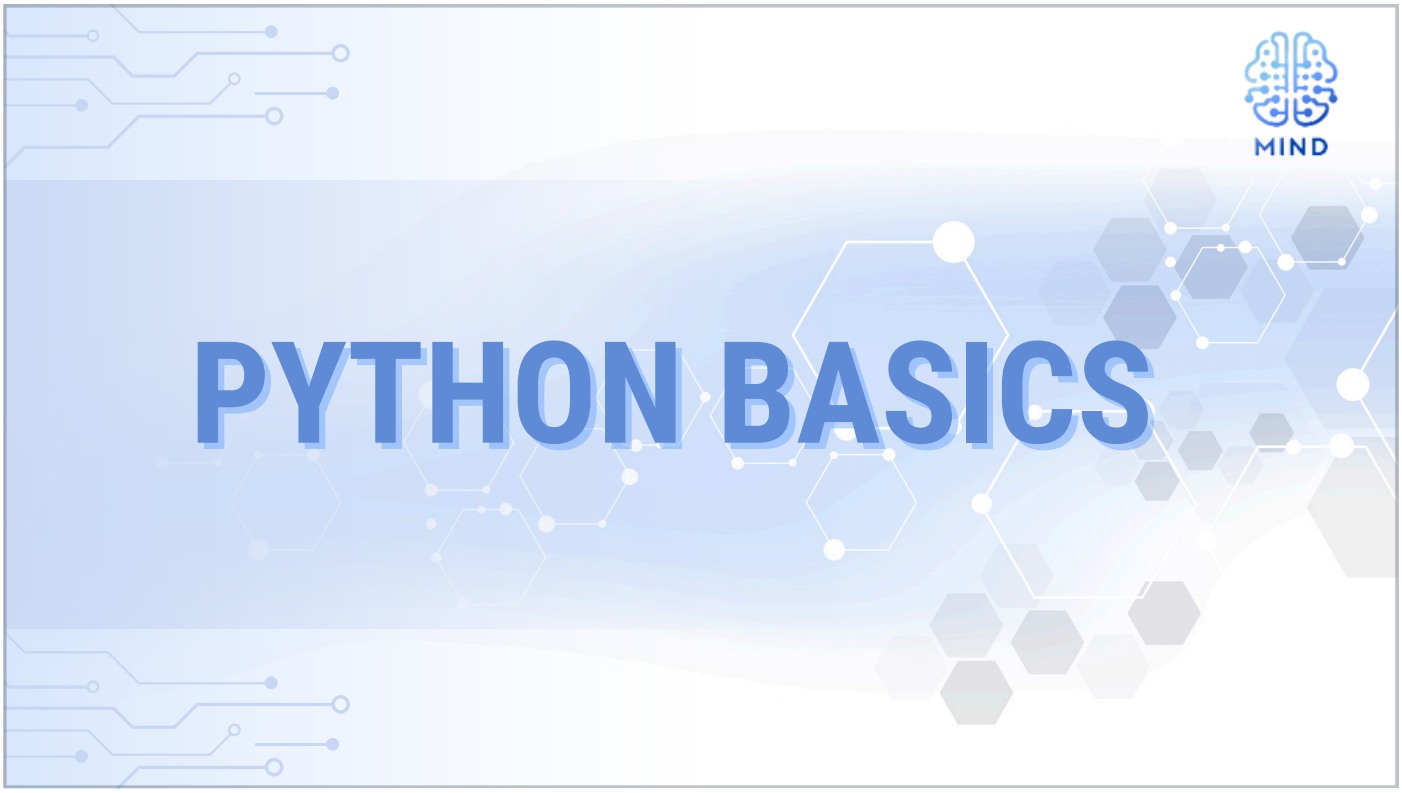

MIND
MIND
PYTHON WORKSHOP

# TABLE OF CONTENTS



MIND


MIND
PYTHON BASICS

# PYTHON STRUCTURE AND FORMAT

## VARIABLES AND DATA TYPES

First, what is a data type??

- Integer **[int]** - numbers **without** decimal points (ex. 5)
- Float **[float]**- numbers **with** decimal points (ex. 5.0)
- String **[””]** - text which requires **quotations** around it (ex. “John”)
  - The quotes allow the computer to know its word data and not an instruction
- Booleans **[bool]** - represents either **True** or **False**

# HOW TO PRINT


- To print something in Python, we simply use the print() function
- Let's try to print "Hello World!"
- Remember, Hello World! is a **STRING**, so we have to put quotes around it!
- ```
  3    print("Hello World!")
  ```

```
C:\Users\taham\PycharmProjects\NeuroCourse\.venv1\Scripts\python.exe C:\Users\taham\PycharmProjects\Beginning\Personal\main.py
Hello World!

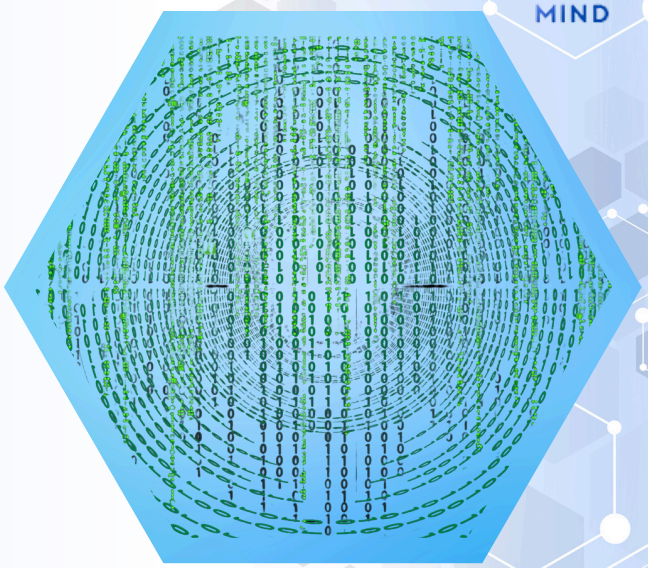
Process finished with exit code 0
```

- **Note**: Python is pretty smart, so if you put in other data types like floats, integers, and booleans, we can still print them, but Python autoconverts them to a string beforehand.
```

# PYTHON OPERATORS

## LOGICAL OPERATORS

- **And [and]** - returns True if both statements are true (ex. x < 5 AND x < 10)
- **Or [or]**- returns True if one of the two statements are true (ex. x < 4 OR x < 7)
- **Not [not/!]** - reverses result, if you make a True statement, it will return False and vice versa  (ex. not x < 5 )



# PYTHON OPERATORS

## ARITHMETIC OPERATORS

- **Addition [+]** - adds numbers (ex. 1 + 1 = 2)
- **Subtraction [-]** - subtracts numbers (ex. 2 - 1 = 1)
- **Multiplication [*]** - multiplies numbers together (ex. 5 * 2 = 10)
- **Division [/]** - divides numbers together and returns a float (ex. 4 / 2 = 2.0)
- **Floor Division [//]** - divides numbers together and returns integer (ex. 5 / 2 = 2)
- **Modulus/Remainder Division [%]** - divides numbers together and returns remainder as an integer (ex. 5 % 2 = 1)
- **Exponents [**]** - multiplies number based on exponents (e. 5 ** 2 = 25)

# PYTHON OPERATORS

MIND

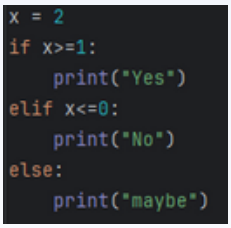
## Comparison Operators

- **Equals [==]** - returns True if equal (ex. 4 == 2 * 2)
- **Not Equal [!=]** - returns True if not equal (ex. 4 != 2 * 1)
- **Greater than [>]** - returns True if compared number is greater (ex. 5 > 2)
- **Less than [<]** - returns True is compared number is less than (ex. 2 < 5)
- **Greater than or equal to [>=]** - returns True if compared number is greater than or equal to (ex. 5 >= 2)
- **Less than or equal to [<=]** - returns True if compared number is less than or equal to (ex. 5 <= 6)

# PYTHON STATEMENTS

## CONDITIONAL STATEMENTS

- **If Statement** **[if]** - executes a block of code if the given condition is true (ex. if x>=1: print(“MIND”))
- **Else If Statement** **[elif]** - executes a block of code if another given condition is true (ex. elif x == 1: print(“Neuro”))
- **Else Statement** **[else]** - executes a bock of code if no other pre-stated conditions are met (ex. else: print(“tech”))



```
x = 2
if x>=1:
    print("Yes")
elif x<=0:
    print("No")
else:
    print("maybe")
```

# PYTHON STATEMENTS

## CONDITIONAL STATEMENTS

- **If Statement** **[if]** - executes a block of code if the given condition is true (ex. if x>=1: print(“MIND”))
- **Else If Statement** **[elif]** - executes a block of code if another given condition is true (ex. elif x == 1: print(“Neuro”))
- **Else Statement** **[else]** - executes a bock of code if no other pre-stated conditions are met (ex. else: print(“tech”))

```
x = 2
if x>=1:
    print("Yes")
elif x<=0:
    print("No")
else:
    print("maybe")
```

```
C:\Users\Raafi\anaconda3\python.exe C:\Users\Raafi\PyCharmProjects\main.py
Yes
```

# PYTHON LOOPS

For **[for]** Loops:

- used for iterating over a sequence
- allows us to execute a set of statements for every item in a list, tuple, set, etc.
- For loops can be structure in various ways, so we can loop through lists (see left) or we can loop through a number until it reaches some condition to stop (see right)

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

```
for i in range(1, 10, 1):

for i in range(start, stop, step_amount)
```

# PYTHON LOOPS

For **[for]** Loops:

- used for iterating over a sequence
- allows us to execute a set of statements for every item in a list, tuple, set, etc.
- For loops can be structure in various ways, so we can loop through lists (see left) or we can loop through a number until it reaches some condition to stop (see right)

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

```
for i in range(1, 10, 1):

for i in range(start, stop, step_amount)
```

```
PS C:\Users\Raafi> & C:/Users/Raafi/AppData/Local/Programs/Python/Python312/python.exe c:/Users/Raafi/VSCodeProjects/MIND
DEMO.py
apple
banana
cherry
```

# PYTHON LOOPS

While **[while]** Loops:

- can execute a set of statements as long as a condition is true, over and over

```
x = 2
while x <= 5:
    print("x is ", x)
    x+=1
```

```
# while (condition):
#       body
```

# PYTHON LOOPS

While **[while]** Loops:

- can execute a set of statements as long as a condition is true, over and over

```
x = 2
while x <= 5:
    print("x is ", x)
    x+=1
```

```
# while (condition):
#       body
```

```
PS C:\Users\Raafi> & C:/Users/Raafi/AppData/Local/Programs/Python/Python312/python.exe c:/Users/Raafi/VSCodeProjects/MIND
DEMO.py
x is  2
x is  3
x is  4
x is  5
```

# PYTHON FUNCTIONS

MIND

- Reusable blocks of code that runs when called upon
- Define **[def]** - uses def keyword to create function

```
def my_function():
  print("MIND Workshop Demo")

my_function()
```

```
PS C:\Users\Raafi> & C:/Users/Raafi/AppData/Local/Programs/Python/Python312/python.exe c:/Users/Raafi/VSCodeProjects/MIND
DEMO.py
MIND Workshop Demo
```

# PYTHON FUNCTIONS

- What if you need some information to execute that particular block of code? Like a formula sometimes functions need to take in information, these are called parameters.

- You define parameters in those empty brackets after the functions name.
  - Ex. def my_function**( Enter parameters )**

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil", "Refsnes")
```

- You don't have to specify the type of each parameter when you add parameters to your custom function. HOWEVER, remember what you intend their types to be, otherwise, you might end up trying to multiply a string or divide a list, which will cause an error in your program

# PYTHON LISTS AND TUPLES

**Lists**

- Used to store multiple items in a single variable list data type
- It can store different types all together, so strings, booleans, floats and integers can all be in the same list
- You can change it after you first make it, this is called it being **mutable**
- Uses square brackets to enclose list [ ]

**Tuples**

- The exact same propeties as list except, once they are made, you CANNOT change them, they are **immutable**.
- Uses circular brackets to enclose a tuple ( )

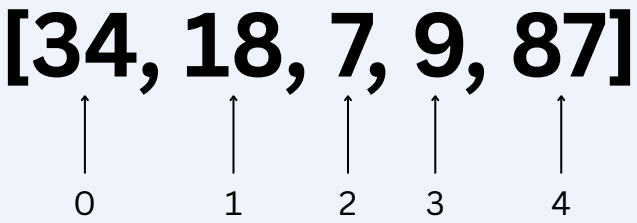
```
mylist = [1, 2, 3, 4, 5]

print(mylist)
```

```
PS C:\Users\Raafi> & C:/Users/Raafi/AppData/Local/Programs/Python/Python312/python.exe c:/Users/Raafi/VSCodeProjects/MIND
DEMO.py
[1, 2, 3, 4, 5]
```

# INDEXING IN PYTHON

- Refers to the process of accessing a specific element in a sequence, such as a string or list, using its position or index number
- Starts at 0




- Imagine you have a list or tuple called **data = [10, 20, 30, 40, 50],** to index into a specific element via code you must do **name[index],** so for data if we want to access the first element we do, **data[0]**, and this goes up to the last element which would be **data[4]**.

# LET'S TRY!

# TASK 1

**Write a function sum_of_numbers that takes a list from 1 to 5 and returns the sum of the numbers. Be sure to print the sum so we can see! (Hint: use a for loop)**

# TASK 2

**Create a list of names and use a loop to print each name with a greeting**

# TASK 3

**Write a function count_vowels(string) that counts the number of vowels (a, e, i, o, u) in a given string. Set the string to your first name.**


MIND
NUMPY

# NUMPY - WHAT IS IT?


- Stands for Numerical Python
- Powerful library built into python for quick  numerical and scientific computations
- Useful for tasks involving data analysis, statistics, linear algebra and scientific computations due to its high performance and ease of use.
- Array object in NumPy is called **ndarray**.

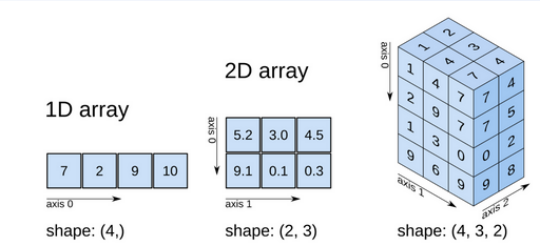
# WHY USE NUMPY?

- We have lists in Python that serve the purpose of arrays but can be inefficient to process.
- Up to 50x faster than traditional python lists.
- Very frequently used in data science, where running time and space complexity are very important.
- Supporting functions are porivded that make working with ndarray very easy.

# ARRAYS IN NUMPY


**How do they differ from lists?**

- Arrays are data structures that store collections of elements, **of the same type**, in a **contiguous block of memory**.
- Arrays have a **fixed size**, meaning you need to define how many elements the array will hold at the creation time.
- Elements in an array can be accessed using an index (starting from 0). This allows for fast access to any element in the array.



# NUMPY FUNDAMENTALS

MIND

## Importing numpy

```
import numpy as np
```

- When working with NumPy, it is an external package that is separate from regular Python, so we need to import it.
- The statement import numpy as np is used to shorten the reference to the package, making it easier to use throughout the code. This way, instead of typing numpy every time, we can simply use np.

# NUMPY FUNDAMENTALS

## Creating an array

```python
import numpy as np

# Creating a 1D array using np.array()
np_arr1 = np.array([1, 2, 3, 4, 5])
print("1D Array (np.array()):", np_arr1)
```

- To create an array, the simplest way is by using the np.**array([<elements>])** function, where you include the elements inside the square brackets.
- **Reminder:** These elements should be of the same type.

```
PS C:\Users\harri\MINDW1\NumpyExamples> py .\numpy_intro.py
1D Array (np.array()): [1 2 3 4 5]
```

# NUMPY FUNDAMENTALS

## Creating an array

```
# Creating a 1D array using np.arange()
np_arr2 = np.arange(1, 6)
print("1D Array (np.arange()):", np_arr2)
```

- In the second method, we use a function similar to the range function we saw earlier (recall for loops in python basics), called np.**arange(start, end)**.
- In this case, the end value is excluded from the output. For example, when using np.arange(1, 6), the resulting array will contain values from 1 to 5, excluding 6.

```
PS C:\Users\harri\MINDW1\NumpyExamples> py .\numpy_intro.py
1D Array (np.array()): [1 2 3 4 5]
1D Array (np.arange)): [1 2 3 4 5]
```

# NUMPY FUNDAMENTALS

## Creating an array

```
# Creating a 1D array using np.linspace()
np_arr3 = np.linspace( start: 1,  stop: 5,  num: 5)
print("1D Array (np.linspace()):", np_arr3)
```

- The third method is using np**.linspace(start, stop, num_samples)**. This function generates evenly spaced values within a specified range. The start value is included, while the stop value is excluded. You also need to specify the number of samples you want.

- For example, if the range is from 0 to 1 and you want 10 samples, it will create an array like [0.0, 0.1, 0.2, ..., 0.9, 1.0].

```
PS C:\Users\harri\MINDW1\NumpyExamples> py .\numpy_intro.py
1D Array (np.array()): [1 2 3 4 5]
1D Array (np.arange)): [1 2 3 4 5]
1D Array (np.linspace)): [1. 2. 3. 4. 5.]
```

# NUMPY FUNDAMENTALS

## Array Arithmetic

```
# Creating two 1D arrays
arr1 = np.array([1,2,3,4])
arr2 = np.array([10,20,30,40])

#Addition
sum_arr = arr1 + arr2
print("Sum of Two Arrays:", sum_arr)
```

```
PS C:\Users\harri\MINDW1\NumpyExamples> py .\numpy_intro.py
Sum of Two Arrays: [11 22 33 44]
```

- In the first example, we can add arrays together. NumPy performs this operation element by element, meaning it adds the values at the same index from each array. This is known as **element-wise addition**.

```
import numpy as np

# Creating two 1D arrays
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([10, 20, 30, 40])

# Product
mul_arr = arr1 * arr2
print("Product of Two Arrays:", mul_arr)
```

```
PS C:\Users\harri\MINDW1\NumpyExamples> py .\numpy_intro.py
Product of Two Arrays: [ 10  40  90 160]
```

- In the second example, NumPy again follows element-wise operations, but here it multiplies the elements at the same index from each array together called **element-wise multiplication.**

# NUMPY FUNDAMENTALS

## Array Arithmetic



```
import numpy as np

# Creating two 1D arrays
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([10, 20, 30, 40])

# Subtraction
sub_arr = arr2 - arr1
print("Subtraction of Two Arrays:", sub_arr)
```

```
PS C:\Users\harri\MINDW1\NumpyExamples> py .\numpy_intro.py
Subtraction of Two Arrays: [ 9 18 27 36]
```

- Here it is performing element-wise operations just **via element-wise subtraction.**

```
import numpy as np

# Creating two 1D arrays
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([10, 20, 30, 40])

# Division
div_arr = arr2 / arr1
print("Division of Two Arrays:", div_arr)
```

```
PS C:\Users\harri\MINDW1\NumpyExamples> py .\numpy_intro.py
Division of Two Arrays: [10. 10. 10. 10.]
```

- In the bottom image, it's performing **element-wise division.**
- You can perform similar operations via other operators as well (modulus division, exponents, etc.), and it will all be done through the element-wise method.

# NUMPY FUNDAMENTALS

## OTHER OPERATORS

| Operators | Description |
|---|---|
| np.sqrt(x) | Square root |
| np.sin(x) | Element-wise sin |
| np.cos(x) | Element-wise cos |
| np.log(x) | Element-wise natural log |
| np.dot(x) | Dot product |

# BROADCASTING


- Powerful feature that allows NumPy to perform element operations on arrays of different shapes and sizes.
- Automatically expands the smaller array to match the larger array to allow operations between arrays of different dimensions.
- Allows for fast element-wise operations between arrays without the need for explicit loops.

# BROADCASTING EXAMPLES

## Scalar and Array

```
import numpy as np

arr = np.array([1,2,3,4])
scalar = 10

result1 = arr +scalar
print("Scalar and Array:", result1)
```

```
PS C:\Users\harri\MINDW1\NumpyExamples> py .\numpy_intro.py
Scalar and Array: [11 12 13 14]
```

- In the code shown, the scalar value 10 is broadcasted across the entire array. This means that the scalar is treated as if it is added to each individual element of the array.
- Specifically, the operation arr + scalar results in adding 10 to each element of the array [1, 2, 3, 4], producing [11, 12, 13, 14].

# BROADCASTING EXAMPLES

## Arrays of Different Shapes

```python
import numpy as np

arr1 = np.array([1, 2, 3])              # Shape (3,)
arr2 = np.array([[10], [20], [30]])     # Shape (3, 1)

result2 = arr1 + arr2
print("Arrays of Different Sizes:\n", result2)
```

```
PS C:\Users\harri\MINDW1\NumpyExamples> py .\numpy_intro.py
Arrays of Different Sizes:
 [[11 12 13]
 [21 22 23]
 [31 32 33]]
```

In this example, arr2 has a shape of (3, 1), while arr1 has a shape of (3). When performing the addition arr1 + arr2, NumPy will automatically broadcast arr1 to match the shape of arr2. This means that arr1 will be "stretched" or replicated across the rows of arr2.

arr1 (stretched):
[[1 2 3],
[1 2 3],
[1 2 3]]

arr2:
[[10],
[20],
[30]]

Now, the two arrays can be added element-wise, where each element in arr1 is added to the corresponding element in arr2, resulting in the broadcasted operation.

# INDEXING

- NumPy provides powerful tools for indexing which are essential for accessing and modifying array elements.
- Useful for signal processing.

## Integer Indexing

```python
import numpy as np

arr = np.array([10, 20, 30, 40, 50])

#Accessing the first element
print("Integer Indexing First Element:", arr[0])

# Access the last element (index -1)
print("Integer Indexing Last Element:", arr[-1])
```

```
PS C:\Users\harri\MINDW1\NumpyExamples> py .\numpy_intro.py
Integer Indexing First Element: 10
Integer Indexing Last Element: 50
```

## Boolean Indexing

```python
import numpy as np

arr = np.array([10, 20, 30, 40, 50])

# Create a boolean mask where elements are greater than 25
mask = arr > 25
print("Boolean Indexing Elements Greater than 25:", arr[mask])
```

```
PS C:\Users\harri\MINDW1\NumpyExamples> py .\numpy_intro.py
Boolean Indexing Elements Greater than 25: [30 40 50]
```

# SLICING

## 1D Array

```
array[start:stop:step]
```

```python
import numpy as np

arr = np.array([10, 20, 30, 40, 50])

# Slice from index 1 to 3 (exclusive of 3)
print("Index 1-3:", arr[1:3])
```

- The slice arr [1:3] starts at index 1 (which is the value 20) and ends just before index 3 (which is the value 40), thus extracting the portion [20, 30].
- The **stop index 3 is exclusive**, meaning it does not include the value at index 3.

```
PS C:\Users\harri\MINDW1\NumpyExamples> py .\numpy_intro.py
Index 1-3: [20 30]
```

# SLICING

## 1D Array

```
array[start:stop:step]
```

```
import numpy as np

arr = np.array([10, 20, 30, 40, 50])

# Slice from index 1 to 3 (exclusive of 3)
print("Index 1-3:", arr[1:3])
# Slice from index 2 to the end
print("Index 2-end:", arr[2:])
```

```
Index 2-end: [30 40 50]
```

- The slice arr[2:] extracts elements starting from index 2 to the end of the array.
- 2 is the starting index, and the ":" indicates continuation until the last element.
- Omitting the stop value means all elements from index 2 onward are included. In other words, the **default value when no stop index is included is to go to the end of the array.**

# SLICING

## 1D Array

```
array[start:stop:step]
```

```
import numpy as np

arr = np.array([10, 20, 30, 40, 50])

# Slice from index 1 to 3 (exclusive of 3
print("Index 1-3:", arr[1:3])
# Slice from index 2 to the end
print("Index 2-end:", arr[2:])
# Slice from the beginning to index 3
print("Index beginning-3:", arr[:3])
```

```
Index beginning-3: [10 20 30]
```

- The slice arr[:3] extracts elements from the beginning of the array up to, but not including, index 3.
- The ":" before 3 means no start index is specified, so it defaults to index 0. In other words, **the default value when no start index is included to go the the beginning of the array.**
- 3 is the stop index, meaning the slice excludes index 3.

# SLICING


## 1D Array

```
array[start:stop:step]
```

```
import numpy as np

arr = np.array([10, 20, 30, 40, 50])

# Slice from index 1 to 3 (exclusive of 3)
print("Index 1-3:", arr[1:3])
# Slice from index 2 to the end
print("Index 2-end:", arr[2:])
# Slice from the beginning to index 3
print("Index beginning-3:", arr[:3])
# Slice with a step of 2
print("Step of 2:", arr[::2])
```

```
Step of 2: [10 30 50]
```

- The slice arr[::2] selects elements from the entire array with a step of 2.
- The double colons "::" indicate no specified start or stop index, so the **slice includes the entire array**. This is because both start and stop are set to their default values which are the beginning and end of the array respectively.
- 2 is the **step** size, meaning every **second element** is selected.
  - Something to note now is we can see from the past examples, **the default step is 1.**

# SLICING

## 2D Array

Rows , Columns

```
array[start:stop:step, start:stop:step]
```

```
import numpy as np

arr_2d = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Slice rows starting from index 1 and columns from index 1
print("Row and column index starting from 1:\n",arr_2d[1:, 1:])
```

```
Row and column index starting from 1:
 [[5 6]
 [8 9]]
```

- The slice arr_2d[1: , 1:] extracts a subarray, removing the first row and first column from the original 2D array.
- The 1: in the row index position means the slice starts from row index 1, including all remaining rows.
- The 1: in the column index position means the slice starts from column index 1, including all remaining columns.
- This effectively removes the top-left section of the array, leaving only the bottom-right portion.

# NUMPY RESHAPING

- Reshaping in NumPy is the process of changing the shape of an array without changing its data.
- Particularly useful when you need to manipulate data for certain operations, like aligning data for matrix multiplication or preparing data for machine learning models.
- The **.reshape** function is applied **to an array** and has parameters depending on how you want to reshape it (how many dimensions the new array should be)

```
arr.reshape(dim1, dim2, ..., dimN)
```

```python
import numpy as np

# Creating a 1D array
arr = np.array([1, 2, 3, 4])

# Reshaping the 1D array into a 2D array with shape (2, 2)
reshaped_arr = arr.reshape(2, 2)

print("Reshaped Array:\n", reshaped_arr)
```

```
PS C:\Users\harri\MINDW1\NumpyExamples> py .\numpy_intro.py
Reshaped Array:
 [[1 2]
 [3 4]]
```

```python
import numpy as np

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

# Let NumPy calculate the number of rows
reshaped_arr = arr.reshape(-1, 2)
print(reshaped_arr)

print("Reshaped Array:\n", reshaped_arr)
```

```
PS C:\Users\harri\MINDW1\NumpyExamples> py .\numpy_intro.py
[[1 2]
 [3 4]
 [5 6]]
Reshaped Array:
 [[1 2]
 [3 4]
 [5 6]]
```

# AGGREGATING VALUES

- Aggregation functions in NumPy compute summary statistics of an array, such as mean, sum, max, and min.
- These operations are crucial for data analysis and signal processing, helping to extract key characteristics.
- They provide insights into the distribution and trends of numerical data.
- We will revisit some of these concepts in SciPy stats when we look into introductory statistical analysis.

```python
import numpy as np

# Creating a 1D array
arr = np.array([1, 2, 3, 4, 5])

# Calculate the mean
print("Mean:", np.mean(arr))

# Sum of elements
print("Sum:", np.sum(arr))

# Max and Min
print("Max:", np.max(arr))
print("Min:", np.min(arr))
```

```
PS C:\Users\harri\MINDW1\NumpyExamples> py .\aggregation_example.py
Mean: 3.0
Sum: 15
Max: 5
Min: 1
```

# LET'S TRY!

# TASK

Create a numpy array called num_array with 10 elements (10, 20, 30, ..., 100)

After generating the array, reshape it into a 2D array with 2 rows and 5 columns.

Once the array is reshaped, print it and then calculate and display the mean, sum, maximum, and minimum of the values within the array.


**Hint:** Use aggregation functions


MIND
SCIPY

# SCIPY - WHAT IS IT?

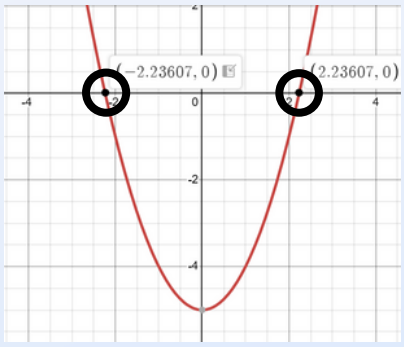
- Stands for **Scientific Python**
- Extends **NumPy** and provides more advanced functions for scientific and technical computing
- Includes many modules, but we will focus on **optimization, interpolation**, and **statistics**

# WHY IS IT USEFUL?

- **Smooth Data Visualization**: When we want a smooth line rather than choppy data points, interpolation helps generate intermediate values.
- **Enhanced Analysis**: With more points, we can better understand trends and behaviors in the data that might not be visible with only a few points.

# MATH REFRESHER

- A **root** is a value for which a given function equals zero
- These are also known as points where the function crosses the x-axis

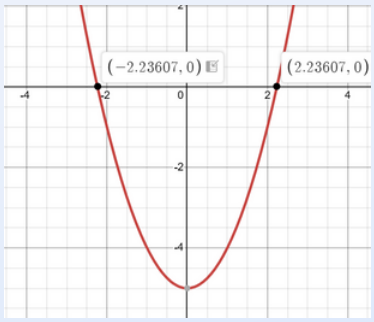


# SCIPY OPTIMIZATION


- Set of procedures defined in SciPy that either find the minimum value of a function, or the root of an equation.

```python
from scipy import optimize
# Define a function
def equation(x):
    return x**2 - 5
# Find the root of the equation (i.e., where f(x) = 0)
root = optimize.root_scalar(equation, bracket=[0, 3])
print("Root:", root.root)
```

```
C:\Users\Raafi\PyCharmProjects\.venv\Scripts\python.exe C:\Users\Raafi\PyCharmProjects\main.py
Root: 2.2360679774997925

Process finished with exit code 0
```

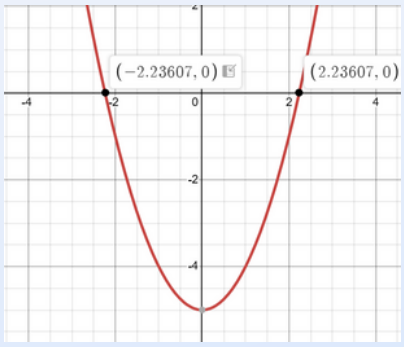


# SCIPY OPTIMIZATION

- So in scipy.optimize we can use a function called **root_scalar(function, bracket = [] )**
- So this function will calculate the missing variable for any scalar equation. (A scalar equation is an equation such that the solution is a single number which is also called a scalar in mathematics)

```
from scipy import optimize
# Define a function
def equation(x):
    return x**2 - 5
# Find the root of the equation (i.e., where f(x) = 0)
root = optimize.root_scalar(equation, bracket=[0, 3])
print("Root:", root.root)
```

```
C:\Users\Raafi\PyCharmProjects\.venv\Scripts\python.exe C:\Users\Raafi\PyCharmProjects\main.py
Root: 2.2360679774997925

Process finished with exit code 0
```



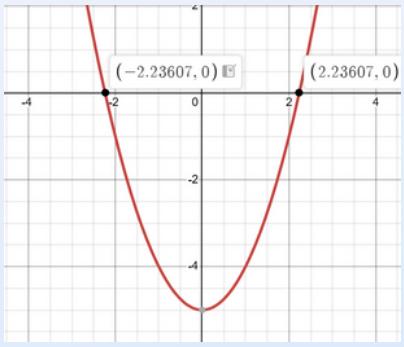
# SCIPY OPTIMIZATION

- The **function parameter** is essentially going to be your equation, it just needs to be expressed as function, for an better understanding see the code below and see how we converted y= x^2 - 5 into a function.

- The **bracket=[] parameter** is essentially the domain of where we want to find the root or solution, so here the equation has a negative and a positive solution, but we set the domain to positive numbers so only the positive solution comes in the output.

```
from scipy import optimize
# Define a function
def equation(x):
    return x**2 - 5
# Find the root of the equation (i.e., where f(x) = 0)
root = optimize.root_scalar(equation, bracket=[0, 3])
print("Root:", root.root)
```

```
C:\Users\Raafi\PyCharmProjects\.venv\Scripts\python.exe C:\Users\Raafi\PyCharmProjects\main.py
Root: 2.2360679774997925

Process finished with exit code 0
```



# SCIPY INTERPOLATION

- Interpolation is a method that allows us to "fill in" values between known data points, estimating what happens in the gaps.
- This is especially useful in signal processing because data is often sampled at discrete intervals.
- With interpolation, we can create a smoother curve, helping us visualize or analyze the data as though it were continuous.

# TYPES OF INTERPOLATIONS


1. **Linear Interpolation:** This method draws a straight line between two points and finds the value on that line.
2. **Polynomial Interpolation:** This fits a curve (a polynomial) to the data, which works well for data that changes unpredictably and helps create smoother estimates.
3. **Spline Interpolation:** This uses smooth curves made of small pieces (usually cubic) to connect all data points without sharp turns.
4. **Nearest Neighbor Interpolation:** This method takes the value of the nearest data point for the unknown value. It's simple but can create uneven, abrupt changes.
5. **Barycentric Interpolation:** This is a more stable method used for large sets of data, helping with better accuracy when doing polynomial interpolation.

# MATPLOTLIB CRASH COURSE

- Matplotlib is a data visualization package that allows you to visualize data through graphing them. We plan on teaching it in more detail in our next workshop, but we need to know just a little bit for one of interpolation, as its hard to see interpolation working without it being plotted.
- **Importing Libraries:** import matplotlib.pyplot as plt imports the pyplot module from Matplotlib, providing an easy interface to create and display plots.
- **Plotting Data:** plt.plot(X_AXIS, Y_AXIS, '-', label="") is used to plot the data.
  - **X_AXIS and Y_AXIS** represent the array of values to be plotted on the x and y axis respectively,
  - **'-'** specifies using a solid line to connect the points.
  - The **label =** argument is used to assign a name to the line, which can later be displayed in a legend.

```
import matplotlib
# matplotlib.use('TkAgg')                          # Use TkAgg backend for interactive plotting
import matplotlib.pyplot as plt


plt.plot( *args: X_AXIS, Y_AXIS, '-', label="")  # '-' gives a smooth line
plt.legend()
plt.show()
```

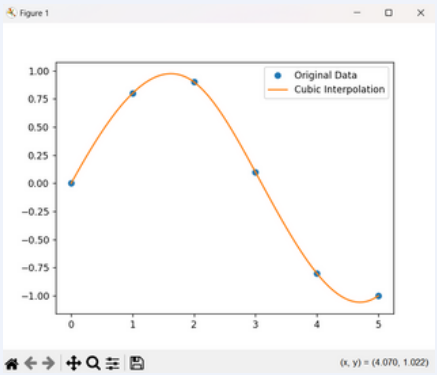
# MATPLOTLIB CRASH COURSE

- **Adding a Legend:** The plt.legend() function adds a legend to the plot, which uses the labels specified in the label argument of plt.plot() to differentiate different data series.
- **Displaying the Plot:** plt.show() displays the plot in an interactive window, allowing you to visualize the graph.
- **Customization Options:** You can further customize the plot by adding titles, axis labels, and gridlines. For example, plt.title("Title of the Graph"), plt.xlabel("X-axis label"), plt.ylabel("Y-axis label"), etc.
- This should be all you need for interpolation, in our second workshop, we will do dive deeper into matplotlib and all of its fun aspects!

```
import matplotlib
# matplotlib.use('TkAgg')                          # Use TkAgg backend for interactive plotting
import matplotlib.pyplot as plt


plt.plot( *args: X_AXIS, Y_AXIS, '-', label="")  # '-' gives a smooth line
plt.legend()
plt.show()
```

# INTERPOLATION CONT'D

```python
import numpy as np
from scipy import interpolate
import matplotlib
matplotlib.use('TkAgg')                          # Use TkAgg backend for interactive plotting
import matplotlib.pyplot as plt

# Known data points (e.g., measurements or observations)
x = np.array([0, 1, 2, 3, 4, 5])
y = np.array([0, 0.8, 0.9, 0.1, -0.8, -1])

# Create an interpolation function based on known data points
f = interpolate.interp1d(x, y, kind='cubic')

# Generate new x values for a smooth curve by filling in the gaps
x_new = np.linspace( start: 0,  stop: 5,  num: 100)        # Creates 100 points between 0 and 5
y_new = f(x_new)                                 # Use the interpolation function to get y values for new x points

# Plot original data points and the interpolated curve
plt.plot( *args: x, y, 'o', label="Original Data")  # 'o' shows the original points
plt.plot( *args: x_new, y_new, '-', label="Cubic Interpolation")  # '-' gives a smooth line
plt.legend()
plt.show()
```




- **interpolate.interp1d:** This function from SciPy's interpolation module lets us create an "interpolation function" from our original data points. Essentially, it builds a model that "connects" these points. Once we have this interpolation function, we can use it to estimate values for any new points in between, making our data appear more continuous and smooth.
- **kind='cubic':** The argument specifies the type of interpolation we want. Here, 'cubic' refers to cubic spline interpolation. Instead of connecting points with straight lines **(kind='linear')**, cubic interpolation uses a curve that smoothly passes through each point, like a flexible ruler bending to fit the data. This method often gives a natural-looking smoothness that's particularly helpful for visualizing data without sharp edges or breaks.

# SCIPY STATS

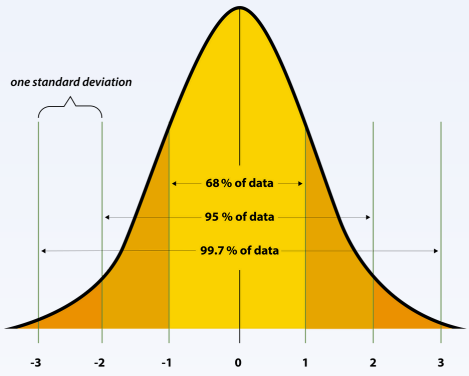
- SciPy's statistics module (scipy.stats) is a powerful part of the SciPy library that provides a wide range of statistical functions, tests, and distributions. It is particularly useful for data analysis, hypothesis testing, probability modeling, and various other statistical applications.
- It provides a wide variety of statstical functions, tests, and distributions that cover a broad range of data analysis tasks—from simple descriptive statistics to advanced hypothesis testing.
- SciPy’s statistical functions are optimized and built on top of NumPy, making them fast and efficient, especially when dealing with large datasets.

# SCIPY AND NUMPY STATISTICS

## STATISTICAL FUNCTIONS

- **Mean** [numpy.mean] - computes the arithmetic **average** of the given data
- **Variance** [numpy.var] - computes the variance of the given data (array elements), represents how much all the data points differed **from each other**.
- **Standard Deviation** [ math.sqrt(numpy.var) ] - the square root of variance is the standard deviation of the data. This represents the amount which all the data **differed from the mean/average.**
- **Normal Continuous Random Variables** [scipy.stats.norm] - A general family of functions with a probability distribution that is shaped like a bell curve. In the next slide, we will be using one of its functions useful in experimentation
  - You can think of a probability curve (like the one show here), as a specific mapping of probabilities to the outcomes on the x axis, and the probabilities are the chance of that outcome happening.

**Normal/Bell Curve**




Explain for the last bullet point, that the .norm is a class that is for various functions within the normal distribution, one of which we are going to use today which is to generate random samples using this probability distribution. This is useful as you can run simulations for any scenarios you want to test. An example is your professor trying to compute grades and see if your class is going to curve, they use functions like this to run simulations with different grades to see when students should have a curve, etc.

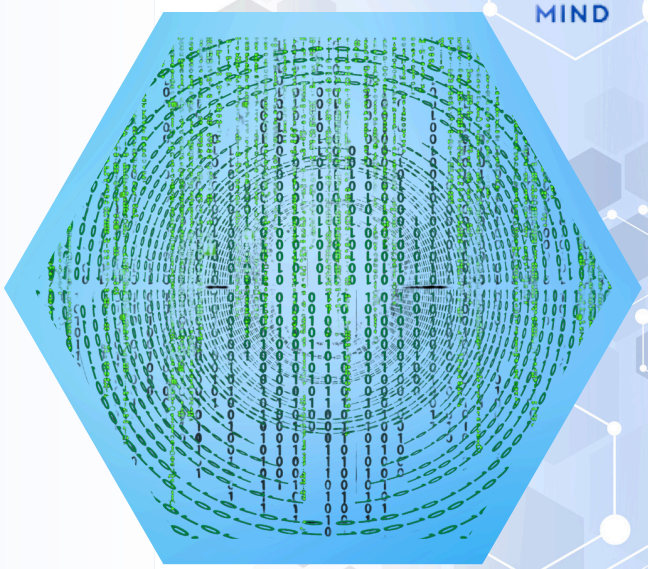
# STATS CONT'D

```python
import math
from scipy import stats
import numpy as np

# Sample data
data = np.array([2, 3, 5, 7, 11, 13, 17, 19, 23])

# Calculate mean and variance
mean = np.mean(data)
variance = np.var(data)

#sd is the square root of the variance
Standard_deviation = math.sqrt(np.var(data))

print("Mean:", mean)
print("Variance:", variance)

# Generate a random sample from a normal distribution
normal_sample = stats.norm.rvs(loc=0, scale=1, size=10)
print("Random Sample from Normal Distribution:", normal_sample)
```

```
PS C:\Users\harri\MINDW1\NumpyExamples> py .\statistics_example.py
Mean: 11.111111111111111
Variance: 49.432098765432101
Random Sample from Normal Distribution: [-1.21277567 -0.60865244 -0.57196097 -0.79977648 -0.27957122  1.69690097
  1.12816558  1.06859711  0.56407504 -0.79772577]
```

- **scipy.stats.norm.rvs()** generates random samples from a normal (Gaussian) distribution, commonly used in signal processing and statistical modeling.
- It creates a dataset of random variates, which are individual draws from the specified normal distribution.
- Parameters:
  - **loc**: The **mean** (μ) of the normal distribution, which determines the center of the generated values.
  - **scale**: The standard deviation (σ), which controls the spread or dispersion of the values around the mean.
  - **size**: The number of random values to generate, defining the shape of the output array.
- This function is useful for simulating real-world data, generating synthetic signals, or conducting simulated experiments.

# LET'S TRY!



# FINAL TASK

**Problem:**

A weather station records temperature readings in °C at different times of the day, but unfortunately, one reading was lost. Your task is to estimate the missing temperature and analyze temperature trends using mathematical techniques. Additionally, you will simulate random temperature fluctuations based on the recorded data and compute some statistical properties of these fluctuations.

**Steps to Complete the Problem:**

1. Estimate the temperature at 15 00 hours using a cubic interpolation plot
2. Simulate 10 random temperature variations for the day, assuming the temperature fluctuations follow a normal distribution. Use the mean and standard deviation of the recorded temperatures to define the normal distribution.
3. Calculate and print the standard deviation, mean, and variance of the simulated temperatures.
4. Print the plot.

**Data:**

- Times of the day (hours): [0, 6, 12, 18, 24]
- Temperatures (°C): [8, 14, 20, 16, 10]
- Assume temperature fluctuations follow a normal distribution with:
  - Mean centered at the recorded values
  - Standard deviation following the recorded values