offline access and custom configurations as your projects become more advanced. For now, Colab is the fastest way to get your hands on the code.

In this first practical session, we will be using the notebook `numpy_python_basics.ipynb`.

Our goal is to review the core concepts from **Subsection 1.1: Python Basics**. While you may have seen these concepts before, running them inside Google Colab will help you get comfortable with the interactive "cell" environment before we dive deep into NumPy.

1.3 Basic Structure of Codes Used in This Tutorial

In this tutorial, all code examples follow a simple and consistent structure. First, we import the required **libraries**. To work with numerical data in Python, we first import the **NumPy** library. By convention, NumPy is imported using the abbreviation `np`, which makes the code shorter and easier to read.

```
1  import numpy as np      # standar NumPy import
```

After importing NumPy as `np`, we access its tools by writing `np.` followed by the name of a function or object. Then, we define the input data, usually as NumPy arrays. For example, in the next section we will see how to create arrays using the function call `np.array()`. Thus, the resulting **array** can be stored in a variable by assigning it a name, for instance, we can choose the letter `a` to refer to the array in `a = np.array(...)`.

```
import numpy as np      # required

x = [1,2,3]             # a Python list with some elements
a = np.array(x)         # convert the list into a NumPy array and store it as 'a'
```

In the next example, **addition** is already defined for NumPy arrays, so we just use the operator `+`.

```
import numpy as np      # required

# Step 1. Define input data
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Step 2. Perform an operation (already defined in NumPy)
c = a + b

# Step 3. Show the result
print(c)
```

Sometimes the mathematical expression we need is already available through existing operations, while in other cases we will write the equation explicitly in code. For instance, the equation $F = m \cdot a$ is not a built-in physics function, so we write it ourselves using the operator `*`.

Additionally, in many cases, we also define a function using **`def`**, which groups a set of operations under a name so it can be reused. Inside the function, we perform the necessary computations and use return to send back the result.

```
import numpy as np              # required

# Step 1. Define a function
def compute_force(mass, acceleration):
    """
    Compute the force using Newton's second law: F = m * a.

    Arguments:
    mass : number or numpy.ndarray
        The mass of the object(s).
    acceleration : number or numpy.ndarray
        The acceleration of the object(s).

    Returns:
    force : number or numpy.ndarray
        The computed force.
    """
    force = mass * acceleration
    return force
```

```python

# Step 2. Define input data
m = np.array([1.0, 2.0, 3.0])   # mass
a = np.array([9.8, 9.8, 9.8])   # acceleration

# Step 3. Call the function
F = compute_force(m, a)

# Step 4. Show the result
print(F)
```

> **Note:** When defining functions, it is important to include short comments or documentation strings (written between triple quotes `""" ... """`) that explain what the function does, what arguments it expects, and what it returns. These docstrings make the code easier to read, understand, and maintain, especially when functions are reused or shared with others.

Finally, the same idea, but with matrices (linear algebra style). In this case, we use the operator `@`.

```python
import numpy as np              # required

# Step 1. Define a function for matrix-vector multiplication
def apply_matrix(A, x):
    """
    Apply a matrix to a vector using matrix-vector multiplication.

    Arguments:
    A : numpy.ndarray
        A two-dimensional matrix.
    x : numpy.ndarray
        A one-dimensional vector.

    Returns:
    y : numpy.ndarray
        The resulting vector after multiplication.
    """
    y = A @ x
    return y

# Step 2. Define input data
A = np.array([[1, 2],
              [3, 4]])
x = np.array([1, 1])

# Step 3. Compute result
y = apply_matrix(A, x)

# Step 4. Show the result
print(y)
```

As you can see, **arrays** appear in all the examples, because they are the basic data structure used to represent vectors, matrices, and numerical data in NumPy. For this reason, before moving on to more advanced

topics, we need to understand how arrays are created, how they are shaped, and how operations are applied to them.

1.4 NumPy Arrays

A NumPy **array** is a data structure used to store numbers in an organized, grid-like form (such as a list, table, or matrix) so they can be processed efficiently.

Unlike Python lists, NumPy arrays store **elements of the same type** and allow fast mathematical operations on all values at once, which makes them ideal for numerical computing and scientific applications.

New arrays can be created in several ways. One simple method is to start from a Python list (a basic collection of values) and convert it into a NumPy array:

```
import numpy as np      # required; omitted in the following exercises

a = np.array([1,2,3])   # This is a one-dimensional array
print(a)
```

```
[1 2 3]
```

> **Note:** `np.array` is a **function** in NumPy. The parentheses `( )` are used to call this function, while the square brackets `[ ]` define the data being passed to it.

```
b = np.array([[1, 2, 3],    # This is a two-dimensional array
              [4, 5, 6],
              [7, 8, 9]])
print(b)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

> **Note:** For 2-dimensional Numpy arrays, all rows must have the same number of elements.

NumPy arrays can be created from both Python **lists** and **tuples**. A list `[1, 2, 3]` and a tuple `(1, 2, 3)` differ in Python because lists can be modified while tuples cannot. For instance, a **list** can be a shopping list, a to-do list of tasks, the name of students in a class, books you plan to read in this year. On the other hand, examples of **tuples** can be the days of the week, the months of the year, the coordinates of a point (x,y,z), a person's date of birth (day, month, year), the dimension of a rectange (width, height).

However, when either one is passed to `np.array()`, NumPy copies the values into a **new NumPy array**. After this conversion, the original list or tuple is no longer relevant, and the resulting NumPy array behaves exactly the same in both cases. In other words, the difference matters before creating the NumPy array (list vs tuple), but not after, because NumPy creates its own data structure.

```
lst = [1, 2, 3]     # Create a NumPy array from a list
a = np.array(lst)

```

```
tpl = (1, 2, 3)     # Create a NumPy array from a tuple
b = np.array(tpl)


print(a)
print(b)
```

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

The outputs are the printed representation of NumPy arrays.

In NumPy, every array has a data type, called `dtype`, which tells NumPy what kind of numbers the array stores.

```
a = np.array([1, 2, 3])
b = np.array([1.0, 2.0, 3.0])

print("Type of array in a:", a.dtype)
print("Type of array in b:", b.dtype)
```

```
Type of array in a: int64

Type of array: float64
```

> **Note:** NumPy arrays have a `dtype` attribute that controls the type of numbers they store (int, float, complex, etc.). In more advanced code, you may want to match the data type of different arrays explicitly.

In NumPy, we can use `np.array()` to create vectors and `np.dot()` to compute their dot product.

```
def length(x):
  """Compute the length of a vector"""
  length_x = np.sqrt(np.dot(x, x))  # using the relationship: ||x|| = sqrt(x·x)

  return length_x

print(length(np.array([1,0])))
```

```
1.0
```

1.5 Shape of Numpy Arrays

The **shape** of a NumPy array describes how many elements it has and how they are arranged. You can use the function np.shape to obtain the shape of a Numpy array.

The shape of NumPy arrays is printed as a tuple of numbers, where each number represents the size of one dimension. For example, a shape of `(3,)` means the array has 3 elements in one dimension. The comma is important because it tells Python that this is a tuple, not just a number.

```
a = np.array([1, 2, 3])
print(a.shape)
```

```
(3,)
```

> **Note:** Remember that tuples (writting always using parenthesis) are similar to lists, but they cannot be changed after they are created (i.e. you cannot add, remove, or replace elements). This property makes tuples useful for representing fixed information, such as the shape of a NumPy array.

```
b = np.array([[1], [2], [3]])
print(b.shape)
```

```
(3, 1)
```

For this example, the tuple `(3,1)` represents the **shape** of the NumPy array, meaning the array has 3 rows and 1 column. For two dimensional arrays, you can consider the first element of the tuple to be the number of rows and the second element to be the number of columns.

```
c = np.array([[1, 2],
              [3, 4],
              [5, 6]])
print(c.shape)
```

```
(3, 2)
```

For this example, the tuple `(3,2)` represents the **shape** of 3 rows and 2 columns. Note that the shape of an array is determined by how values are grouped with brackets, not by line breaks. Line breaks only improve readability and do not affect the array's structure.

As indicated, ehen working with NumPy arrays, `b.shape` returns a tuple describing dimensions:

```
b = np.array([1, 2, 3, 4, 5])
D = b.shape
print(D)
```

```
(5,)
```

However, to extract just the dimension value from this **1-element tuple**, we use **tuple unpacking** with a comma:

```
b = np.array([1, 2, 3, 4, 5])
D, = b.shape
print(D)
```

```
5
```

This is a Python idiom that's very common in scientific computing code. It makes the intention clear: "I'm expecting a 1-element tuple and I want that single value", in other words, the comma in `D,` signals for taking the single element from this tuple.

If A is a matrix (a two-dimensional NumPy array), the expression

```
n = A.shape[0]
```

means that we store the numbers of rows of A in a variable called n. Here, `A.shape` returns the size of the matrix as a pair:

```
(number of rows, number of columns)
```

Therefore, `A.shape[0]` selects the **first value** of that pair, which corresponds to the number of **rows**. Similarly, `A.shape[1]` selects the second value, which correspond to the number columns.

```
d = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])

print(d.shape)
print(d.shape[0])
print(d.shape[1])
```

```
(3, 3)
3
3
```

For this example, the tuple `(3,3)` represents the **shape** of 3 rows and 3 columns.

In a **matrix**, each element is identified by two indices: the **row index** and the **column index**. The principal diagonal runs from the top-left corner to the bottom-right corner of the matrix.

Along this diagonal, each element lies in the same row and column position. This means that the row index and the column index are equal. For this reason, every element on the principal diagonal has an index of the form (i, i), such as $(0, 0)$, $(1, 1)$, and so on.

This indexing pattern is what allows diagonal elements to be accessed easily in code using expressions like `A[i, i]`.

Creating Numpy arrays based on shape

NumPy allows you to create arrays by specifying their shape, **without providing the individual values explicitly**. This is useful when you need arrays of a certain size initialized with default values.

Common examples include arrays filled with **zeros**, **ones**, or **random numbers**, where the shape defines the number of rows and columns. You can use the functions np.zeros and np.ones to create Numpy arrays of a specified shape.

```
a = np.zeros((5,))
print(a)
```

```
[0. 0. 0. 0. 0.]
```

```
b = np.zeros((2, 3))
print(b)
```

```
[[0. 0. 0.]
 [0. 0. 0.]]
```

```
c = np.ones((3, 2))
print(c)
```

```
[[1. 1.]
 [1. 1.]
 [1. 1.]]
```

> **Note:** Remember that the input for these functions represent the shape of the Numpy arrays which you want to produce.

Reshaping arrays

Reshaping an array means changing its shape (dimensions) **without changing its data**. Reshaping is useful when you want to reorganize the same data into a different structure, such as converting a one-dimensional array into a matrix or modifying the number of rows and columns. A common use case is transforming one-dimensional NumPy arrays into two-dimensional row vectors or column vectors.

NumPy allows this using the function np.reshape to change the shape of a Numpy array.

Reshape is often used in conjunction with the function np.arange. The function `np.arange(x)` returns a one-dimensional array containing the values $[0, 1, \ldots, x-1]$.

To reshape an array, you specify the desired shape as a tuple. One value in the tuple can be set to `-1`, which tells NumPy to automatically determine the appropriate size for that dimension. An error will occur if more than one value is set to `-1`.

```
a = np.arange(3)            # Step 1: Create a one-dimensional NumPy array

print("Array a:")
print(a)

print("Shape of a:")
print(a.shape)
```

```
Array a:
[0 1 2]
Shape of a:
(3,)
```

The sequence starts at **0** because Python (and NumPy) use **zero-based indexing**. This means that counting begins at zero instead of one. Starting from zero makes it easier for the computer to keep track of positions and lengths of data. So, in this example, **0** is the first position, **1** is the second position, and **2** is the third position. Together, these numbers represent the first three positions in the array.

```
a = a.reshape((1, -1))      # Step 2: Reshaping it to a row vector

print("Reshaped array a:")
print(a)

print("New shape of a:")
print(a.shape)
```

```
Reshaped array a:
[[0 1 2]]
New shape of a:
(1, 3)
```

In the shape for the first output, `(3,)` the value `3` indicates the number of elements in the single dimension of the array. For the second output, the first value of the tuple is `1` because a row vector only has 1 row.

```
a = np.arange(3)
a = a.reshape((-1, 1))      # Reshaping it to a column vector

print("Reshaped array a:")
print(a)
print("New shape of a:")
print(a.shape)
```

```
Reshaped array a:
[[0]
 [1]
 [2]]
New shape of a:
(3, 1)
```

These two operations can be combined into a single line, since the `reshape()` method can be applied directly to the array created by `np.arange()`.

```
a = np.arange(3).reshape((-1, 1))
```

This produces the same column vector with shape `(3, 1)`.

Note: Remember that the operations are read from **left to right**. In the last example, `np.arange(3)` creates the array, and then `reshape((-1, 1))` is applied to that result. So, in general, chained NumPy operations are applied step by step, from left to right, with each operation acting on the result of the previous one.

```
a = np.arange(6)
print("A: Original")
print(a)
print(a.shape)

a = a.reshape((3, 2))
print("\nA: Reshaped\n")
print(a)
print("\n", a.shape)
```

```
A: Original
[0 1 2 3 4 5]
(6,)

A: Reshaped

[[0 1]
 [2 3]
 [4 5]]

(3, 2)
```

Note: Observe that the newline character `\n` (for instance, `print("\nA: Reshaped\n")`) is used to insert **line breaks** in printed output. It allows you to add blank lines to separate sections and improve readability without changing the content of the data being printed.

1.6 Accessing NumPy Arrays

Once we have a NumPy array, we often want to look at or use **individual values** inside it. This is called accessing an array. We already known that Numpy arrays are 0 indexed and each value in a NumPy array has a position, called an **index**. In Python, counting starts at zero, so the first value is at position 0, the second at position 1, and so on.

To access a value (i.e., element of a Numpy array), we write the name of the array followed by the index inside **square brackets** `[ ]`. If A is a one dimensional array, then you can use `A[i]` to access the i^{th} element of A. If A is a two dimensional array, then you can use `A[i, j]` to access element $a_{i,j}$, where i is the row and j is the column.

```
a = np.arange(4)

print(a)

print("The 2th element of A")
print(a[2])
```

```
[0 1 2 3]
The 2th element of A
2
```

```
a = np.arange(9).reshape(3, 3)

print(a)

print("\nThe 1th row of the array:")
print(a[1])

print("\nElement at 0th row and 0th column:")
print(a[0, 0])

print("\nElement at 2th row and 1th column:")
print(a[2, 1])
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]

The 1th row of the array:
[3 4 5]

Element at 0th row and 0th column:
0

Element at 2th row and 1th column:
7
```

Array slicing

Array **slicing** means selecting a part of an array instead of just one single value. It allows us to take several values at once by specifying a range of positions. Instead of asking for *one position*, slicing asks for a *range of positions*. Think of slicing like cutting a piece of cake: you choose where to start cutting and where to stop, so the slice includes everything in between.

Read more about the slicing notation in the documentation.

Slicing uses the format:

```
start : end
```

Here, `start` indicates where to begin (included), while `end` shows where to stop (not included).

Suppose A is a one dimensional array with elements at positions

```
Index:  0   1   2   3   4
Value:  A[0] A[1] A[2] A[3] A[4]
```

If you want to get all elements whose index is greater than or equal to 1 but less than 4, you use the slice `A[start:end]`:

```
A[1:4]
```

This selects the elements at positions:

```
1, 2, 3
```

The element at position **4** is not included, because the end index is always excluded.

If you want to get all elements of the array, you can write: `A[:]`. This means from the beginning to the end of the array.

For a two-dimensional array A, you can obtain the i^{th} column using: `A[:, i]`. This notation means: take all rows `(:)` from column `i`.

The result is returned as a **one-dimensional array**, even though it comes from a column. If you need the result to be treated as a column vector, it must be reshaped explicitly.

```
a = np.arange(16).reshape(4, 4)
print(a)

print("\nThe 1th column of the array:")
print(a[:, 1])

print("\nThe 3th column of the array reshaped into a column vector:")
print(a[:, 3].reshape(-1, 1))
```

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

The 1th column of the array:
[1 5 9 13]

The 3th column of the array reshaped into a column vector:
[[ 3]
 [ 7]
 [11]
 [15]]
```

A built-in Python function used to generate a sequence of numbers is range(). It is most commonly used when we want to repeat an action a certain number of times, especially in loops.

The number produced by `range()` start at **0** by default, increase by **1**, and stop **before** the final number.

```
range(stop)
```

This generates numbers starting from 0 up to, but not including, `stop`. For example, `range(5)` represents the list of numbers **0, 1, 2, 3, 4**.

The function `range()` can be used with arrays. If an array has length n, `range(n)` produces exactly the indices needed to access all its elements, that is, the first index 0 up to the last index $n - 1$.

```
a = np.arange(16).reshape(4, 4)
n = a.shape[0]

print(a)
print(range(n))
```

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

range(0, 4)
```

> **Note:** Even though it looks like a tuple, `range(0, 4)` is **NOT** a tuple. It is an object that represents a **sequence of numbers**, namely, start at 0, stop before 4.

Exercise 1: Find the trace of a matrix

The trace of a square matrix is defined as the sum of the elements on its main diagonal (from top-left to bottom-right).

For example, for the matrix

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

the trace is:

$$\text{trace}(A) = 1 + 4 = 5$$

In this exercise, you will implement a function that computes the **trace** without using NumPy's built-in trace functions.

Hint: Each element of the principal diagonal has an index of the form (i, i).

```
import numpy as np

def trace(A):
    """
    Compute the trace of a square matrix A

    Arguments:
```

```
    A : numpy.ndarray
        A two dimensional square matrix

    Returns:
    s : number
        The trace of A
    """

    # Initialize the sum of diagonal elements
    s = 0

    ### BEGIN SOLUTION

    # Step 1. Obtain the number of rows in A
    n = None        # Replace with the correct syntax to get the number of rows

    # Step 2. Loop over all elements of the principal diagonal
    for i in range(n):
        s += None   # Replace the None with the required element of A

    ### END SOLUTION

    return s
```

Verify your solutions `s` for this exercise by computing the trace manually using indexing and loops for the following matrices A. This helps reinforce how NumPy performs matrix operations under the hood.

```
A = np.array([[3, 2, 7],
              [4, 9, 0],
              [1, 8, 5]])
s = 17


A = np.array([[12, -2, 31, 18],
              [32, -77, -24, 19],
              [87, 93, -53, 13],
              [49, 81, 63, 70]])
s = -48
```

1.7 Operations on NumPy Arrays

NumPy does not treat arrays as single numbers. Instead, it applies the operation to each element inside the array. This means that NumPy allows you to perform mathematical operations directly on arrays. When you use operators such as '*', '+', '-', '**' and '/' on NumPy arrays, the operation is applied **element by element**.

```
a = np.array([[4, 1, 2],
              [7, 2, 3]])

b = np.array([[3, 6, 9],
              [7, 8, 2]])
```

```
print(a + b)
```

```
[[ 7  7 11]
 [14 10  5]]
```

```
print(a - b)
```

```
[[ 1 -5 -7]
 [ 0 -6  1]]
```

```
print(a * b)
```

```
[[12  6 18]
 [49 16  6]]
```

```
print(a / b)
```

```
[[1.33333333 0.16666667 0.22222222]
 [1.         0.25       1.5       ]]
```

```
print(a ** b)
```

```
[[    64      1    512]
 [823543    256      9]]
```

Comparison operators

There also exist other operators such as '==' (equal to), '`!=`' (not equal to), '>' (greater than), '<' (less than), '>=' (greater than or equal), '<=' (less than or equal), called comparison or relational operators. They returnn `true` or `false` depending on the comparison, for instance, `a != b` returns `True` if `a` and `b` are different, and returns `False` if `a` and `b` are the same.

```
a = np.array([1, 2, 3, 4])
b = np.array([1, 0, 3, 5])

print(a == b)
print(a != b)
print(a > b)
print(a <= b)
```

```
[True, False, True, False]
[False, True, False, True]
[False, True, False, False]
[True, False, True, True]
```

```
alpha = np.array([0.5, 0.0, 0.5, 0.0])
mask = alpha != 0  # Check each element: is it NOT equal to 0?
```

```
alpha = [0.5, 0.0, 0.5, 0.0]
Check: 0.5 != 0 → True
   0.0 != 0 → False
   0.5 != 0 → True
   0.0 != 0 → False
mask = [True, False, True, False]
```

`np.sqrt()`, `np.exp()`, and `np.log()`

Some other useful functions is Numpy are np.sqrt(), np.exp(), np.log() which apply the corresponding operation to every element of the inputted Numpy array.

```
print(np.sqrt(a))
```

```
[[2.         1.         1.41421356]
 [2.64575131 1.41421356 1.73205081]]
```

```
print(np.exp(a))
```

```
[[  54.59815003    2.71828183    7.3890561 ]
 [1096.63315843    7.3890561    20.08553692]]
```

```
print(np.log(a))
```

```
[[1.38629436 0.         0.69314718]
 [1.94591015 0.69314718 1.09861229]]
```

`np.mean()`

NumPy provides the function `np.mean()` to compute the arithmetic mean along the specified axis.

```
numbers = np.array([1, 2, 3, 4, 5])
average = np.mean(numbers)              # calculate the mean (average)

print(f"Numbers: {numbers}")
print(f"Mean: {average}")
```

```
Numbers: [1 2 3 4 5]
Mean: 3.0
```

In many data analysis techniques, we **center the data** by subtracting the mean of each feature. This shifts the data to have zero mean without changing its shape or relationships.

The data are centered for:

- Removes Bias: Eliminates systematic offsets so we focus on variation
- Improves Numerical Stability: Prevents large numbers that can cause computational issues
- Standard Starting Point: Many algorithms assume centered data
- Interpretability: Makes patterns easier to see and compare

```python
data = np.array([[10, 5],
                 [8, 4],
                 [12, 6],
                 [9, 5.5],
                 [11, 5.8]])

print("Original data:")
print(data)
print(f"\nMean of x: {np.mean(data[:, 0]):.2f}")
print(f"Mean of y: {np.mean(data[:, 1]):.2f}")

# CENTERING: Subtract column means
data_normalized = data - np.mean(data, axis=0)

print("\nCentered data (data - mean):")
print(data_normalized)
print(f"\nNew mean of x: {np.mean(data_normalized[:, 0]):.2f}")
print(f"New mean of y: {np.mean(data_normalized[:, 1]):.2f}")
```

```
Original data:
[[10.   5. ]
 [ 8.   4. ]
 [12.   6. ]
 [ 9.   5.5]
 [11.   5.8]]

Mean of x: 10.00
Mean of y: 5.26

Centered data (data - mean):
 [[ 0.    -0.26 ]
 [-2.    -1.26 ]
 [ 2.     0.74 ]
 [-1.     0.24 ]
 [ 1.     0.54 ]]

New mean of x: 0.00
New mean of y: 0.00
```

In this example, `axis=0` tells the computer: “For each column, look at all the numbers in that column and calculate their average.” So, `np.mean(data, axis=0)` is used to obtain one average for each column.

`np.pad()`

Sometimes, we need two NumPy arrays to have the **same length** before we can combine them in an operation. One simple way to do this is to **pad** (extend) the shorter array with extra values, such as zeros.

NumPy provides the function `np.pad()` for this purpose.

For example:

```
A = np.array([1, 2, 3])
B = np.array([4, 5])

# Pad B with one zero at the end so it has the same length as A
B_padded = np.pad(B, (0, 1))

print("A:", A)
print("B padded:", B_padded)
```

```
A: [1 2 3]
B padded: [4 5 0]
```

In this example, `(0, 1)` means: - Add 0 values at the beginning of the array - Add 1 value at the end of the array

This way, `np.pad()` allows us to extend an array with zeros (or other values) to make its size match another array. This is useful in polynomial multiplication when the two coefficient arrays do not have the same length.

`np.flip()`

Additionally, it is useful to **reverse the order** of elements in a NumPy array. For example, you may want to turn:

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

into:

```
[4, 3, 1, 1]
```

NumPy provides the function `np.flip()` to do exactly this. Here is a simple example:

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

A_reversed = np.flip(A)

print("Original array:", A)
print("Reversed array:", A_reversed)
```

```
Original array: [1 2 3 4]
Reversed array: [4 3 2 1]
```

So, `np.flip(A)` returns a new array with the same elements as A but in reverse order. This function is useful in polynomial multiplication because reversing one of the coefficient arrays can help align terms correctly when computing the sums of products.

`np.where()`

It is a selection/conditional function that uses boolean arrays but doesn't create them. This function takes boolean arrays as input and returns indices or selected values as output.

```
np.where(condition, x, y)
```

The syntax returns such elements from `x` where `condition` is `True` and elements from `y` where `condition` is `False`.

```
idx = np.where(alpha != 0)[0]
```

Think of a classroom, then

- `alpha != 0` indicates which students raised their hand? -> Boolean mask
- `np.where(alpha != 0)` means what are the seat numbers of students who raised hands? -> Indices
- `[0]` is for give me just the list of seat numbers -> First element of tuple

Remember that `np.where()` does not return the mask itself but it returns where the True values are located in the mask

```
alpha = np.array([0.5, 0.0, 0.5, 0.0])
y = np.array([+1, +1, -1, -1])

# Method 1: Get mask, then indices
mask = alpha != 0
print("mask = alpha != 0:")
print(f"  mask = {mask}")

sv_idx = np.where(mask)[0]
print(f"\nsv_idx = np.where(mask)[0]:")
print(f"  sv_idx = {sv_idx}")

sv_labels = y[sv_idx]
print(f"\nsv_labels = y[sv_idx]:")
print(f"  sv_labels = {sv_labels}")

# Method 2: Direct boolean indexing (no indices needed)
sv_labels_direct = y[alpha != 0]
print(f"\nsv_labels_direct = y[alpha != 0]:")
print(f"  sv_labels_direct = {sv_labels_direct}")
```

```
mask = alpha != 0:
    mask = [ True False  True False]

```

```
sv_idx = np.where(mask)[0]:
    sv_idx = [0 2]

sv_labels = y[sv_idx]:
    sv_labels = [ 1 -1]

sv_labels_direct = y[alpha != 0]:
    sv_labels_direct = [ 1 -1]
```

Broadcasting

Broadcasting is a rule that allows NumPy to perform operations between arrays of different shapes. Instead of requiring arrays to have exactly the same size, NumPy automatically adjusts the smaller array so that the operation can be carried out element-wise.

Using broadcasting, Numpy arrays that are not of the same dimension can be strected/duplicated so that they are of the same dimenson.

To know more about broadcasting, you can read the documentation.

The simplest example for this is when you have to multiply all elements of a Numpy array with a scaler or add a scaler to all elements of the array. In this case, broadcasting means *use the same value multiple times so that the operation can be applied to all elements*. Imagine adding the same amount of salt to every bowl of soup. You don't need a different spoon for each bowl — you reuse the same one. Then, the regular element-wise operation is applicable.

```
print(a * 2)
```

```
[[ 8  2  4]
 [14  4  6]]
```

```
print(a + 2)
```

```
[[6 3 4]
 [9 4 5]]
```

Broadcasting lets a smaller array act as if it were expanded to match the shape of a larger array, so that element-wise operations can be performed. This way, broadcasting can be applied during operation between two arrays of different shapes. Suppose that a is an array of shape $(2, 3)$, while c has shape $(1, 3)$. When an operation is performed between a and c, NumPy automatically applies broadcasting to make the shapes compatibles.

Conceptually, you can think of the array c as being **repeated along the row axis**, so that it behaves as if it had shape $(2, 3)$. This allows the operation to be carried out element-wise between the two arrays. This duplication is conceptual only — NumPy does not actually copy the data in memory.

```
c = np.array([5, 3, 4]).reshape((1, 3))

```

```
print('a:\n', a)
print('c:\n', c)
print('a + c:\n', a + c)
```

```
a:
 [[4 1 2]
 [7 2 3]]
c:
 [[5 3 4]]
a + c:
 [[ 9  4  6]
 [12  5  7]]
```

NumPy applies broadcasting automatically. No additional function or special command is needed. When you perform an operation between arrays, NumPy checks their shapes and applies broadcasting **by default if the shapes are compatible**.

You will get an error if you try to perform an element-wise operation on two arrays whose shapes are not compatible. For element-wise operations, NumPy requires the arrays to be broadcastable, meaning their dimensions must match or follow specific broadcasting rules.

```
a = np.arange(6).reshape((2, 3))
print(a)

b = np.arange(4).reshape((2, 2))
print(b)

print(a + b)
```

```
[[0 1 2]
 [3 4 5]]
[[0 1]
 [2 3]]



---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

ValueError: operands could not be broadcast together with shapes (2,3) (2,2)
```

Now you will write a function which makes use of broadcasting to perform operations on a Numpy array.

Exercise 2: Is broadcasting used in the `sigmoid` function?

The sigmoid function:

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

is important in machine learning because it maps any real-valued input to a value between 0 and 1. This property makes it especially useful for modeling probabilities and binary outcomes.

In practice, sigmoid is used in binary classification models, such as logistic regression and the output layer of binary neural networks. The sigmoid output represents the probability that an input belongs to the positive class. Sigmoid is mathematically paired with the binary cross-entropy (log loss) function, which provides well-behaved gradients for optimization and enables effective training through gradient descent. For this reason, sigmoid is typically used only in output layers, while other activation functions (e.g., ReLU) are preferred in hidden layers.

In this exercise, you will use NumPy broadcasting to apply the **sigmoid function** to different types of inputs. First of all, write a function which computes the sigmoid of x. Note that x might be a real number or a Numpy array.

```
import numpy as np

def sigmoid(x):
    """
    Computes the sigmoid of x

    Arguments:
    x : float or a numpy.ndarray
        A real number or a Numpy array

    Returns:
    s : float or a numpy.ndarray
        The sigmoid of x
    """

    ### BEGIN SOLUTION

    s = None # Replace the None with the required expression for s

    ### END SOLUTION

    return s
```

Verify your solutions s for this exercise by computing the expected values for the following float and arrays.

```
x = 2.1
expected = 0.8909031


x = np.array([[3.4, -7.1, 9.4],
              [-2.7, 8.882, -2.114]])
expected = np.array([[9.67704535e-01, 8.24424686e-04, 9.99917283e-01],
                     [6.29733561e-02, 9.99861153e-01, 1.07743524e-01]])
```

For the first example, no broadcasting is needed because x is just a number.

For the second example, the things happen in a more iteresting way. First, we see that `x` is an array, so NumPy applies the operator `-` to each element:

```
-x  →  [0, -1, -2]
```

Then, the exponential is applied element by element:

```
np.exp(-x)  →  [⁰e, ⁻e¹, ⁻e²]
```

Now, we have `1` (a single number) and `np.exp(-x)` (an array). Broadcasting is automatically applied when NumPy treats the `1` as if it were copied to match the array:

```
1  →  [1, 1, 1]
```

So the operation becomes:

```
[1, 1, 1] + [⁰e, ⁻e¹, ⁻e²]
```

Although this addition is now done element by element, the `[1, 1, 1]` array is not really created — NumPy does this logically.

Finally, the division occurs when we divide `1` (a single number) and `denominator` (an array). Here, again, NumPy broadcasts the `1`:

```
1  →  [1, 1, 1]
```

In such a way, the division is done element by element. Thus, you can see that **broadcasting** happens whenever a NumPy operation mixes a scalar (like `1`) with an array.

Performing operations along an axis

When working with NumPy arrays, especially two-dimensional arrays (matrices), we often want to apply an operation by rows or by columns instead of to the whole array at once. NumPy allows us to do this using the concept of an axis. In NumPy, **axis 0** refers to the rows direction (down the rows), while **axis 1** refers to the columns direction (across the columns).

NumPy also has other useful functions like np.sum(), np.max() and np.min(). When you just pass the Numpy array to these functions, they simply return the answer of the operation **over the entire array**.

Now suppose you want to find the maximum element of each column of an array. You can do this by passing an additional argument called `axis` to the function. For instance, if A is a two dimensional array, if you want to find the maximum of each column of A, you can use `np.max(A, axis=0)`. Conversely, if A is a two dimensional array, you use `np.max(A, axis=1)` if you want to find the maximum of each row of A.

```
# Minimum of all elements of array

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

```

```
print(a)
print(np.min(a))
```

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

```
# Sum of all columns of an array, returned as one dimensional array
print(np.sum(a, axis=0))
```

```
[6 6 9]
```

A point to note is that these functions may return one dimensional arrays, which might cause errors with broadcasting. In order to make sure that two dimensional arrays are returned, you must pass the argument `keepdims=True` in the function: `np.max(a, axis=0, keepdims=True)`

```
# Sum of all columns of an array, returned as a two dimensional array
print(np.sum(a, axis=0, keepdims=True))
```

```
[[6 6 9]]
```

```
# Sum of all rows of an array
print(np.sum(a, axis=1, keepdims=True))
```

```
[[ 6]
 [15]]
```

Exercise 3: Normalizing all columns of an array

Normalization means rescaling values so that they follow a common rule. A very common type of normalization is to divide each value in a column by a number related to that column (for example, the sum or the maximum of the column).

In many machine learning applications, **data** is normalized before training a model to improve performance and stability. This is because different columns of the data often represent different quantities and can be on very different scales. For example, one column might contain ages (values around tens), while another column might contain income (values in thousands). If we do not normalize the data, columns with larger numerical values can dominate the learning process, even if they are not more important. Normalization rescales each column so that all columns are treated more fairly by the model.

In practice, this means that **each column is handled separately**: we compute statistics (such as the mean and the standard deviation) for one column, and then use them to rescale the values in that same column. In this way, each column can be seen as a **vector of values** that is normalized independently from the others.

As a result, after normalization, all columns have a **similar scale**, which usually helps machine learning algorithms train faster and produce better results.

Suppose that:

- x_i is the i^{th} column of the input array
- c_i is the i^{th} column of the output (normalized) array

The normalization is defined as:

$$c_i = \frac{x_i - \text{mean}(x_i)}{\sigma(x_i)}$$

where:

- $\text{mean}(x_i)$ is the average (mean) of all the elements in column x_i
- $\sigma(x_i)$ is the standard deviation of all the elements in column x_i

You may revise np.mean() and np.std() for convenience.

For this exercise, you will **normalize all the columns** of a two-dimensional NumPy array. You may assume that $\sigma(x)$ is never equal to 0. Try to complete this exercise without using any for loops.

NumPy functions make use of a technique called vectorization which make them much faster than for loops. Using vectorized implementations can often make magnitudes of difference in the training time of your models: where your model might initially take a couple of days to train, it would now be done in a couple of hours.

```
import numpy as np

def normalize(x):
    """
    Normalize all the columns of x

    Arguments:
    x : numpy.ndarray
        A two dimensional Numpy array

    Returns:
    c : numpy.ndarray
        The normalized version of x
    """

    ### BEGIN SOLUTION

    # Step 1.   Calculate the mean of all columns
    mean = None     # Replace the None with the required expression for mean

    # Setep 2.  Calculate the standard deviation all columns
    sigma = None    # Replace the None with the required expression for sigma

    # Step 3.   Compute the final answer
```

```
    c = (x - mean) / sigma

    ### END SOLUTION

    return c
```

Verify your solutions c for this exercise by computing the expected values for the following arrays.

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


x = np.array([[324.33, 136.11, 239.38, 237.17],
              [123.43, 325.24, 243.52, 745.25],
              [856.36, 903.02, 430.25, 531.35]])
expected = np.array([[-0.35694152, -0.97689929, -0.73023324, -1.28391946],
                     [-1.00662188, -0.39712973, -0.68372539,  1.1554411 ],
                     [ 1.3635634 ,  1.37402902,  1.41395863,  0.12847837]])
```

1.8 Linear Algebra

Linear algebra is the branch of mathematics that works with **vectors** and **matrices**, and it is a fundamental tool in science, engineering, and machine learning. In Python, the NumPy library provides simple and efficient ways to create, manipulate, and compute with vectors and matrices. Using NumPy, we can perform operations such as matrix addition, multiplication, transposition, and solving systems of equations with just a few lines of code. Because NumPy uses optimized, vectorized implementations, these operations are not only easy to write but also very fast, making it an ideal tool for learning and applying linear algebra in practice.

In this notebook, we will represent vectors using either **one-dimensional NumPy arrays** or **two-dimensional column arrays** with shape $(d, 1)$. We will represent matrices using **two-dimensional Numpy arrays**.

```
v = np.arange(6)
print('Vector:\n', v)
print("Shape of vector:", v.shape)

M = np.array([[1, 2, 3],
              [4, 5, 6]])
print('\nMatrix:\n', M)
print("Shape of matrix:", M.shape)
```

```
Vector:
 [0 1 2 3 4 5]
Shape of vector: (6,)

Matrix:
 [[1 2 3]
```

```
 [4 5 6]]
Shape of matrix: (2, 3)
```

Transposing a Matrix

Transposing a matrix means **swapping its rows and columns**.

- The first row becomes the first column
- The second row becomes the second column
- And so on…

So, if a matrix has shape (m, n), its transpose will have shape (n, m). In NumPy, you can transpose a matrix using the `.T` attribute (or the `np.transpose()` function). For example, if `M` is a matrix, then you can compute its transpose using `M.T`.

```python
M = np.array([[5, 2, 9],
              [6, 1, 0]])

print('M\n', M)
print('\nTranspose of M (M.T)\n', M.T)


B = np.arange(9).reshape((3, 3))

print('\nB\n', B)
print('\nB.T', B.T)
```

```
M
 [[5 2 9]
 [6 1 0]]

Transpose of M (M.T)
 [[5 6]
 [2 1]
 [9 0]]

B
 [[0 1 2]
 [3 4 5]
 [6 7 8]]

B.T
 [[0 3 6]
 [1 4 7]
 [2 5 8]]
```

Note that in NumPy, taking the transpose of a one-dimensional (1D) array has **NO** effect.

This is because a 1D array does not have rows and columns—it is just a list of numbers with shape `(n,)`. Since there is no distinction between rows and columns, NumPy cannot swap them, so the transpose looks exactly the same.

```
a = np.ones(3)
print('a:', a)
print('Shape of a:', a.shape)
print('\na.T:', a.T)
print('Shape of a.T:', a.T.shape)
```

```
a:
 [1. 1. 1.]
Shape of a: (3,1)

A.T:
 [1. 1. 1.]
Shape of a.T: (3,1)
```

However, if you use a 2D array with shape $(d, 1)$ (a column vector), then transposing does change its shape and structure.

In this case:

- The original array has shape $(d, 1)$ -> a column vector
- Its transpose will have shape $(1, d)$ -> a row vector

```
a = np.ones((3,1))
print('a:\n', a)
print('Shape of a:', a.shape)

print('\na.T:\n', a.T)
print('Shape of a.T:', a.T.shape)
```

```
a:
 [[1.]
 [1.]
 [1.]]
Shape of a: (3, 1)

a.T:
 [[1. 1. 1.]]
Shape of A.T: (1, 3)
```

Dot Product and Matrix Product

In linear algebra, we often need to **combine vectors and matrices using multiplication**. Two of the most common operations are the **dot product** and the **matrix product**.

- The dot product takes two vectors and produces a single number (a **scalar**).
- The matrix product combines matrices (or a matrix and a vector) to produce a **new vector or matrix**.

In NumPy, you can compute these using - `np.dot(A, B)` - Or the `@` operator

You can revise the documentation of np.dot() for more details.

```python
def dot(a, b):
    """Compute dot product between a and b.
    Args:
        a, b: (2,) ndarray as R^2 vectors

    Returns:
        a number which is the dot product between a, b
    """

    dot_product = a[0]*b[0] + a[1]*b[1]

    return dot_product

# Test your code
a = np.array([1,0])
b = np.array([0,1])
print(dot(a,b))  # Should output 0
```

```
0.0
```

This makes geometric sense due to the vectors `[1,0]` and `[0,1]` are perpendicular (orthogonal), so their dot product is 0. That is:

- `a = [1, 0], b = [0, 1]`
- `a[0]` and `a[1]` access the first and second components of vector `a`
- `b[0]` and `b[1]` access the first and second components of vector `b`
- `a[0]*b[0] = 1 * 0 = 0`
- `a[1]*b[1] = 0 * 1 = 0`
- Sum = 0 + 0 = 0

Both `np.dot(A, B)` or the `@` operator do the same thing, but the `@` operator is shorter, cleaner, and closer to mathematical notation. So, from now on, we will use only the `@` operator.

```python
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])

print('x:', x)
print('y:', y)

print('\nnp.dot(x, y):', np.dot(x, y))
print('x @ y:', x @ y)
```

```
x: [1 2 3]
y: [4 5 6]

np.dot(x, y): 32
x @ y: 32
```

You can also use the `@` operator to compute the matrix product between two matrices or between a matrix and a vector (written as a column vector).

In both cases, the dimensions must be compatible: - For `A @ B`, the number of columns of `A` must match the number of rows of `B`. - For `A @ v`, the number of columns of `A` must match the size of the vector `v`.

```
A = np.array([[2, 0, 1],
              [1, 3, 4],
              [0, 2, 1]])

B = np.arange(6).reshape(3, 2)

C = np.array([3, 2, 8]).reshape((-1, 1))   # column vector

print('A:\n', A)
print('B:\n', B)
print('A @ B:\n', A @ B)

print('\nA:\n', A)
print('C:\n', C)
print('A @ C:\n', A @ C)
```

```
A:
 [[2 0 1]
 [1 3 4]
 [0 2 1]]
B:
 [[0 1]
 [2 3]
 [4 5]]
A @ B:
 [[ 4  7]
 [22 30]
 [ 8 11]]

A:
 [[2 0 1]
 [1 3 4]
 [0 2 1]]
C:
 [[3]
 [2]
 [8]]
A @ C:
 [[14]
 [41]
 [12]]
```

In the next example, we explore how different inner products change the way we measure angles between vectors. The standard Euclidean dot product gives us our usual geometric intuition, but many applications require modified inner products that weigh dimensions differently.

The code below demonstrates this concept by defining an inner product through a symmetric matrix **A**, which acts as a weighting matrix.

```
A = np.array([[1, -1/2],[-1/2,5]])      # the matrix A defines the inner product
x = np.array([0,-1])
y = np.array([1,1])

def find_angle(A, x, y):
    """Compute the angle"""
    inner_prod = x.T @ A @ y
    norm_x = np.sqrt(x.T @ A @ x)       # length of x (norm_x) in this inner product
    norm_y = np.sqrt(y.T @ A @ y)       # length of y (norm_y) in this inner product
    alpha = inner_prod/(norm_x*norm_y)
    angle = np.arccos(alpha)
    return np.round(angle,2)

print(find_angle(A, x, y))
```

```
2.69
```

When using the `@` operator for matrix multiplication, the shapes of the matrices must be compatible

Remember the rule: - If `A` has shape `(m, n)` - And `B` has shape `(n, p)`

Then `A @ B` is valid and will have shape `(m, p)`.

If the number of columns of `A` does not match the number of rows of `B`, NumPy cannot perform the multiplication and will raise an error.

```
A = np.array([[1, 2, 3],
              [4, 5, 6]])   # Shape: (2, 3)

B = np.array([[1, 2],
              [3, 4]])      # Shape: (2, 2)

print("A shape:", A.shape)
print("B shape:", B.shape)

# This will cause an error because the shapes are not compatible
A @ B
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, with
gufunc signature (n?,k),(k,m?)->(n?,m?) (size 3 is different from 2)
```

Exercise (Optional): Computing Angles with a Non-Standard Inner Product

For vectors **x** and **y** in $\mathbb{R}^2$, the inner product is defined by a symmetric positive definite matrix A is:

$$\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^T A \mathbf{y}$$

The angle θ between vectors is then computed using the familiar cosine formula adapted to this new inner product:

$$\cos\theta = \frac{\langle \mathbf{x}, \mathbf{y} \rangle}{\sqrt{\langle \mathbf{x}, \mathbf{x} \rangle} \cdot \sqrt{\langle \mathbf{y}, \mathbf{y} \rangle}}$$

In this exercise, compute the angle between

$$\mathbf{x} = \begin{bmatrix} 1 \\ \\ 1 \end{bmatrix}$$

and

$$\mathbf{y} = \begin{bmatrix} 1 \\ \\ -1 \end{bmatrix}$$

using the inner product defined by

$$\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^T \begin{bmatrix} 1 & 0 \\ \\ 0 & 5 \end{bmatrix} \mathbf{y}$$

```python
import numpy as np

# Step 1. Fill in the arrays
A = np.array( )
x = np.array( )
y = np.array( )

# Step 2. Define the function `find_angle`
def find_angle(A, x, y):
    pass
    return np.round(angle, 4)

print(find_angle(A, x, y))
```

To verify your answer, use these arrays

```python
A = np.array([[1, 0], [0, 5]])
x = np.array([1, 1])
y = np.array([1, -1])

```

```
2.3 rad
```

Lastly, remember that in NumPy, the operators * and @ do not mean the same thing:

- * performs element-wise multiplication Each element is multiplied by the element in the same position.
- @ performs matrix multiplication (linear algebra product) Rows of the first matrix are combined with columns of the second matrix.

This is a very common source of confusion for beginners, so it's important to keep the difference in mind.

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

B = np.array([[5, 6],
              [7, 8]])

print("A:\n", A)
print("B:\n", B)

print("\nA * B (element-wise multiplication):\n", A * B)
print("\nA @ B (matrix multiplication):\n", A @ B)
```

```
A:
 [[1 2]
 [3 4]]
B:
 [[5 6]
 [7 8]]

A * B (element-wise multiplication):
 [[ 5 12]
 [21 32]]

A @ B (matrix multiplication):
 [[19 22]
 [43 50]]
```

Outer product

The **outer product** of two vectors produces a matrix. The outer product of two vectors can be calculated using the function `np.outer()`. The function `np.outer()` creates a matrix where each row is a scaled version of `b`.

There is a key difference with **dot product**:

- $ab^\top$ = Outer product → Creates a **matrix** from two vectors
- $a^\top b$ = Dot product → Computes a **scalar** from two vectors

```
a = np.array([1, 2, 3])      # length 3
b = np.array([4, 5])         # length 2
```

```

outer = np.outer(a, b)
print(outer)
```

```
[[ 4  5]
 [ 8 10]
 [12 15]]
```

```
u = np.array([1, 2, 2])
outer_uu = np.outer(u, u)
print(outer_uu)
```

```
[[1 2 2]
 [2 4 4]
 [2 4 4]]
```

This operation is important in projection matrices, where we need $bb^\top$ to create $D \times D$ matrix:

```
b = np.array([1, 2, 2])
bbT = np.outer(b, b)    # 3×3 matrix
bTb = np.dot(b, b)      # scalar 9

P = bbT / bTb           # projection matrix

print("bb^T (outer product):")
print(bbT)
print("\nb^Tb (dot product):")
print(bTb)
print("\nProjection matrix P = bb^T / b^Tb:")
print(P)
```

```
bb^T (outer product):
[[1 2 2]
 [2 4 4]
 [2 4 4]]

b^Tb (dot product):
9

Projection matrix P = bb^T / b^Tb:
[[0.11111111 0.22222222 0.22222222]
 [0.22222222 0.44444444 0.44444444]
 [0.22222222 0.44444444 0.44444444]]
```

The `numpy.linalg` library

NumPy provides a special module called `numpy.linalg` that contains many useful tools for linear algebra.

With `numpy.linalg`, you can easily:

- Solve systems of linear equations
- Compute determinants and inverses of matrices
- Compute eigenvalues and eigenvectors
- Compute matrix norms
- And perform many other common linear algebra operations

Instead of implementing these operations yourself, you can rely on this library, which is fast, well-tested, and easy to use. You can revise the documentation of np.linalg for more details.

For example, we can compute the **determinant** of a square matrix by using np.linalg.det().

```
A = np.array([[2, 0, 1],
              [1, 3, 4],
              [0, 2, 1]])

detA = np.linalg.det(A)     # This computes the determinant
print("det(A) =", detA)
```

```
det(A) = 8.0
```

> **Note:** Remember that small numerical differences (like `8.000000000000002`) can sometimes appear because of floating-point arithmetic

We can compute the inverse of a matrix by using np.linalg.inv().

The inverse of a matrix `A` is another matrix A^{-1} such that $A@A^{-1} = I$, where `I` is the identity matrix.

```
A = np.array([[2, 0, 1],
              [1, 3, 4],
              [0, 2, 1]])

A_inv = np.linalg.inv(A)        # Compute the inverse of A

print("A:\n", A)
print("\nInverse of A:\n", A_inv)


I = np.eye(3)       # Generate the identity matrix
print("\nIdentity matrix I:\n", I)

# Check that A @ A_inv is (approximately) the identity matrix
print("\nA @ A_inv:\n", A @ A_inv)
```

```
A:
 [[2 0 1]
 [1 3 4]
 [0 2 1]]

Inverse of A:
 [[-0.625  0.25   0.875]
 [ 0.125  0.25  -0.375]
```

```
 [ 0.25  -0.5    0.75 ]]

Identity matrix I:
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]

A @ A_inv:
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

> **Note:** Not every matrix has an inverse (the determinant must be non-zero). Because of floating-point arithmetic, results are usually approximately the identity matrix, not perfectly exact. In this last example, we use `np.eye(3)` to create a `3×3` identity matrix.

We can compute the eigenvalues and eigenvectors of a matrix using np.linalg.eig().

This function returns two objects:

- An array with the eigenvalues
- A matrix whose columns are the corresponding eigenvectors

Let's use a simple symmetric matrix so the result is easy to interpret:

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

# Compute the eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(A)

print("A:\n", A)
print("\nEigenvalues:\n", eigenvalues)
print("\nEigenvectors (columns):\n", eigenvectors)
```

```
A:
 [[2 1]
 [1 2]]

Eigenvalues:
 [3. 1.]

Eigenvectors (columns):
 [[ 0.70710678 -0.70710678]
 [ 0.70710678  0.70710678]]
```

So far, we have seen how to compute eigenvalues and eigenvectors using `np.linalg.eig()`. But what do these results actually mean?

By definition, if v is an eigenvector of a matrix A with eigenvalue λ, then multiplying the matrix by this vector does not change its direction—it only scales it:

$$A@v = \lambda v$$

In the following example, we will take one of the eigenvalues and eigenvectors returned by NumPy and verify this property step by step using simple matrix and vector operations.

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

# Compute eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(A)

print("A:\n", A)
print("\nEigenvalues:\n", eigenvalues)
print("\nEigenvectors (columns):\n", eigenvectors)

# Take the first eigenvalue and eigenvector
lambda_0 = eigenvalues[0]
v_0 = eigenvectors[:, 0]

print("\nFirst eigenvalue (lambda):", lambda_0)
print("First eigenvector (v):\n", v_0)

# Compute both sides of A @ v = lambda * v
left = A @ v_0
right = lambda_0 * v_0

print("\nA @ v:\n", left)
print("\nlambda * v:\n", right)
```

```
A:
 [[2 1]
 [1 2]]

Eigenvalues:
 [3. 1.]

Eigenvectors (columns):
 [[ 0.70710678 -0.70710678]
 [ 0.70710678  0.70710678]]

First eigenvalue (lambda): 3.0
 First eigenvector (v):
 [0.70710678 0.70710678]

A @ v:
 [2.12132034 2.12132034]

lambda * v:
 [2.12132034 2.12132034]
```

> **Note:** An **eigenvector** keeps its direction when multiplied by the matrix, while the **eigenvalue** tells you how much it is stretched or shrunk

Sorting eigenvalues and reordering eigenvectors

Suppose that we want the eigenvalues in the diagonal matrix `D` to appear in **non-decreasing order**. However, `np.linalg.eig(A)` does **not** guarantee any particular order for the eigenvalues. Therefore, we need to **sort them manually**.

```
eig_vals = np.array([6, -1])
```

The function `np.argsort(eig_vals)` does **not** sort the values directly. Instead, it returns the **indices** that would sort the array.

```
eig_vals = np.array([6, -1])
idx = np.argsort(eig_vals)
print(idx)
```

```
[1 0]
```

This means:

- The smallest value is at index `1` (which is `-1`)
- The next value is at index `0` (which is `6`)

So `idx` tells us the order of indices that sorts `eig_vals`.

NumPy also allows indexing an array using another array of indices. This is often called **fancy indexing**.

```
eig_vals = np.array([6, -1])
idx = np.array([1, 0])

eig_vals_sorted = eig_vals[idx]
print(eig_vals_sorted)
```

```
[-1  6]
```

So the line `eig_vals = eig_vals[idx]` means "reorder `eig_vals` using the index order stored in `idx`.

Recall that `S` is a matrix whose **columns are eigenvectors**, and each column corresponds to one eigenvalue in `eig_vals`.

If we reorder the eigenvalues, we must also reorder the columns of `S` in the same way, so that each eigenvector still matches its eigenvalue.

The syntax:

```
S[:, idx]
```

means:

- : → take all rows
- idx → take only the columns in the order given by idx

So the line `S = S[:, idx]` means "reorder the columns of `S` using the same index order used to sort the eigenvalues."

We can write down all the lines together,

```
idx = np.argsort(eig_vals)
eig_vals = eig_vals[idx]
S = S[:, idx]
```

that indicates:

- Compute the index order that sorts the eigenvalues
- Reorder the eigenvalues using that order
- Reorder the columns of `S` in the same way, so each eigenvector still corresponds to its eigenvalue

  Note: Remember that the function `np.linalg.eig()` does not guarantee sorted eigenvalues.

When we say that eigenvalues should be in **non-decreasing order**, we mean they should be ordered from **smallest to largest** (allowing equal values if any). For example, suppose a matrix has the following eigenvalues (as returned by NumPy, in arbitrary order):

```
eig_vals = np.array([6, -1, 3, 3])
print(eig_vals)
```

```
[ 6 -1  3  3 ]
```

These eigenvalues are not sorted. To **sort** them in **non-decreasing order**, we can do:

```
idx = np.argsort(eig_vals)
eig_vals_sorted = eig_vals[idx]

print("Sorted eigenvalues:", eig_vals_sorted)
```

```
Sorted eigenvalues: [-1  3  3  6]
```

Exercise 4: Diagonalizing a Matrix

Diagonalization is an important concept in linear algebra that builds directly on **eigenvalues** and **eigenvectors**.

A square matrix `A` is said to be **diagonalizable** if it can be written in the form $A = SDS^{-1}$,

where:

- `S` is a matrix whose columns are the eigenvectors of `A`,
- `D` is a diagonal matrix containing the corresponding eigenvalues of `A`,
- `S^{-1}` is the inverse of `S`.

You might find np.zeros(), np.linalg.eig() and np.linalg.inv() useful. Note that for this exercise, you may assume that A is always diagonalizable.

For testing purposes:

- Each eigenvector in S must be of **unit length**. This is automatically satisfied if you use np.linalg.eig(). If you do not, you may need to normalize the eigenvectors yourself.
- The eigenvalues in `D` must appear in **non-decreasing order**.

The idea behind diagonalization is to rewrite a matrix in a simpler form (diagonal), which makes many computations easier, such as computing powers of a matrix or understanding how the matrix acts on vectors.

In this exercise, you need to find matrices S and D. Recall that in order to do this, you must first find all the eigenvalues and eigenvectors of A. Then, S is the matrix of all the eigenvectors arranged as columns, and D is the matrix of the corresponding eigenvalues arranged along the diagonal.

> **Note:** Eigenvectors are not unique. If `v` is an eigenvector, then any non-zero multiple of `v`is also a valid eigenvector. For this reason, there are many possible correct matrices `S`. Different implementations (or NumPy itself) may return eigenvectors that differ by a scaling factor or by a sign. As long as the columns of `S` are valid eigenvectors corresponding to the eigenvalues in `D`, the diagonalization is correct. In this exercise, we will use normalized (unit-length) eigenvectors, as returned by `np.linalg.eig()`.

Suppose

$$A = \begin{bmatrix} 1 & 5 \\ 2 & 4 \end{bmatrix}$$

Then, one possible choise is

$$S = \begin{bmatrix} -2.5 & 1 \\ 1 & 1 \end{bmatrix}$$

and

$$D = \begin{bmatrix} -1 & 0 \\ 0 & 6 \end{bmatrix}$$

Note that this is just one valid diagonalization. NumPy will typically return normalized eigenvectors, so the matrix `S` you obtain in practice may look different, even though it represents the same eigen-directions and still satisfies $A = SDS^{-1}$.

```python
import numpy as np

def diagonalize(A):
    """
    Diagonalizes the input matrix A.

```

```
    Parameters:
    A : np.ndarray
        A two-dimensional NumPy array which is guaranteed to be diagonalizable.

    Returns:
    S : np.ndarray
        Matrix whose columns are the eigenvectors of A.
    D : np.ndarray
        Diagonal matrix of eigenvalues.
    S_inv : np.ndarray
        Inverse of S.
    """

    ### BEGIN SOLUTION

    # Step 1. Retrieve the number of rows in A
    n = 0

    # Step 2. Get the eigenvalues and eigenvectors of A
    eig_vals, S = None, None

    # Step 3. Start by initializing D to a matrix of zeros of the appropriate shape
    D = None

    # Step 4. Set the diagonal elements of D to be the eigenvalues
    for i in range(n):
        pass

    # Step 5. Compute the inverse of S
    S_inv = None

    ### END SOLUTION

    return S, D, S_inv
```

Verify your solutions S, D, S_inv for this exercise by computing the expected values for the following `(2x2)` and `(4x4)`arrays.

```
A = np.array([[1, 5],
              [2, 4]])

S, D, S_inv = diagonalize(A)

print("A:\n", A)
print("\nS:\n", S)
print("\nD:\n", D)
print("\nS_inv:\n", S_inv)

print("\nReconstructed A (S @ D @ S_inv):\n", S @ D @ S_inv)
print("\nAll close?", np.allclose(A, S @ D @ S_inv))
```

```python
A = np.array([[4, -9,  6, 12],
              [0, -1,  4,  6],
              [2, -11, 8, 16],
              [-1, 3,  0, -1]])

S, D, S_inv = diagonalize(A)

print("A:\n", A)
print("\nS:\n", S)
print("\nD:\n", D)
print("\nS_inv:\n", S_inv)

# Reconstruct A using the diagonalization
A_reconstructed = S @ D @ S_inv

print("\nReconstructed A (S @ D @ S_inv):\n", A_reconstructed)

# Check if the reconstruction is correct (up to numerical precision)
print("\nIs the reconstruction correct? ", np.allclose(A, A_reconstructed))
```

Exercise 5: Polynomial Multiplication

A **polynomial** is an expression made of variables and coefficients, such as:

$$p(x) = 2x^2 + 3x + 1$$

and

$$q(x) = x + 4$$

Multiplying two polynomials means **distributing** every term of the first polynomial over every term of the second polynomial, and then **combining like terms**.

For example:

$$(2x^2 + 3x + 1)(x + 4)$$

$$= 2x^3 + 8x^2 + 3x^2 + 12x + x + 4$$

$$= 2x^3 + 11x^2 + 13x + 4$$

So the product is another polynomial whose degree is the **sum of the degrees** of the original polynomials.

But, why is this related to arrays and linear algebra? If we store the coefficients of a polynomial in an array, polynomial multiplication becomes a **systematic combination of coefficients**, very similar to:

- Convolutions
- Dot products
- Matrix-vector products

For example, the polynomial:

$$p(x) = 2x^2 + 3x + 1$$

can be represented by the coefficient array:

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

And,

$$q(x) = x + 4$$

can be represented as:

```
[1, 4]
```

Multiplying these two arrays in the right way produces the coefficients of the product polynomial.

In this exercise, you will implement a function to multiply two polynomials using NumPy. This task brings together several concepts you have learned so far, such as **array slicing**, **dot products**, and **vectorization**.

You are given two **one-dimensional NumPy arrays** `A` and `B` containing the coefficients of two polynomials. We will use the convention that:

- `A[i] = a_i` is the coefficient of (x^i) in the first polynomial
- `B[i] = b_i` is the coefficient of (x^i) in the second polynomial

Your goal is to compute the coefficients of the product polynomial $C = A \cdot B$.

More formally, if `C` is the resulting one-dimensional array and `C[i] = c_i`, then:

$$c_i = \sum_{j+k=i} a_j \, b_k$$

There are multiple ways to implement polynomial multiplication. If your approach requires a NumPy function that we have not introduced yet, we encourage you to consult the NumPy documentation. However, try to implement the function using only **one `for` loop over** (i), and compute the summation using **NumPy operations** (instead of nested loops). This will make your code faster thanks to **vectorization**.

Hints: - `A` and `B` may have different lengths. Pad the **end** of the shorter array with zeros so they have the same length. You may find `np.pad()` useful. - For a fixed (i), notice how the valid indices (j) and (k) must satisfy (j+k=i). This often leads to multiplying two **slices** of the arrays. The summation may resemble a **dot**

product. - You can reverse an array using `np.flip()`, which can be helpful for aligning terms. - Make sure your output does **not** end with unnecessary trailing zeros. Look for a NumPy-based way to remove them.

```
import numpy as np

def multiply(A, B):
    """
    Multiplies two polynomials represented by their coefficient arrays.

    Parameters
    ----------
    A : np.ndarray
        Coefficients of the first polynomial.
    B : np.ndarray
        Coefficients of the second polynomial.

    Returns
    -------
    C : np.ndarray
        Coefficients of the product polynomial A * B.
    """

    ### BEGIN SOLUTION

    # Step 1. Find the number of coefficients of both polynomials
    na = None
    nb = None

    # Step 2. Pad the smaller array with zeros so A and B have the same length
    if False:
        pass
    else:
        pass

    # Step 3. Initialize the output array with zeros
    C = None

    # Step 4. Perform the multiplication
    # You might want to break the loop over i into two separate phases
    pass

    # Step 5. Remove any extra zeros from the end of C
    pass

    ### END SOLUTION

    return C
```

Verify your solutions `C` for this exercise by computing the expected values for the following arrays.

```
# Test case 1
A = np.array([1, 2])
```

```
B = np.array([3, 4])
C_exp = np.array([3, 10, 8])

C = multiply(A, B)
print("Test 1 result:   ", C)
print("Test 1 expected: ", C_exp)
print("Test 1 correct?  ", np.allclose(C, C_exp))

print()


#Test case 2
A = np.array([5, 6])
B = np.array([1, 3, 5, 9])
C_exp = np.array([5, 21, 43, 75, 54])

C = multiply(A, B)
print("Test 2 result:   ", C)
print("Test 2 expected: ", C_exp)
print("Test 2 correct?  ", np.allclose(C, C_exp))
```

Exercise 6: Support Vector Classifiers

In machine learning methods, support vector machines (SVM) are algorithms or models used in applications of regression and classificaction. In this framework, the **support vectors** are the critical data points that shape the model. They are literally data **vectors** (points in feature space) that happen to define the SVM model, and constitute a **subset of your training rows** that the model actually uses to build the decision boundary. For example, if you draw a line to separate two groups, the points closest to that line are the support vectors. If you move them, the line moves. If you remove other points far away, the line usually stays the same.

After training an SVM, the model predicts using a formula of this form:

$$f(x) = \sum_{i \in SV} \alpha_i \, y_i \, K(x_i, x) + b$$

where:

- $f(x)$ is a real-valued function: $\mathbb{R}^n \rightarrow \mathbb{R}$.
- $x = the new point (sample) you want to predict
- $x_i = (x_1, x_2, \ldots, x_n)$ = training points, with n the number of features per ith training sample
- y_i = their labels for classification (e.g., +1, -1)
- $K(\cdot, \cdot)$ = kernel function (or dot product in linear SVM)
- α_i = learned weights
- b = bias
- SV = only the support vectors

The function $f(x)$ takes an input vector $x \in \mathbb{R}^n$ and outputs a real number $f(x) \in \mathbb{R}$.

In binary classification:

$$\begin{cases} f(x) > 0 & \Rightarrow \text{class } +1 \\ f(x) < 0 & \Rightarrow \text{class } -1 \\ f(x) = 0 & \Rightarrow \text{ on the decision boundary} \end{cases}$$

The sign of $f(x)$ gives the class, while the magnitude | $f(x)$ | relates to how far you are from the boundary (in model space).

In the case of SVM used for classification (i.e. SVC), only support vectors appear in such a the prediction formula. Because of how SVM is trained (optimization with constraints), Each training point gets a coefficient α_i, then most $\alpha_i = 0$ and those points different to zero are the suport vectors. Mathematically, only support vectors survive in the final formula.

In this exercise, you will work with the **linear case**. You build a toy SVC decision function and evaluate it on a point. You will use 3D vectors, a linear kernel (i.e. a dot product), and a data set

$$X = \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$$

where:

- Each $x_i \in \mathbb{R}^n$ is a vector (one sample). Here, x_1, x_2, …, x_N are different elements (for the set X) with n number of features
- Each $y_i \in \{+1, -1\}$ is its label

From these, SVM learns the coefficients α_i, selects some of the x_i as support vectors and builds the decision function.

Assume after training, SVC found three support vectors in $\in \mathbb{R}^3$:

$$x_1 = (1, 0, 0), \quad y_1 = +1, \quad \alpha_1 = 1$$

$$x_2 = (0, 1, 0), \quad y_2 = -1, \quad \alpha_2 = 1$$

$$x_3 = (0, 0, 1), \quad y_3 = +1, \quad \alpha_2 = 0.5$$

Also, assume $b = 0$ and the kernel linear $K(x_i, x) = x_i \cdot x$

The decision formula is:

$$f(x) = 1(+1)(x_1 \cdot x) + 1(-1)(x_2 \cdot x) + 0.5(+1)(x_3 \cdot x)$$

Suposse a point for classification to be

$$x = (2, 1, 4)$$

So, after the dot produc operations, the decision formula becomes

$$f(x) = 2 - 1 + 0.5(4) = 3$$

This implies that $f(x) > 0$, thus the point is predicted to be classified as class $+1$. In practice, you will have more a set of points (real sample) to be classified.

Write a **NumPy-only** script that reproduces the toy example: build the linear kernel matrix, identify support vectors from alpham, compute the decision formula $f(x)$, and the predicted class from the sign of $f(x)$.

```python
import numpy as np

def svm_linear_predict(X, y, alpha, b, x_new):
    """
    Predicts the class for a new sample using a linear kernel.

    Parameters
    ----------
    X : np.ndarray, shape (N, D)
        Training data with N samples and D features.
    y : np.ndarray, shape (N,)
        Training labels (+1 or -1 for each sample).
    alpha : np.ndarray, shape (N,)
        Learned alpha parameters from SVM training.
    b : float
        Learned bias term.
    x_new : np.ndarray, shape (D,)
        New sample to classify.

    Returns
    -------
    f : float
        Decision function value f(x).
    y_pred : int
        Predicted class (+1 or -1).
    """

    ### BEGIN SOLUTION

    # Step 1: Find support vectors (indices where alpha != 0)
    sv_idx = None

    # Step 2: Compute kernel values between each training sample and x_new
    # For linear kernel: k_i = X[i] dot x_new
    k = None

    # Step 3: Compute decision function f(x)
    # f(x) = sum_{i in SV} alpha_i * y_i * K(x_i, x) + b
```

```
    f = None

    # Step 4: Predict class based on sign of f(x)
    y_pred = None

    ### END SOLUTION

    return f, y_pred


def svm_linear_kernel_matrix(X):
    """
    Computes the linear kernel matrix for training data.

    Parameters
    ----------
    X : np.ndarray, shape (N, D)
        Training data with N samples and D features.

    Returns
    -------
    K : np.ndarray, shape (N, N)
        Kernel matrix where K[i, j] = X[i] dot X[j].
    """

    ### BEGIN SOLUTION

    # Compute the kernel matrix using linear kernel (dot product)
    # Hint: Use matrix multiplication
    K = None

    ### END SOLUTION

    return K
```

Verify your solutions for this exercise by computing the expected values for the following arrays.

```
# Test case 1
X = np.array([
    [1, 0, 2],
    [0, 1, 1],
    [2, 1, 0],
    [1, 2, 1],
], dtype=float)

y = np.array([+1, +1, -1, -1], dtype=float)
alpha = np.array([0.5, 0.0, 0.5, 0.0], dtype=float)
b = -0.2
x_new = np.array([1, 1, 1], dtype=float)


# Compute kernel matrix
```

```python
K = svm_linear_kernel_matrix(X)
print("Kernel matrix K (linear):")
print(K)

# Make prediction
f, y_pred = svm_linear_predict(X, y, alpha, b, x_new)
print(f"\nDecision value f(x): {f}")
print(f"Predicted class: {y_pred}")
```

1.9 Saving and Loading Data with `.npy` and `.npz` Files

NumPy's `.npy` files are the **native binary format** for storing a single NumPy arrays. They are designed to save and load arrays efficiently and without losing any information.

You can think of an `.npy` file as an exact snapshot of an array. It stores not only the data, but also the shape, the data type (`dtype`), and the memory layout. This is especially useful when working with large datasets or expensive computations that you do not want to repeat every time you run your program.

However, you still might wonder why not use text formats like `.txt` or `.csv`. This is because the `.npy` format has three important advantages:

- Speed: Saving and loading are much faster because the data is stored in binary form, without needing to parse text.
- Size. Files are usually smaller thanks to efficient binary storage.
- Fidelity. All NumPy-specific information (such as `dtype='complex128'` or `dtype='<U10'` for strings) is preserved exactly.

NumPy provides two main functions for working with this format: `np.save()` and `np.load()`. To **save** an array, you provide a filename and the array. NumPy automatically adds the `.npy` extension. To **load** the array, NumPY reads the file and returns the array exactly as it was saved, ready to use.

```python
data_array = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.float64)

np.save('my_data.npy', data_array)      # save the array to disk

loaded_array = np.load('my_data.npy')   # load the array back

print(loaded_array)
print(f"\nType: {loaded_array.dtype}")
print(f"\nShape: {loaded_array.shape}")
```

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

Type: float64

Shape: (2, 3)
```

Let's consider a simple example. Suppose you have a raw **CSV** file `raw_grades.csv` that contains student grades, but with some inconsistencies:

```
student_id,math,physics,chemistry
101,85.5,92.0,78.5
102,90.0,88.5,NA
103,76.0,82.5,91.0
104,NA,95.0,87.5
105,88.0,91.5,84.0
```

Notice that some values are missing and are represented by the string `"NA"`. Before using this data for analysis (or machine learning), these values must be **cleaned and handled properly**.

A typical workflow is:

- Load the raw data from the CSV file
- Convert the data to a NumPy array, treating `"NA"` as missing values
- Clean or replace the missing values (for example, with `np.nan` or with column averages)
- Save the cleaned array to a binary NumPy file (`.npy`) so it can be quickly reloaded later without repeating the cleaning step

For example, you can load the CSV and handle missing values like this:

```python
# Load the data, telling NumPy that "NA" represents missing values
raw_data = np.genfromtxt(
    "raw_grades.csv",
    delimiter=",",
    skip_header=1,
    dtype=float,
    missing_values="NA",
    filling_values=np.nan
)

print("Raw data loaded into NumPy:")
print(raw_data)
print(f"\nShape: {raw_data.shape}")
print(f"\nHas NaN values: {np.any(np.isnan(raw_data))}")
```

```
Raw data loaded into NumPy:
[[101.  85.5 92.  78.5]
 [102.  90.  88.5  nan]
 [103.  76.  82.5 91. ]
 [104.  nan 95.  87.5]
 [105.  88.  91.5 84. ]]

Shape: (5, 4)

Has NaN values: True
```

At this point, the missing values are represented as `np.nan`, which NumPy understands and can handle in computations.

After cleaning or processing the data (for example, replacing `np.nan` with column means), you can save the result:

```
np.save("clean_grades.npy", raw_data)
```

Later, you can load the cleaned data instantly:

```
clean_data = np.load("clean_grades.npy")
print("Clean data loaded from .npy file:")
print(clean_data)
```

```
Clean data loaded from .npy file:
[[101.   85.5  92.   78.5]
 [102.   90.   88.5   nan]
 [103.   76.   82.5  91. ]
 [104.    nan  95.   87.5]
 [105.   88.   91.5  84. ]]
```

> **Note:** It is a widely used convention and best practice in data analysis and machine learning that **rows** represent individual samples or subjects (e.g., students), while **columns** represent features or properties (e.g., math, physics, chemistry). This layout is used by NumPy, pandas, scikit-learn, and most ML libraries because it makes operations like statistics, filtering, and modeling consistent and convenient.

While `.npy` files are used to store a single NumPy array, the **`.npz`** format is used to store **multiple arrays in one file**.

You can think of an `.npz` file as a container that holds several `.npy` arrays together. Each array is stored with a name, so you can load them later and access each one separately. This is useful when you want to save related data together, such as:

- input data and labels
- multiple intermediate results
- or several arrays that belong to the same experiment

To save multiple arrays, you use `np.savez()`. To load them, you use `np.load()`, which returns an object that works like a dictionary, where each array can be accessed by its name.

PART 2. Introduction to Assert Statements and Testing

When you write code, it is not enough for it to **run without errors**—you also want to be sure that it produces the **correct results**. This is where **testing** becomes important.

In Python, one simple and very useful way to test your code is by using **assert statements**. An **`assert`** statement checks whether a condition is `True`. If it is, the program continues normally. If it is not, Python raises an error and tells you that something went wrong.

In other words, **`assert`** is a way of saying:

> "I expect this to be true. If it is not, stop the program and tell me."

This is extremely helpful when:

- You want to **verify** that your functions return the correct results
- You want to **catch mistakes early**
- You want to **debug** your code more easily

```
def add(a, b):
    return a + b

# This should be true, so nothing happens
assert add(2, 3) == 5

# This is false, so Python will raise an error
assert add(2, 2) == 5
```

As showed, the assert statement lets you check if a condition in your program evaluates to true. If the condition does evaluate to true, then nothing happens and the program execution continues normally. However, if it evaluates to false, then the program execution is immediately terminated, an error is raised and an error message is printed.

Exercise 7: Using `assert` for Debugging

```
# An assert statement where condition evaluates to true
x = 5
assert x == 5
```

Since the condition evaluates to true, nothing happens.

```
# An assert statement where the condition evaluates to false
x = 5
assert x == 4
```

```
---------------------------------------------------------------------------

AssertionError                            Traceback (most recent call last)

      1 # An assert statement where the condition evaluates to false
      2 x = 5
----> 3 assert x == 4


AssertionError:
```

This time, since the assert statement evaluates to false, an error is thrown and the line where the assert statement failed is printed to the standard output. You can also print a message which further explains what error took place.

```
# An asseert statement with an error message
x = 5
```

```
assert x == 4, "x does not store the intended value"
```

```
---------------------------------------------------------------------------

AssertionError                            Traceback (most recent call last)

      1 # An asseert statement with an error message
      2 x = 5
----> 3 assert x == 4, "x does not store the intended value"


AssertionError: x does not store the intended value
```

In this exercise, you will use **assert statements** inside the functions you write to help you **debug** your code.

The `assert` statements are a very powerful tool for finding bugs. They allow you to **check that something you expect to be true is actually true**. If it is not, Python stops the program and shows an error, telling you exactly where the problem occurred.

Let's look at a simple example where an `assert` statement helps us **spot a bug**. Your goal is to **find the bug in the code and fix it**.

```
import numpy as np

def test_assert():
    """
    This function demonstrates the use of assert statements in debugging
    """

    A = np.arange(5)
    s = 0

    # Step 1, add all elements of A to s
    for i in range(A.shape[0]):
        s += A[i]

    # Step 2, subtract all the elements of A in reverse order
    for i in range(A.shape[0] - 1, -1, -1):   # Unfortunately, there is a bug in this loop
        s -= A[i]

    # If everything were correct, s should be 0 at this point
    # This assert checks that assumption
    assert s == 0

test_assert()
```

When you run this code, the assert statement will fail and raise an **AssertionError**, telling you that something went wrong. This means there is a **bug in the code** above.

Your task is to inspect the loops, find the mistake, and fix it so that the assertion passes.

Exercise 8: The Numpy Testing Module

NumPy provides a very useful testing utility called np.testing.assert_allclose() which allows us to test our functions.

We strongly encourage you to read the documentation of this function, as it is widely used in scientific and numerical computing.

Why not use == with floating-point numbers?

- When working with **floating-point numbers**, you often get **small rounding errors** due to the way numbers are represented in the computer. Because of this, two values that should be equal mathematically may differ by a very small amount
- For example, instead of getting exactly `1.0`, you might get something like `0.9999999998` or `1.0000000001`. In such cases, a direct comparison using == may incorrectly report that the numbers are different
- `np.testing.assert_allclose()` solves this problem by checking whether two values (or arrays) are **close enough**, rather than exactly equal

How does `np.testing.assert_allclose()` work? The function takes **two required arguments**:

- The value (or array) you want to test
- The expected value (or array) you want to compare against

It then checks whether they are **equal within a small tolerance**. If they are, nothing happens. If they are not, Python raises an error and tells you that the test failed.

You can also **customize the tolerance** if needed, depending on how strict you want the comparison to be.

This makes `np.testing.assert_allclose()` especially useful for:

- Testing functions that return **floating-point arrays**
- Verifying **numerical algorithms**
- Writing **reliable tests** for scientific code

In the next exercise, you are given a function that **attempts** to compute the inverse of a 2×2 matrix. However, this function is **incorrect**.

```python
def inverse(A):
    """
    Computes (incorrectly) the inverse of A

    A must have shape (2, 2)
    """

    return np.array([[A[1, 1], -A[0, 1]],
                     [-A[1, 0], A[0, 0]]])
```

A test has already been written for this function, but unfortunately, that **test passes even though the function is wrong**. This means the test is not strong enough.

Your tasks are:

- Write a new test for this function that fails, showing that the current implementation is incorrect
- Fix the function so that it becomes correct
- Verify that your corrected function now passes the test you wrote

Here is one test case you can start with:

```
A = np.array([[3, 5],
              [1, 2]])
A_exp = np.array([[2, -5],
                  [-1, 3]])
```

Try to think of **another test case** where the current implementation should fail, and add it below:

```
A = np.array([

    # Add your own test matrix here

])
```

PART 3. Debugging Your Code

As you work through the rest of the sections in this notebook, you will inevitably encounter situations where your code does **not** work as expected. This is completely normal—and it happens to everyone, even experienced programmers.

Debugging can sometimes feel difficult or frustrating, especially when you do not immediately understand what went wrong. In this final part of the notebook, we will introduce some **practical guidelines and strategies** to help you find and fix bugs in your code more effectively.

The goal is not only to solve the current exercises, but also to help you become more **confident and systematic** when debugging your own programs in the future.

3.1 Errors in Matrix Multiplication

One of the most common mistakes when working with NumPy and linear algebra is trying to multiply arrays whose **dimensions are not compatible**.

Let's first look at what happens when we try to multiply two matrices with incompatible shapes:

```
A = np.zeros((3, 2))
B = np.zeros((4, 5))
print(A @ B)
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-86-c1fcd344e478> in <module>
      1 A = np.zeros((3, 2))
      2 B = np.zeros((4, 5))
----> 3 print(A @ B)


ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, with
gufunc signature (n?,k),(k,m?)->(n?,m?) (size 4 is different from 2)
```

That error message looks complicated, but for our purposes we only need to focus on the key part:

```
size 4 is different from 2
```

This tells us that:

- `A` has shape (3, 2) -> it has **2 columns**
- `B` has shape (4, 5) -> it has **4 rows**

For matrix multiplication `A @ B` to work, the number of columns of `A` must match the number of rows of `B`.

Here, $2 \neq 4$, so the multiplication is not defined, and NumPy raises an error.

A simple and effective strategy is to print the shapes of your arrays before and after each matrix multiplication and check that they are compatible:

```
print(A.shape)
print(B.shape)
```

Exercise 9: Practicing Code Debugging

We would advice you to use np.outer() and np.inner() when computing the dot product of 1D arrays. If X is a vector (represented as a 1D array in this course), then `np.inner(X, X)` calculates $X^T \cdot X$ (the regular dot product) and `np.outer(X, X)` computes $X \cdot X^T$.

If you are performing matrix multiplication between a 2D array and a 1D array, we would advise you to first reshape the 1D array into a 2D array of shape $(d, 1)$.

In this exercise, you will practice **debugging NumPy code** and using **tests** to verify that your implementation is correct. The goal is not only to fix the code, but also to **understand why it was wrong** and how to reason about array shapes and operations.

Before starting, keep in mind the following useful NumPy functions:

- `np.inner()`: computes the dot product of two 1D arrays.
  If `X` is a vector (represented as a 1D array in this course), then `np.inner(X, X)` computes (X^T X), the usual dot product.

- `np.outer()`: computes the outer product.
  For a vector `X`, `np.outer(X, X)` computes (X X^T).
- When multiplying a **2D array** by a **1D array**, it is often helpful to first reshape the 1D array into a **column vector** with shape `(d, 1)`.

The function below is intended to compute the **sum of the maximum element of each row** of a 2D NumPy array. However, there is a **bug** in the implementation.

Your task is to **find and fix the bug** so that the function passes the provided tests.

```
import numpy as np

def sum_of_max(A):
    """
    Computes the sum of the maximum element of each row of A.

    A must be a 2D NumPy array.
    """
    return np.sum(np.max(A, axis=1))
```

The following tests describe the expected behavior of the function. After you fix the implementation, both tests should pass.

```
A = np.array([[1, 2],
              [3, 4]])
np.testing.assert_allclose(sum_of_max(A), 6)

A = np.array([[24, 69, 83],
              [74, 14, 27]])
np.testing.assert_allclose(sum_of_max(A), 157)
```

Wrap-Up and Next Steps

Congratulations on making it to the end of this notebook! This was a long and demanding journey, and completing it is a great achievement.

By now, you should feel more comfortable working with **NumPy arrays**, performing **basic linear algebra operations**, and writing **cleaner, more reliable code** using **tests and assertions**. You have also practiced **debugging strategies** that will help you identify and fix problems more systematically in your future projects.

Remember that learning programming and numerical computing is a **gradual process**. You do not need to memorize everything—what matters most is that you now know:

- How to explore and use the documentation
- How to test your code
- How to debug problems when things go wrong

As a next step, you are encouraged to:

- Revisit the exercises and try to solve them again without looking at your previous solutions
- Modify the examples and see how the behavior changes
- Apply these tools to your own projects or to more advanced topics such as data analysis, machine learning, or scientific computing

Keep experimenting, keep breaking things (on purpose!), and keep learning. That's how you really become confident with NumPy and Python.
