# NumPy

- **What is NumPy?**
    - A Python library for numerical computing.
    - Provides support for working with large, multi-dimensional arrays and matrices.
    - Contains a variety of functions to perform mathematical and logical operations on arrays.
- **Why use NumPy?**
    - It is faster than regular Python lists for numerical operations.
    - Efficient for handling large datasets and complex calculations.
    - Commonly used in fields like data science, machine learning, and scientific computing.
- **Key Features:**
    - **Arrays**: Main feature; similar to Python lists but more powerful.
    - **Mathematical Operations**: Provides built-in functions for quick calculations (e.g., addition, subtraction, multiplication).
    - **Multi-Dimensional Support**: Can work with 1D, 2D (matrix), or even higher-dimensional arrays.
    - **Broadcasting**: Allows performing operations on arrays of different shapes.
- **Basic Example:**

```
import numpy as np
arr = np.array([1, 2, 3])
print(arr)
```

- **Installation**:

```
pip install numpy
```

- **Common Uses**:
    - Data processing and transformation.
    - Linear algebra, statistical analysis.
    - Efficient memory usage for large datasets.

# Why do we need arrays in python when we already have lists ?

Back in 2008, when machine learning and deep learning were emerging, Python wasn't the obvious choice for data scientists. Why? Because Python was considered slow, especially when working with large datasets. The internal data structures like lists and dictionaries were not optimized for numerical computations, making operations sluggish.

Then came a revolutionary development: someone created NumPy arrays. These arrays were implemented in C, a much faster language for numerical operations. They created a wrapper around these efficient arrays, making them accessible in Python. Why did they do it? Because while Python was slow in its native data structures, its syntax was simple and easy to understand.

So, NumPy arrays became a game-changer. They provided the speed of C with the simplicity of Python syntax, making Python a viable option for numerical computing tasks. This combination of speed and ease of use propelled Python into the forefront of data science and machine learning, leading to its widespread adoption in the field.

# Lists vs NumPy Arrays:

- **Data Type**:
    - **List**: Can store different data types (e.g., strings, integers, floats).
    - **NumPy Array**: Stores only one data type (e.g., all integers or all floats) for efficiency.
- **Speed**:
    - **List**: Slower for mathematical operations.
    - **NumPy Array**: Faster because it uses fixed-size memory and optimized functions for numerical operations.
- **Functionality**:
    - **List**: General-purpose, lacks built-in math functions.
    - **NumPy Array**: Designed for numerical operations, supports element-wise operations, matrix manipulations, etc.
- **Memory Efficiency**:

    - **List**: Takes more memory.
    - **NumPy Array**: More memory-efficient due to fixed data types.
- **Operations**:
    - **List**: Manual loops required for operations like addition.
    - **NumPy Array**: Supports direct vectorized operations (e.g., adding two arrays in one step).

In [1]:
```
# Importing Numpy
import numpy as np
```

In [2]:
```
# Speed Comparision between Numpy vs Lists

# list comprehensions
a = [i for i in range(10000000)]
b = [i for i in range(10000000,20000000)]
c = []
import time
start = time.time()
for i in range(len(a)):
    c.append(a[i]+b[i])

print(time.time()-start)
```

```
8.832384586334229
```

In [3]:
```
a = np.arange(10000000)
b = np.arange(10000000,20000000)
import time
start = time.time()
c = a+b
print(time.time()-start)
```

```
0.5782768726348877
```

In [4]:
```
3.953155040740967/0.24687910079956055
```

Out[4]:
```
16.012513930656716
```

## Creating NumPy Arrays (Theory):

In [5]:
```
# 1D Array
# nd.array
# n--> number of dimensions

# 1d array
a = [1,2,3,4]
a= np.array(a)
print(a)
```

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

In [6]:
```
a=[1,2,3]
b=[4,5,6]
c=[7,8,9]

np.array([a,b,c])
#excel
# matrix
```

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

In [7]:
```
# Creating 2D and 3D array
a=[1,2]
b=[3,4]
c=[5,6]
d=[7,8]
np.array([[a,b],[b,c]])
# images
# tensor
```

Out[7]:
```
array([[[1, 2],
        [3, 4]],

       [[3, 4],
        [5, 6]]])
```

In [8]:
```
a=[1,2]
b=[3,4]
c=[5,6]
d=[7,8]
np.array([[[a,b],[b,c]]])
```

Out[8]:
```
array([[[[1, 2],
         [3, 4]],

        [[3, 4],
         [5, 6]]]])
```

In [9]:
```
a=[1,2]
b=[3,4]
c=[5,6]
d=[7,8]
np.array([[[a,b],[b,c]]]).ndim
```

Out[9]:
```
4
```

1. **dtype**:
    - When creating a NumPy array, you can specify the data type ( `dtype` ) for the elements (e.g., integers, floats). This

ensures that all elements in the array have the same type, providing consistency and efficient memory use.

In [10]:
```
a=np.array([1,2,3,4,'5'])
b=np.array([1,2,3,4])
c=np.array([1.1,2,34])

print(a.dtype)
print(b.dtype)
print(c.dtype)
```

```
<U21
int64
float64
```

In [11]:
```
a=np.array([1,2,3,4,'5'],dtype='int64')
b=np.array([1,2,3,4])
c=np.array([1.1,2,34])

print(a.dtype)
print(b.dtype)
print(c.dtype)
```

```
int64
int64
float64
```

In [12]:
```
print('integer\t',np.array([1,2,3],dtype=int))
print('float\t',np.array([1,2,3],dtype=float))
print('string\t',np.array([1,2,3],dtype=str))
print('boolean\t',np.array([1,2,3],dtype=bool))
print('complex\t',np.array([1,2,3],dtype=complex))
```

```
integer         [1 2 3]
float         [1. 2. 3.]
string         ['1' '2' '3']
boolean         [ True  True  True]
complex         [1.+0.j 2.+0.j 3.+0.j]
```

2. **np.arange**:
    - Creates an array of evenly spaced values within a specified range. It's similar to Python's built-in `range()` function but returns an array. You can specify the start, stop, and step values.

In [13]:
```
# range vs arange
print(range(0,10),'\n') # --> python
print(np.arange(0,10)) # numpy
```

```
range(0, 10)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

3. **with reshape**:
    - After creating an array, you can use the `reshape()` method to change its shape (number of rows and columns) without changing the data. This is useful for organizing data into multi-dimensional arrays.

In [14]:
```
a=np.arange(0,10)
a.reshape(2,5)
# a.reshape(3,7)
a.reshape(5,2)
a.reshape(1,10)
```

Out[14]:
```
array([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]])
```

In [15]:
```
a=np.arange(1,13)
print(a.reshape(3,4))
print(a.reshape(4,3))
print(a.reshape(2,6))
print(a.reshape(1,12))
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]]
[[ 1  2  3  4  5  6  7  8  9 10 11 12]]
```

4. **np.ones and np.zeros**:
    - **np.ones** creates an array filled with ones, and **np.zeros** creates an array filled with zeros. You can specify the shape of the array (e.g., number of rows and columns).

In [16]:
```
np.zeros(3)
```

Out[16]:
```
array([0., 0., 0.])
```

In [17]:
```
np.ones(10,dtype=int).reshape(2,5)
```

Out[17]:
```
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]])
```

5. **np.random**:
    - Provides various functions to generate arrays filled with random numbers. You can create arrays with random integers, random floats, or samples from specific probability distributions.

```
In [18]: np.random.random(10)
```

```
Out[18]: array([0.60212636, 0.29503544, 0.7171469 , 0.12308647, 0.43374563,
       0.60204969, 0.67211498, 0.10156589, 0.44281613, 0.0098321 ])
```

6. **np.linspace**:
    - Creates an array of evenly spaced values between a specified start and stop value. Unlike `arange`, you specify the number of values to generate, not the step size.

```
In [19]: np.linspace(1,10,7)
```

```
Out[19]: array([ 1. ,  2.5,  4. ,  5.5,  7. ,  8.5, 10. ])
```

7. **np.identity**:
    - Creates a square identity matrix, where all diagonal elements are 1, and all other elements are 0. It is used in linear algebra applications.

```
In [20]: np.identity(3)
```

```
Out[20]: array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

8. **np.eye**:
    - Similar to `np.identity`, but allows you to specify the number of rows and columns separately, and you can also specify an offset for the diagonal.

```
In [21]: np.eye(4,3)
```

```
Out[21]: array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.],
       [0., 0., 0.]])
```

## Random Functions in NumPy:

1. **`randint`**:
    - Generates random integers from a specified range (inclusive of the lower bound, exclusive of the upper bound).
    - Syntax: `np.random.randint(low, high, size)`
    - Example: `np.random.randint(1, 10, 5)` generates 5 random integers between 1 and 9.

In [22]:
```
np.random.randint(0,10,10)
```

Out[22]:
```
array([0, 3, 2, 7, 4, 0, 8, 4, 1, 4], dtype=int32)
```

In [23]:
```
np.random.rand(0,100,13)
np.random.randint(0,101,50)
```

Out[23]:
```
array([14, 86, 73,  4, 35, 31, 71, 10, 99, 98, 34, 19, 87, 87,  9, 71,  9,
       34, 81, 93, 51, 87, 32, 99,  4, 21, 85, 37, 89,  8, 33, 95, 11, 57,
       41, 58, 85, 60, 52, 70, 24, 90, 99, 29, 39, 28, 67, 64,  2,  2],
      dtype=int32)
```

2. **`random_integers`**:
    - Generates random integers from a specified range (both bounds are inclusive).
    - Syntax: `np.random.random_integers(low, high, size)`
    - Example: `np.random.random_integers(1, 10, 5)` generates 5 random integers between 1 and 10.
    - **Note**: Deprecated in newer versions of NumPy. `randint` is preferred.

In [24]:
```
np.random.random_integers(0,5,10)
```

```
C:\Users\vipul\AppData\Local\Temp\ipykernel_11524\3932849665.py:1: DeprecationW
arning: This function is deprecated. Please call randint(0, 5 + 1) instead
  np.random.random_integers(0,5,10)
```

Out[24]:
```
array([1, 1, 1, 0, 5, 1, 4, 3, 0, 1], dtype=int32)
```

3. **`random`**:
    - Generates random floats between 0 and 1, following a uniform distribution.
    - Syntax: `np.random.random(size)`
    - Example: `np.random.random(5)` generates 5 random floats between 0 and 1.

In [25]:
```
np.random.random(10)
```

Out[25]:
```
array([0.59262307, 0.02786025, 0.35279806, 0.49837005, 0.81438146,
       0.45276792, 0.20132137, 0.26535981, 0.61272282, 0.80267417])
```

4. **`rand`**:
    - Generates random floats between 0 and 1, similar to `random()`, but can take multiple shape arguments.
    - Syntax: `np.random.rand(d0, d1, ..., dn)`
    - Example: `np.random.rand(2, 3)` generates a 2x3 array of random floats.

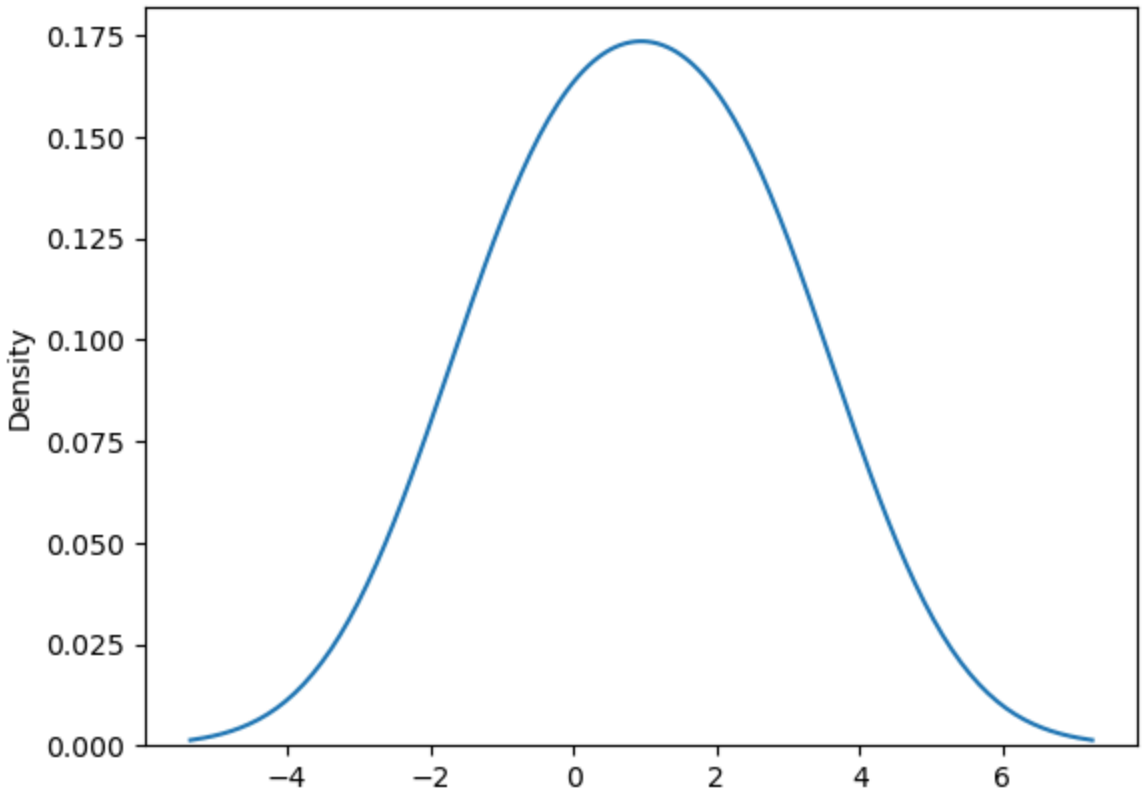
In [26]:
```
np.random.rand(3,4)*1000
```

Out[26]:
```
array([[ 48.65255583, 451.1424896 , 373.64873762, 658.25419699],
       [481.52344754, 777.63043178, 237.93131882, 471.18051612],
       [468.90935219, 114.05810805, 493.22391715, 322.0376047 ]])
```

5. **`randn`**:
    - Generates random numbers from a standard normal distribution (mean 0, variance 1).
    - Syntax: `np.random.randn(d0, d1, ..., dn)`
    - Example: `np.random.randn(3, 4)` generates a 3x4 array of random numbers from a normal distribution.

In [27]:
```
# np.std(np.random.randn(10))
import seaborn as sns
sns.kdeplot(np.random.randn(2))
```

Out[27]:
```
<Axes: ylabel='Density'>
```




## NumPy Array Attributes

In [28]:
```
# Sample Array
a=np.random.randint(0,100,10)
b=np.random.randint(0,100,9).reshape(3,3)
c=np.random.randint(0,10,8).reshape(2,2,2)
x=np.array(['Prince','Vipul','Gaurav'])
y=np.random.rand(9)

print(a)
print('-'*50)
print(b)
print('-'*50)
print(c)
print('-'*50)
print(x)
print('-'*50)
print(y)
```

```
[99 39 88 44 14 53 81 12 16 11]
--------------------------------------------------
[[ 5 37 21]
 [88 50 24]
 [ 0  4 81]]
--------------------------------------------------
[[[6 6]
  [1 6]]

 [[9 1]
  [2 0]]]
--------------------------------------------------
['Prince' 'Vipul' 'Gaurav']
--------------------------------------------------
[0.51825205 0.58783172 0.43561689 0.28837236 0.85149727 0.35113577
 0.89565041 0.69109532 0.95340381]
```

1. **ndim**:
    - Represents the number of dimensions (or axes) an array has. For example, a 1D array has 1 dimension, a 2D array has 2 dimensions (rows and columns), and so on.

In [29]:
```
print(a.ndim)
print(b.ndim)
print(c.ndim)
```

```
1
2
3
```

2. **shape**:
    - Describes the structure of the array in terms of its dimensions. It is represented as a tuple, where each value indicates the size of the array along a particular axis (e.g., rows and columns for a 2D array).

In [30]:
```
b.shape
c.shape
```

Out[30]: (2, 2, 2)

3. **size**:
    - Indicates the total number of elements present in the array, which is the product of the values in the shape tuple (e.g., number of rows multiplied by the number of columns).

In [31]:
```
a.size
```

```
b.size
x.size
```

Out[31]: 3

4. **itemsize**:
    - Refers to the memory size (in bytes) of each individual element in the array. The size depends on the data type of the array elements.

In [32]:
```
print(a.itemsize)
print(b.itemsize)
print(x.itemsize)
print(y.itemsize)
```

```
4
4
24
8
```

5. **dtype**:
    - Specifies the data type of the elements stored in the array, such as integers, floats, or complex numbers. It ensures consistency in the type of data being stored across the array.

In [33]:
```
print(a.dtype)
print(x.dtype)
print(y.dtype)
```

```
int32
<U6
float64
```

## Changing Data Type with `astype`

- **`astype`** is a NumPy method used to change the data type of an array's elements.
- It creates a **new array** with the desired data type while leaving the original array unchanged.
- Commonly used to ensure consistency of data types across an array, such as converting from **floats to integers**, **integers to strings**, or **integers to floats**.

In [34]:
```
# astype
```

In [35]:
```
a.astype('str')
a.astype('float')
a.astype('bool')
#a.astype('complex')
```

Out[35]:
```
array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
        True])
```

## Array Operations in NumPy:

### 1. **Scalar Operations**:

- Scalar operations involve performing operations between an array and a single value (scalar).
- **Arithmetic Operations**: - Operations like addition, subtraction, multiplication, and division can be applied between an array and a scalar. - The operation is applied to each element of the array. - Example: Adding 5 to each element of an array.
- **Relational Operations**:
  - Relational comparisons (e.g., `<`, `>`, `<=`, `>=`, `==`, `!=`) can be applied between an array and a scalar.
  - This returns a new array of boolean values indicating whether each element satisfies the condition.

In [36]:
```
#scalar operations
a+10
a*10
```

Out[36]:
```
array([990, 390, 880, 440, 140, 530, 810, 120, 160, 110], dtype=int32)
```

In [37]:
```
#comparison operator
a[a>40]
```

Out[37]:
```
array([99, 88, 44, 53, 81], dtype=int32)
```

### 2. **Vector Operations**:

- Vector operations involve performing operations between two arrays.
- **Arithmetic Operations**:
  - Element-wise operations, such as addition, subtraction, multiplication, and division, can be performed between two arrays of the same shape.

    - The corresponding elements from each array are used in the operation.
- **Relational Operations**:
    - Element-wise comparisons can be performed between two arrays of the same shape using relational operators ( `<` , `>` , `<=` , `>=` , `==` , `!=` ).
    - The result is a new array of boolean values, where each element represents whether the condition holds true for the corresponding elements from both arrays.

In [38]:
```
b
f=np.random.randint(30,80,9).reshape(3,3)
f
```

Out[38]:
```
array([[40, 44, 77],
       [54, 77, 76],
       [60, 60, 79]], dtype=int32)
```

In [39]:
```
b+f
```

Out[39]:
```
array([[ 45,  81,  98],
       [142, 127, 100],
       [ 60,  64, 160]], dtype=int32)
```

In [40]:
```
b>f
```

Out[40]:
```
array([[False, False, False],
       [ True, False, False],
       [False, False,  True]])
```

## Mathematical Functions in NumPy:

1. **Basic Mathematical Functions**:
    - **min**: Returns the minimum value of an array.
    - **max**: Returns the maximum value of an array.
    - **sum**: Computes the sum of all elements in the array.
    - **prod**: Computes the product of all elements in the array.

In [41]:
```
np.array(a)
a
```

Out[41]:
```
array([99, 39, 88, 44, 14, 53, 81, 12, 16, 11], dtype=int32)
```

In [42]:
```
#min
np.min(a)
```

Out[42]: np.int32(11)

In [43]:
```
#max
np.max(a)
```

Out[43]: np.int32(99)

In [44]:
```
#sum
np.sum(a)
```

Out[44]: np.int64(457)

In [45]:
```
#prod
np.prod(a)
```

Out[45]: np.int64(1897658186231808)

2. **Statistical Functions**:
    - **mean**: Returns the average (mean) of the array elements.
    - **median**: Returns the median (middle value) of the array elements.
    - **var**: Computes the variance of the array elements, measuring the spread of data.
    - **std**: Returns the standard deviation, which shows how much variation exists from the mean.

In [46]:
```
#mean
np.mean(a)
```

Out[46]: np.float64(45.7)

In [47]:
```
#median
np.median(a)
```

Out[47]: np.float64(41.5)

In [48]:
```
#var
marks=np.random.randint(0,101,100)
print(marks)
print(np.mean(marks))
print(np.median(marks))
print(np.var(marks))
```

```
[63 39 12 34  6 50 65  5 10 66 38 73 64 64 44 13 44 38 32 22 98 45 25 87
 86 75 76 53 68 19 66 94 90 93  8 24 57 91 48 31 55 37 24 73 25  3 88  2
 46 51  0 67 33 44 19 31 96 74 90 66 54 36 58 57 12 97 60 34 55 62 63 14
 96 82 17 36 21 89 86 83 34 73 14 95 56  8 71 67 49 55 54 36 74 51 93  5
 19 97 95 16]
51.14
53.5
798.4803999999999
```

In [49]:
```
np.std(marks)
```

Out[49]:
```
np.float64(28.257395492153908)
```

In [50]:
```
#10000 **2
```

3. **Trigonometric Functions**:
    - NumPy provides various trigonometric functions, such as:
        - **np.sin**: Computes the sine of each element.
        - **np.cos**: Computes the cosine of each element.
        - **np.tan**: Computes the tangent of each element.
    - Inverse trigonometric and hyperbolic functions are also available.

In [51]:
```
#sin
np.sin(a)
```

Out[51]:
```
array([-0.99920683,  0.96379539,  0.0353983 ,  0.01770193,  0.99060736,
        0.39592515, -0.62988799, -0.53657292, -0.28790332, -0.99999021])
```

In [52]:
```
#cos
np.cos(a)
```

Out[52]:
```
array([ 0.03982088,  0.26664293,  0.99937328,  0.99984331,  0.13673722,
       -0.91828279,  0.77668598,  0.84385396, -0.95765948,  0.0044257 ])
```

In [53]:
```
#tan
np.tan(a)
```

Out[53]:
```
array([-2.50925350e+01,  3.61455441e+00,  3.54205013e-02,  1.77046993e-02,
        7.24460662e+00, -4.31158197e-01, -8.10994416e-01, -6.35859929e-01,
        3.00632242e-01, -2.25950846e+02])
```

4. **Dot Product**:
    - The dot product is a key operation in linear algebra and NumPy.
    - **np.dot**: Computes the dot product of two arrays (vectors or matrices). This operation is widely used in matrix multiplication and machine learning.

In [54]:
```
#dot product
#b*f
np.dot(b,f)
g=np.random.randint(10,20,12).reshape(3,4)
h=np.random.randint(10,20,12).reshape(4,3)
np.dot(g,h)
g=np.random.randint(10,20,12).reshape(3,4)
h=np.random.randint(10,20,8).reshape(4,2)
np.dot(g,h)
```

Out[54]:
```
array([[833, 812],
       [842, 847],
       [858, 819]], dtype=int32)
```

5. **Logarithmic and Exponential Functions**:
    - **np.log**: Computes the natural logarithm (base `e`) of each element in the array.
    - **np.log10**: Computes the logarithm with base 10 for each element.
    - **np.exp**: Calculates the exponential (e^x) for each element in the array.

In [55]:
```
#log
np.log(1)
```

Out[55]:
```
np.float64(0.0)
```

In [56]:
```
#log10
np.log10(10)
```

Out[56]:
```
np.float64(1.0)
```

In [57]:
```
#exp
np.exp(0)
```

Out[57]:
```
np.float64(1.0)
```

**Rounding Functions**:

- **np.round**: Rounds each element in the array to the nearest integer or specified number of decimals.
- **np.floor**: Rounds each element down to the nearest integer (the largest integer less than or equal to the element).
- **np.ceil**: Rounds each element up to the nearest integer (the smallest integer greater than or equal to the element).

In [58]:
```
#round
```

```
In [58]: q=np.random.rand(5)*100
         q
Out[58]: array([54.76047296, 36.91931641, 38.36106761, 76.48771312, 13.02027567])

In [59]: np.round(q,2)
Out[59]: array([54.76, 36.92, 38.36, 76.49, 13.02])

In [60]: #floor
         np.floor(q)
Out[60]: array([54., 36., 38., 76., 13.])

In [61]: #ceil
         np.ceil(q)
Out[61]: array([55., 37., 39., 77., 14.])
```

6. **Rounding Functions**:
    - **np.round**: Rounds each element in the array to the nearest integer or specified number of decimals.
    - **np.floor**: Rounds each element down to the nearest integer (the largest integer less than or equal to the element).
    - **np.ceil**: Rounds each element up to the nearest integer (the smallest integer greater than or equal to the element).

## Indexing and Slicing in NumPy

### 1. **Normal Indexing**:

- **Purpose**: Access individual elements of an array by their index.
- **Example**: `array[index]`
- For multi-dimensional arrays, use a tuple of indices separated by commas, like `array[row, column]`.

```
In [63]: a
Out[63]: array([99, 39, 88, 44, 14, 53, 81, 12, 16, 11], dtype=int32)

In [64]: a[2]
Out[64]: np.int32(88)

In [65]: b
```

Out[65]:
```
array([[ 5, 37, 21],
       [88, 50, 24],
       [ 0,  4, 81]], dtype=int32)
```

In [66]:
```
b[1,1]
```

Out[66]:
```
np.int32(50)
```

In [67]:
```
b[2,1]
```

Out[67]:
```
np.int32(4)
```

### 2. **Fancy Indexing**:

- **Purpose**: Access specific elements using an array of indices.
- **Example**: `array[[index1, index2, ...]]` (for one-dimensional arrays) or `array[[row_indices], [col_indices]]` for multidimensional arrays.

In [69]:
```
a[[0,1,5,2,1,0]]
```

Out[69]:
```
array([99, 39, 53, 88, 39, 99], dtype=int32)
```

In [70]:
```
f=np.array(['shariq','prince','varsha','anshul','gaurav'])
f
```

Out[70]:
```
array(['shariq', 'prince', 'varsha', 'anshul', 'gaurav'], dtype='<U6')
```

In [71]:
```
f[[2,-1,2,2]]
```

Out[71]:
```
array(['varsha', 'gaurav', 'varsha', 'varsha'], dtype='<U6')
```

### 3. **Boolean Indexing**:

- **Purpose**: Select elements based on a condition.
- **Example**: `array[condition]` where the condition is a boolean expression, like `array > value`.

## Summary:

- **Normal Indexing**: Access single elements by their exact position.
- **Fancy Indexing**: Access multiple specific elements using arrays of indices.
- **Boolean Indexing**: Filter and select elements based on a condition.

In [72]:
```
a[a>50]
```

Out[72]:
```
array([99, 88, 53, 81], dtype=int32)
```

In [73]:
```
b[30>b]
```

Out[73]:
```
array([ 5, 21, 24,  0,  4], dtype=int32)
```

## Iterating in NumPy Arrays

Iterating over NumPy arrays is a common operation, allowing you to access and manipulate each element in the array.

In [74]:
```
#1d
for i in a:
    print(i)
```

```
99
39
88
44
14
53
81
12
16
11
```

In [75]:
```
#2d
for i in b:
    for j in i:
        print(j)
```

```
5
37
21
88
50
24
0
4
81
```

In [76]:
```
#3d
for i in c:
    for j in i:
        for k in j:
            print(k)
```

```
6
6
1
6
9
1
2
0
```

## Reshaping in NumPy

1. **`reshape`**:
    - The `reshape` function is used to change the shape of a NumPy array without changing its data.
    - It allows you to specify new dimensions for the array, as long as the total number of elements remains the same.
    - Syntax: `array.reshape(new_shape)`
    - Example: If you have an array of shape (6,) and want to reshape it to (2, 3), you can do so with `array.reshape(2, 3)`.

In [77]:
```
np.random.randint(10,20,12).reshape(4,3)
```

Out[77]:
```
array([[18, 17, 17],
       [17, 13, 14],
       [19, 11, 12],
       [14, 18, 10]], dtype=int32)
```

2. **`transpose`**:
    - The `transpose` function is used to permute the dimensions of an array.
    - It effectively flips the axes of the array, allowing for reorientation of multi-dimensional data.
    - For a 2D array, `transpose` swaps rows and columns.
    - Syntax: `array.transpose()` or `array.T`
    - Example: For a 2D array with shape (2, 3), calling `transpose()` or `array.T` will result in an array of shape (3, 2).

In [78]:
```
b.T
```

Out[78]:
```
array([[ 5, 88,  0],
       [37, 50,  4],
       [21, 24, 81]], dtype=int32)
```

In [81]:
```
b.transpose() #same
```

Out[81]:
```
array([[ 5, 88,  0],
       [37, 50,  4],
       [21, 24, 81]], dtype=int32)
```

3. **`ravel`**:
   - The `ravel` function is used to flatten a multi-dimensional array into a one-dimensional array.
   - It returns a flattened view of the original array whenever possible, without copying the data.
   - Syntax: `array.ravel()`
   - Example: If you have a 2D array of shape (2, 3), calling `ravel()` will produce a 1D array with shape (6).

In [82]:
```
c.ravel()
```

Out[82]:
```
array([6, 6, 1, 6, 9, 1, 2, 0], dtype=int32)
```

In [83]:
```
x=np.linspace(-10,10,100)
x
```

Out[83]:
```
array([-10.        ,  -9.7979798 ,  -9.5959596 ,  -9.39393939,
        -9.19191919,  -8.98989899,  -8.78787879,  -8.58585859,
        -8.38383838,  -8.18181818,  -7.97979798,  -7.77777778,
        -7.57575758,  -7.37373737,  -7.17171717,  -6.96969697,
        -6.76767677,  -6.56565657,  -6.36363636,  -6.16161616,
        -5.95959596,  -5.75757576,  -5.55555556,  -5.35353535,
        -5.15151515,  -4.94949495,  -4.74747475,  -4.54545455,
        -4.34343434,  -4.14141414,  -3.93939394,  -3.73737374,
        -3.53535354,  -3.33333333,  -3.13131313,  -2.92929293,
        -2.72727273,  -2.52525253,  -2.32323232,  -2.12121212,
        -1.91919192,  -1.71717172,  -1.51515152,  -1.31313131,
        -1.11111111,  -0.90909091,  -0.70707071,  -0.50505051,
        -0.3030303 ,  -0.1010101 ,   0.1010101 ,   0.3030303 ,
         0.50505051,   0.70707071,   0.90909091,   1.11111111,
         1.31313131,   1.51515152,   1.71717172,   1.91919192,
         2.12121212,   2.32323232,   2.52525253,   2.72727273,
         2.92929293,   3.13131313,   3.33333333,   3.53535354,
         3.73737374,   3.93939394,   4.14141414,   4.34343434,
         4.54545455,   4.74747475,   4.94949495,   5.15151515,
         5.35353535,   5.55555556,   5.75757576,   5.95959596,
         6.16161616,   6.36363636,   6.56565657,   6.76767677,
         6.96969697,   7.17171717,   7.37373737,   7.57575758,
         7.77777778,   7.97979798,   8.18181818,   8.38383838,
         8.58585859,   8.78787879,   8.98989899,   9.19191919,
         9.39393939,   9.5959596 ,   9.7979798 ,  10.        ])
```

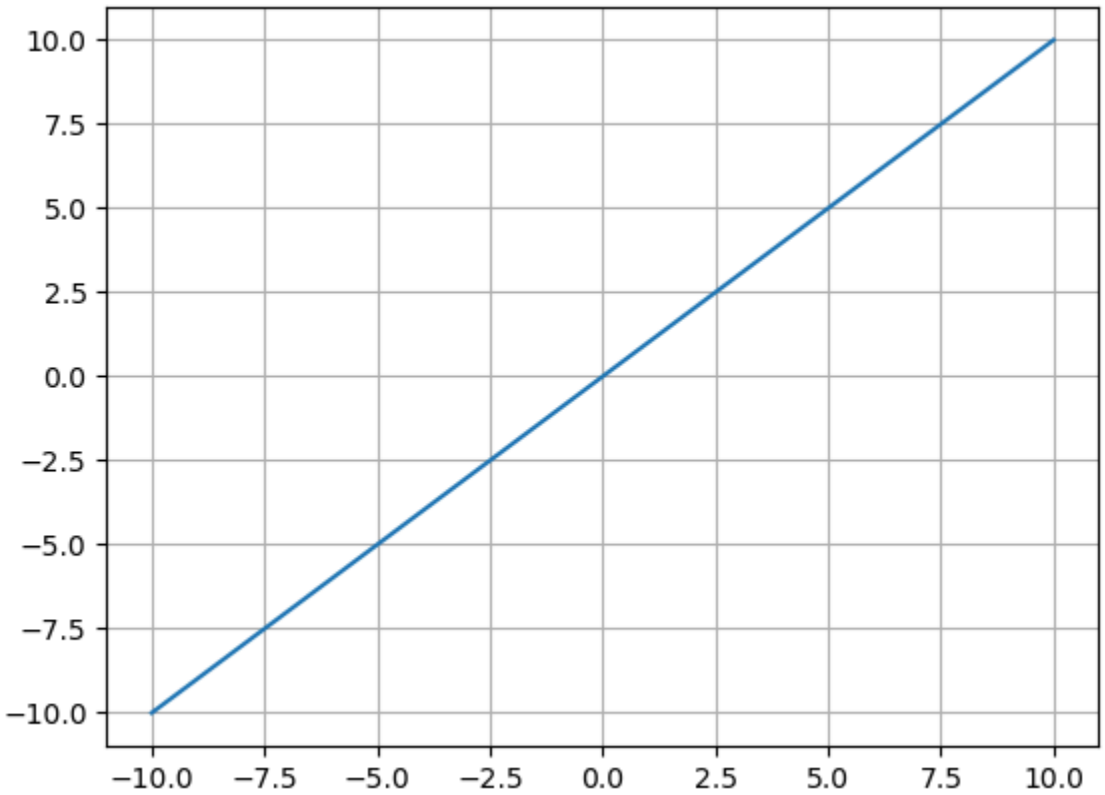
# Plotting graphs

## Common graphs and their functions

### 1.) Linear function

A linear equation is an equation where the highest power of each variable is one. You'll find this concept applied in areas such as algebraic expressions, word problems, and graphical analysis. In a linear equation, variables like x or y are never raised to powers higher than one, and the solution always produces a straight line when graphed.

In [87]:
```
import matplotlib.pyplot as plt
y=x
plt.plot(x,y)
plt.grid()
plt.show()
```



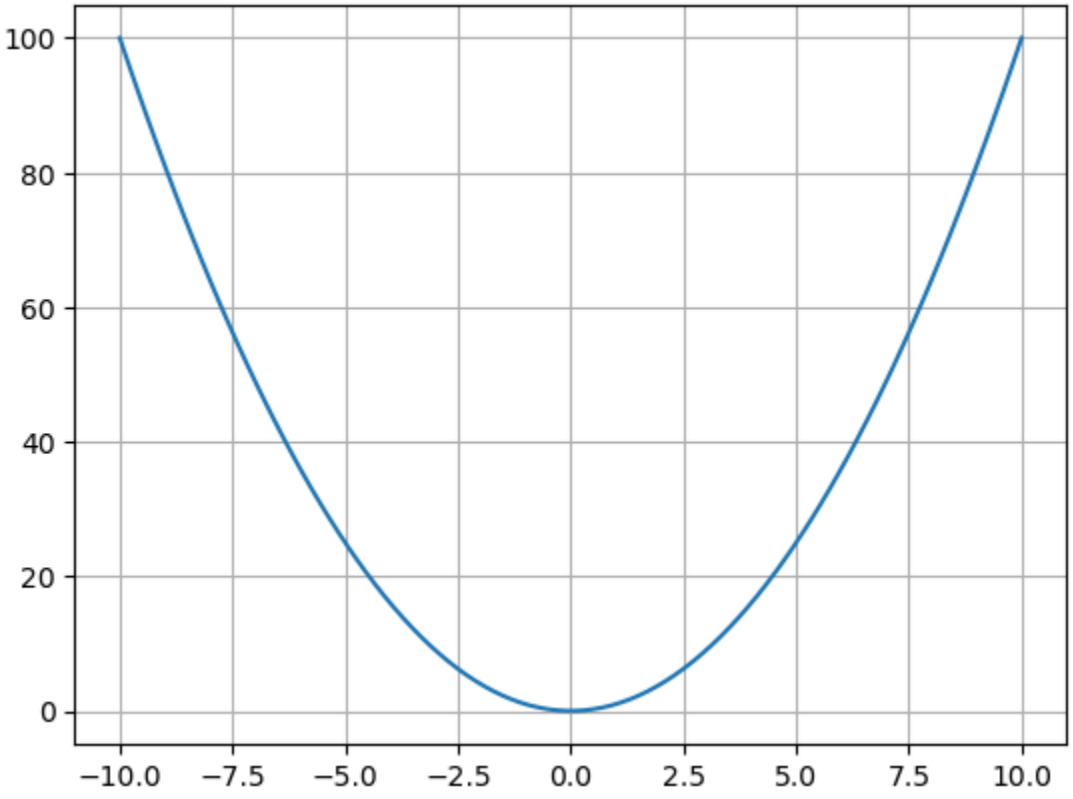

2. **Quadratic Function: ( y = x^2 )**
    - **Description:** Represents a parabolic curve opening upwards.
    - **Usage:** Useful for modeling scenarios involving

acceleration, area, or optimization problems. It shows how the output increases at an increasing rate as ( x ) moves away from zero.

In [88]:
```
(-2)**2
2**2
```

Out[88]: 4

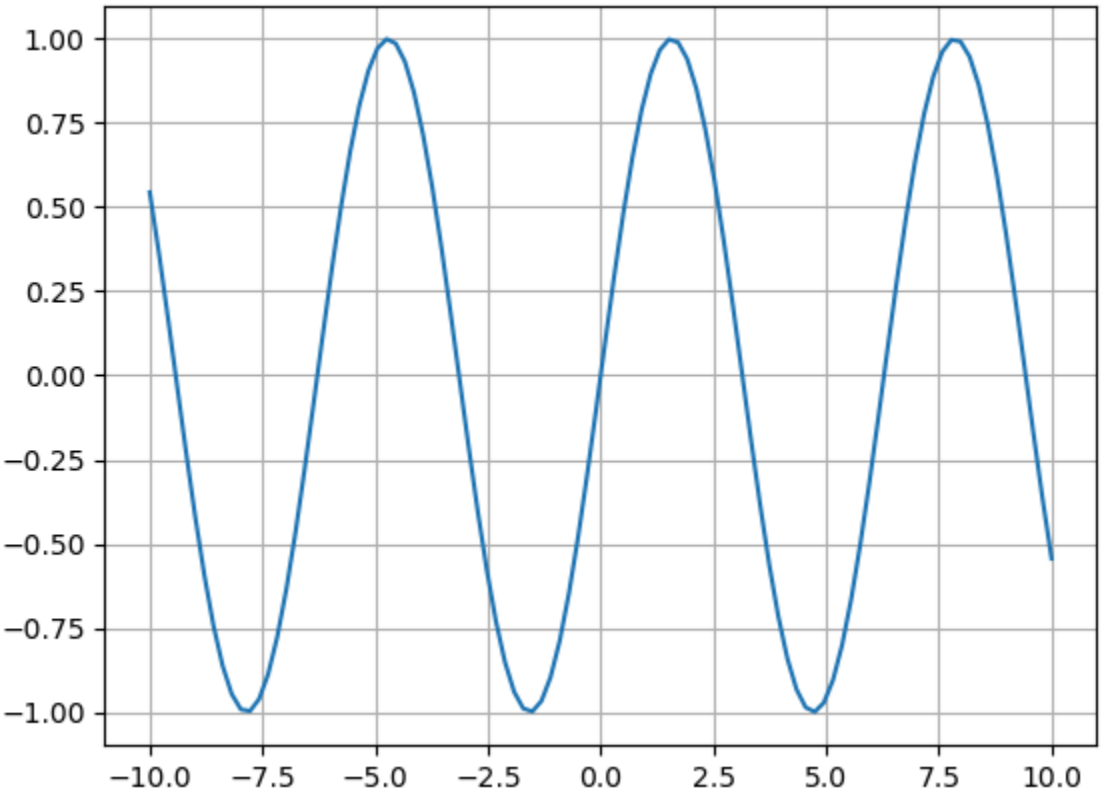
In [89]:
```
y=x**2
plt.plot(x,y)
plt.grid()
plt.show()
```




3. **Sine Function: ( y = \sin(x) )**
    - **Description:** A periodic wave that oscillates between -1 and 1.
    - **Usage:** Commonly used in physics and engineering to model waveforms, oscillations, and cyclic behaviors, such as sound waves and light waves.

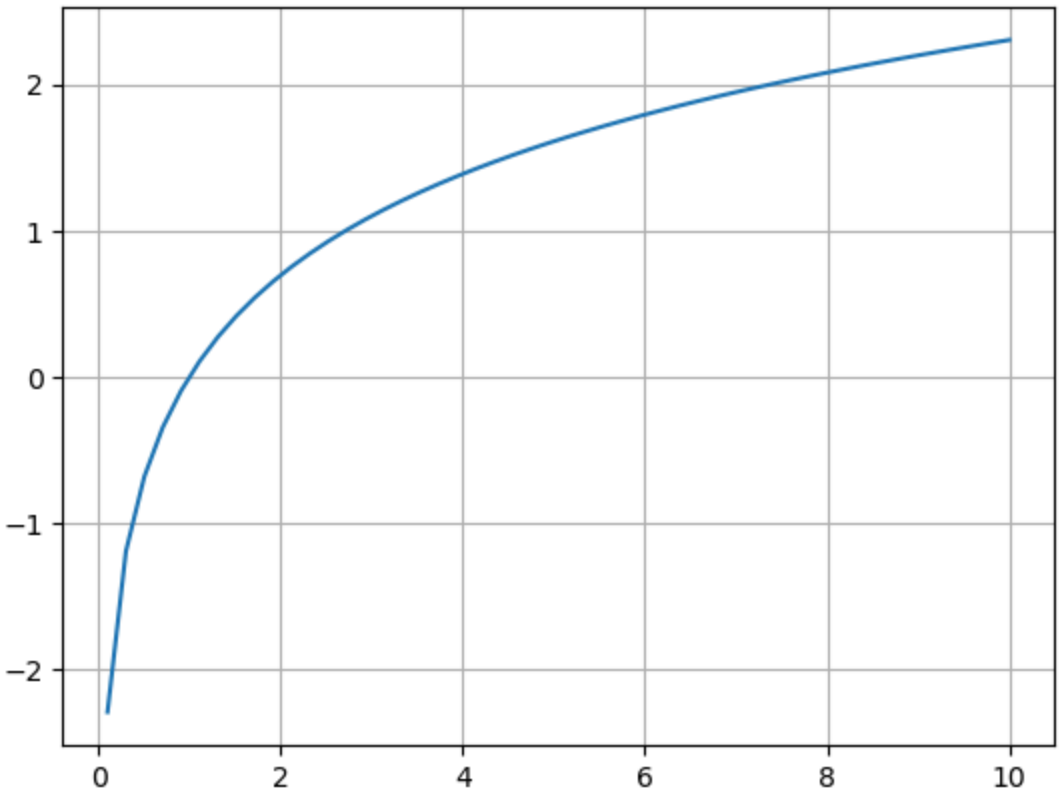
In [90]:
```
y=np.sin(x)
plt.plot(x,y)
```

```
plt.grid()
plt.show()
```




4. **Logarithmic Function: ( y = x \log(x) )**
    - **Description:** A combination of linear and logarithmic growth. For ( x > 0 ), it increases slowly initially but grows faster as ( x ) increases.
    - **Usage:** Frequently found in computational complexity, information theory, and scenarios where growth slows down, like population growth in limited environments.

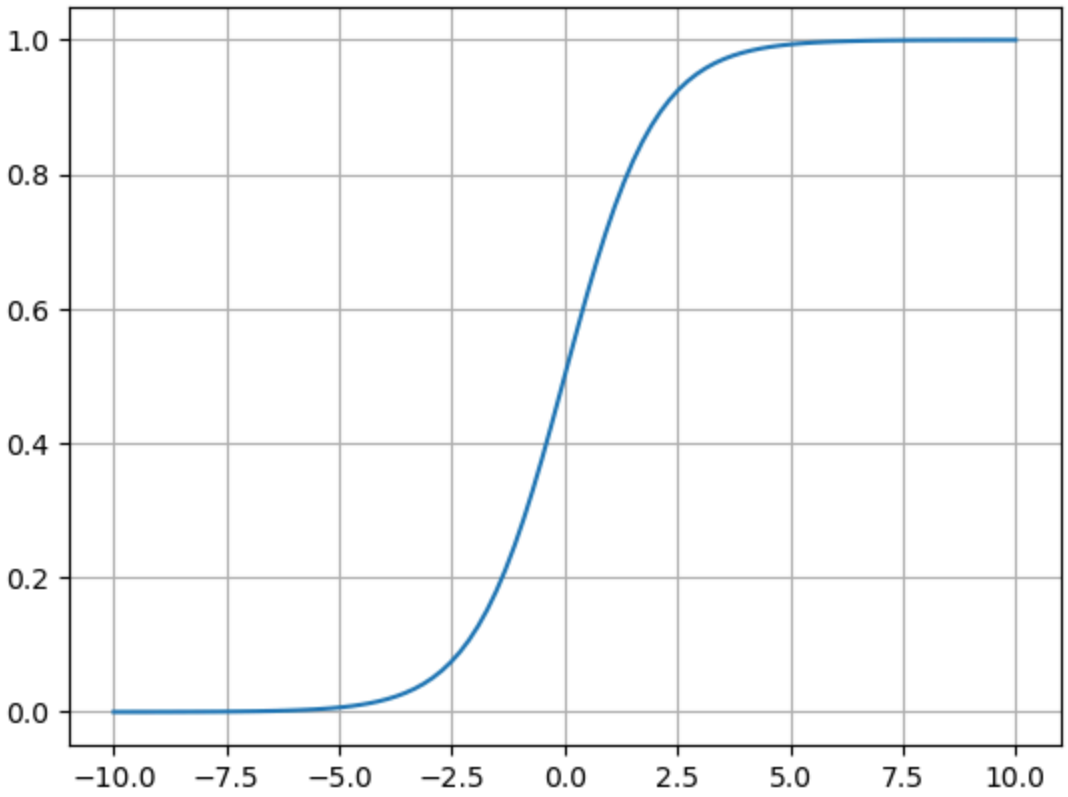
In [95]:
```
import warnings
warnings.filterwarnings('ignore')
y=np.log(x)
plt.plot(x,y)
plt.grid()
plt.show()
```




5. **Sigmoid Function:**
    - **Description:** A smooth, S-shaped curve defined by the formula ( y = \frac{1}{1 + e^{-x}} ).
    - **Usage:** Widely used in statistics and machine learning, particularly in logistic regression and neural networks, to model probabilities that are constrained between 0 and 1.

In [97]:
```
# weight = [40,50,60,70,80,90,100]
# obese = [0,0,0,0,1,1,1]
```

In [98]:
```
y = (1/(1+np.exp((-x))))
plt.plot(x,y)
plt.grid()
plt.show()
```




# Working with missing values (NaN, which stands for "Not a Number") in NumPy involves specific methods and functions. Here are some important points to help you understand:

## 1. Representation of Missing Values

- In NumPy, missing values are represented using `np.nan`.
- You can use it in any numeric array.

In [99]:
```
None
np.nan
z=np.array([10,20,30,np.nan,32,np.nan,31])
z
```

Out[99]:
```
array([10., 20., 30., nan, 32., nan, 31.])
```

## 2. Identifying Missing Values

- You can check if an array contains any NaN values using the `np.isnan()` function.

```
import numpy as np
```

```
arr = np.array([1, 2, np.nan, 4])
missing = np.isnan(arr)  # This will give a boolean array
print(missing)  # Output: [False False  True False]
```

In [100…
```
np.isnan(z)
```

Out[100…
```
array([False, False, False,  True, False,  True, False])
```

## 3. Dropping Missing Values

- If you want to remove NaN values, you can use boolean indexing with `np.isnan()`:

```
arr_no_nan = arr[~np.isnan(arr)]  # Remove NaN values
print(arr_no_nan)  # Output: [1. 2. 4.]
```

In [101…
```
z[~np.isnan(z)]
```

Out[101…
```
array([10., 20., 30., 32., 31.])
```

## 4. Replacing Missing Values

- You can replace missing values with a specific value using `np.nan_to_num()`:

```
arr_filled = np.nan_to_num(arr, nan=0)  # Replace NaN with 0
print(arr_filled)  # Output: [1. 2. 0. 4.]
```

In [102…
```
np.nan_to_num(z,nan=0)
```

Out[102…
```
array([10., 20., 30.,  0., 32.,  0., 31.])
```

## 5. Calculating Statistics with Missing Values

- When calculating statistics, you need to be aware that NaN values can affect your calculations.
- You can use functions like `np.nansum()`, `np.nanmean()`, `np.nanstd()`, etc., which ignore NaN values.

```
mean_value = np.nanmean(arr)  # Calculates mean while
ignoring NaN values
print(mean_value)  # Output: 2.333...
```

In [104…
```
# np.mean(z)
```

```
print(np.nanmean(z))
print(np.nanmedian(z))
print(np.nanstd(z))
```

```
24.6
30.0
8.475848040166836
```

## Summary

- Use `np.nan` to represent NaN.
- For **Identifying**, use `np.isnan()`, for **Dropping**, use boolean indexing, and for **Replacing**, use `np.nan_to_num()`.
- Use functions that ignore NaN values when calculating **Statistics**.

## Stacking in NumPy

1. `vstack` **(Vertical Stacking)**:
    - Combines arrays by stacking them vertically (row-wise), creating a new array where the arrays are placed on top of each other.
    - Syntax: `np.vstack((array1, array2, ...))`

In [105…
```
b
x=np.random.randint(10,20,9).reshape(3,3)
x
```

Out[105…
```
array([[18, 19, 13],
       [12, 10, 11],
       [11, 15, 19]], dtype=int32)
```

In [110…
```
k = np.vstack((b,x))
```

2. `hstack` **(Horizontal Stacking)**:
    - Combines arrays by stacking them horizontally (column-wise), creating a new array where arrays are placed side by side.
    - Syntax: `np.hstack((array1, array2, ...))`

In [112…
```
l = np.hstack((b,x))
```

## Splitting in NumPy

Splitting in NumPy refers to dividing an array into multiple sub-arrays along a specified axis.

1. **`hsplit` (Horizontal Split)**:
    - **Purpose**: Splits an array into multiple sub-arrays along the horizontal axis (column-wise).
    - **Syntax**: `np.hsplit(array, indices_or_sections)`
    - **Usage**: This is used to split an array into multiple parts horizontally.

In [114…
```
np.hsplit(l,2)
```

Out[114…
```
[array([[ 5, 37, 21],
        [88, 50, 24],
        [ 0,  4, 81]], dtype=int32),
 array([[18, 19, 13],
        [12, 10, 11],
        [11, 15, 19]], dtype=int32)]
```

2. **`vsplit` (Vertical Split)**:
    - **Purpose**: Splits an array into multiple sub-arrays along the vertical axis (row-wise).
    - **Syntax**: `np.vsplit(array, indices_or_sections)`
    - **Usage**: This is used to split an array into multiple parts vertically.

## Summary:

- **`hsplit`**: Splits an array horizontally (along columns).
- **`vsplit`**: Splits an array vertically (along rows).

In [115…
```
np.vsplit(k,2)
```

Out[115…
```
[array([[ 5, 37, 21],
        [88, 50, 24],
        [ 0,  4, 81]], dtype=int32),
 array([[18, 19, 13],
        [12, 10, 11],
        [11, 15, 19]], dtype=int32)]
```

## Broadcasting in NumPy

**Broadcasting** is a powerful mechanism that allows NumPy to perform element-wise operations on arrays of different shapes without needing to create copies or expand the smaller array to the size of the larger one.

## Key Concepts:

- **Different shapes**: When performing operations (like addition, multiplication) between arrays, they don't need to have the exact same shape.
- **Automatic expansion**: NumPy automatically expands the smaller array along the dimensions where necessary to match the shape of the larger array.

In [116…
```
# same shape
b+x
b
```

Out[116…
```
array([[ 5, 37, 21],
       [88, 50, 24],
       [ 0,  4, 81]], dtype=int32)
```

In [118…
```
# diffrent shape
b + np.arange(0,3)
```

Out[118…
```
array([[ 5, 38, 23],
       [88, 51, 26],
       [ 0,  5, 83]])
```

## Rules of Broadcasting

Broadcasting in NumPy lets you perform operations on arrays with different shapes by "stretching" the smaller array without actually copying data.

Here are the basic rules:

1. **Match from the end**: NumPy compares the shapes of the two arrays starting from the last dimension (right to left).
2. **Same size or 1**: Two dimensions are compatible if:
    - They are the same size, or
    - One of them is 1 (it will be "stretched" to match the other).
3. **Missing dimensions**: If one array has fewer dimensions, NumPy will automatically add extra dimensions of size 1 at the front so it can match the larger array.

## Example:

- **Array shapes: (4, 3)** and **(3)**

- NumPy stretches the second array `(3)` to `(1, 3)` and then to `(4, 3)` for the operation.

## Summary:

- **Dimensions must match** or **one must be 1**.
- **Smaller arrays** can be stretched to match the larger array's shape.

This allows you to work with arrays of different sizes easily without reshaping them manually.

In [ ]: