SUnit1:

Data science: definition, Datafication, Exploratory Data Analysis, The Data science process,

A data scientist role in this process.

NumPy Basics: The NumPy ndarray: A Multidimensional Array Object, Creating ndarrays,Data Types for ndarrays, Operations between Arrays and Scalars, Basic Indexing and Slicing, Boolean Indexing, Fancy Indexing, Data Processing Using Arrays, Expressing Conditional Logic as Array Operations, Methods for Boolean Arrays , Sorting , Unique.

Data science:

Definition:

Data science is the study of data to extract meaningful insights for business. It is a multidisciplinary approach that combines principles and practices from the fields of mathematics, statistics, artificial intelligence, and computer engineering to analyze large amounts of data. This analysis helps data scientists to ask and answer questions like what happened, why it happened, what will happen, and what can be done with the results.

The word first appeared in the ’60s as an alternative name for statistics. In the late ’90s, computer science professionals formalized the term.

Data science is used to study data in four main ways:

1. Descriptive analysis: Descriptive analysis examines data to gain insights into what happened or what is happening in the data environment. It is characterized by data visualizations such as pie charts, bar charts, line graphs, tables, or generated narratives

2. Diagnostic analysis: Diagnostic analysis is a deep-dive or detailed data examination to understand why something happened. It is characterized by techniques such as drill-down, data discovery, data mining, and correlations.

3. Predictive analysis: Predictive analysis uses historical data to make accurate forecasts about data patterns that may occur in the future.

4. Prescriptive analysis: It not only predicts what is likely to happen but also suggests an optimum response to that outcome.

Datafication:

Process of “taking all aspects of life and turning them into data”. We are being datafied, or rather our actions are, and when we “like” someone or something online, we are intending to be datafied, or at least we should expect to be. But when we merely browse the Web, we are unintentionally, or at least passively, being datafied through cookies that we might or might not be aware of. And when we walk around in a store, or even on the street, we are being datafied in a completely unintentional way, via sensors, cameras, or Google glasses.

Exploratory Data Analysis

Exploratory Data Analysis (EDA) is a process of describing the data by means of statistical and visualization techniques in order to bring important aspects of that data into focus for further analysis.

Types of EDA Techniques

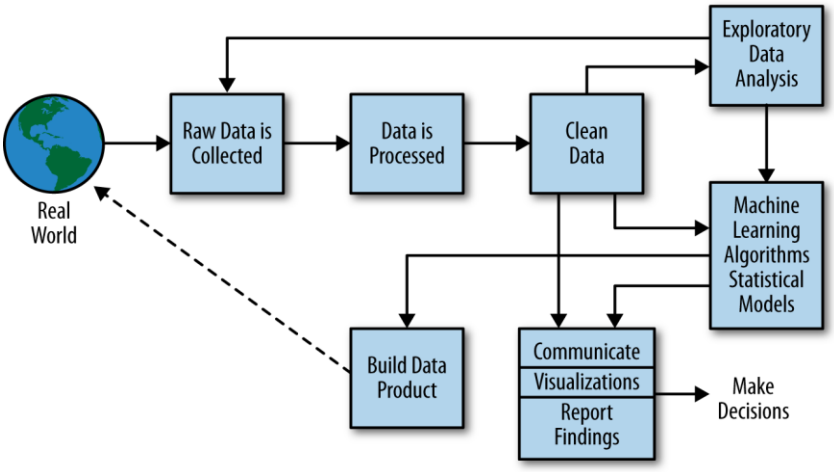
- **Univariate Analysis:** Univariate analysis examines individual variables to understand their distributions and summary statistics. This includes calculating measures such as mean, median, mode, and standard deviation, and visualizing the data using histograms, bar charts, box plots, and violin plots.
- **Bivariate Analysis:** Bivariate analysis explores the relationship between two variables. It uncovers patterns through techniques like scatter plots, pair plots, and heatmaps. This helps to identify potential associations or dependencies between variables.
- **Multivariate Analysis:** Multivariate analysis involves examining more than two variables simultaneously to understand their relationships and combined effects. Techniques such as contour plots, and principal component analysis (PCA) are commonly used in multivariate EDA.
- **Visualization Techniques:** EDA relies heavily on visualization methods to depict data distributions, trends, and associations. Various charts and graphs, such as bar charts, line charts, scatter plots, and heatmaps, are used to make data easier to understand and interpret.
- **Outlier Detection:** EDA involves identifying outliers within the data—anomalies that deviate significantly from the rest of the data. Tools such as box plots, z-score analysis, and scatter plots help in detecting and analyzing outliers.
- **Statistical Tests:** EDA often includes performing statistical tests to validate hypotheses or discern significant differences between groups. Tests such as t-tests, chi-square tests, and ANOVA add depth to the analysis process by providing a statistical basis for the observed patterns.

Experimental example:

https://www.analyticsvidhya.com/blog/2021/08/exploratory-data-analysis-and-visualization-techniques-in-data-science/

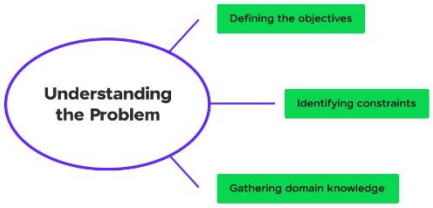
The Data science process

The data science process is a systematic approach to analyzing and interpreting data to extract meaningful insights and solve real-world problems.



Data science process involves the following steps:

1.Understanding the problem



- **Defining the objectives: What is the end goal? Are you trying to predict future sales, classify customer reviews, or detect fraud?**
- **Identifying constraints: What are the limitations? These could be data availability, time constraints, or computational resources.**
- **Gathering domain knowledge: Understanding the context of the problem helps in making informed decisions throughout the process.**

2. Data Collection:

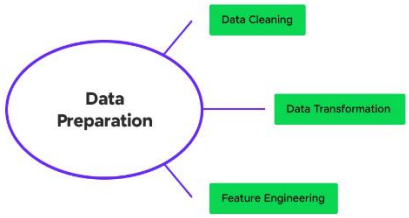
After clear understanding of the problem, the next step in the data science process is to collect the relevant data. This can be done through various means such as:

- **Web Scraping:** This method is useful for gathering data from public websites. It requires knowledge of web technologies and legal considerations to avoid violating terms of

service. Python libraries like Beautiful Soup and Scrapy are commonly used for web scraping tasks.

- **APIs:** Many online platforms, such as social media sites and financial data providers, offer APIs to access their data. Using APIs ensures that you get structured data, often in real-time, which is crucial for time-sensitive analyses.
- **Databases:** Internal databases are gold mines of historical data. SQL is the go-to language for querying relational databases, while NoSQL databases like MongoDB are used for unstructured data.
- **Surveys and Sensors:** Surveys are effective for collecting user opinions and feedback, while sensors are invaluable in IoT applications for gathering real-time data from physical devices.

3. Data preparation:



Raw data in the data science process is often messy and needs to be cleaned and prepared before analysis. Data preparation involves:

- **Data cleaning:** Handling data such as missing values, removing duplicates, and correcting errors.
- **Data transformation:** Converting data into a suitable format, such as normalizing or standardizing values.
- **Feature engineering:** Creating new features from existing data that can improve the performance of machine learning models.

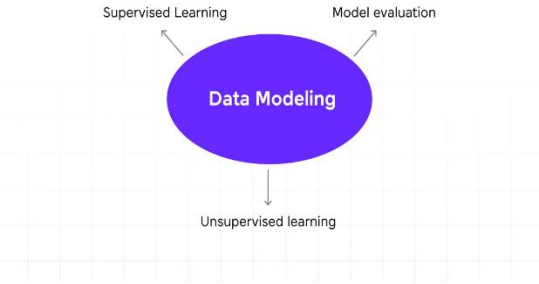
4. Exploratory Data Analysis (EDA):

EDA is a crucial step in the data science process where you explore the data to uncover patterns, anomalies, and relationships. This involves:

- **Descriptive statistics:** Calculating mean, median, standard deviation, etc.
- **Visualization:** Creating plots such as histograms, scatter plots, and box plots to visualize data distributions and relationships.
- **Correlation analysis:** Identifying relationships between different variables.

Exploratory Data Analysis (EDA) is not just a preliminary step but a vital part of the data science process. It helps in forming hypotheses about the data and guides the selection of appropriate models. Descriptive statistics provide a summary of the central tendency, dispersion, and shape of the dataset's distribution. This helps in getting a quick overview of the data and identifying any anomalies.

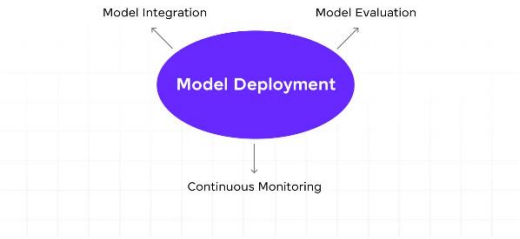
5. Data Modeling



After understanding the data, the next step in the data science process is to build predictive models. This involves selecting and applying appropriate algorithms. Common techniques include:

- **Supervised learning:** Algorithms like linear regression, decision trees, and neural networks are used when the outcome is known.
- **Unsupervised learning:** Techniques like clustering and dimensionality reduction are used when the outcome is unknown.
- **Model evaluation:** Using metrics such as accuracy, precision, recall, and F1 score to evaluate model performance.

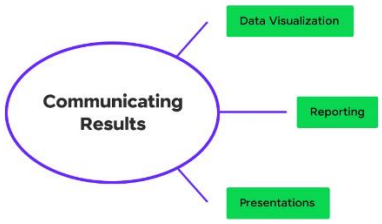
6. Model Deployment



Once you have a satisfactory model, the next step in our data science process is to deploy it in a real-world environment. This involves:

- Model integration: Integrating the model into an application or system.
- API development: Creating APIs to allow other systems to access the model.
- Continuous monitoring: Regularly checking the model's performance to ensure it continues to perform well over time.

7. Communicating Results



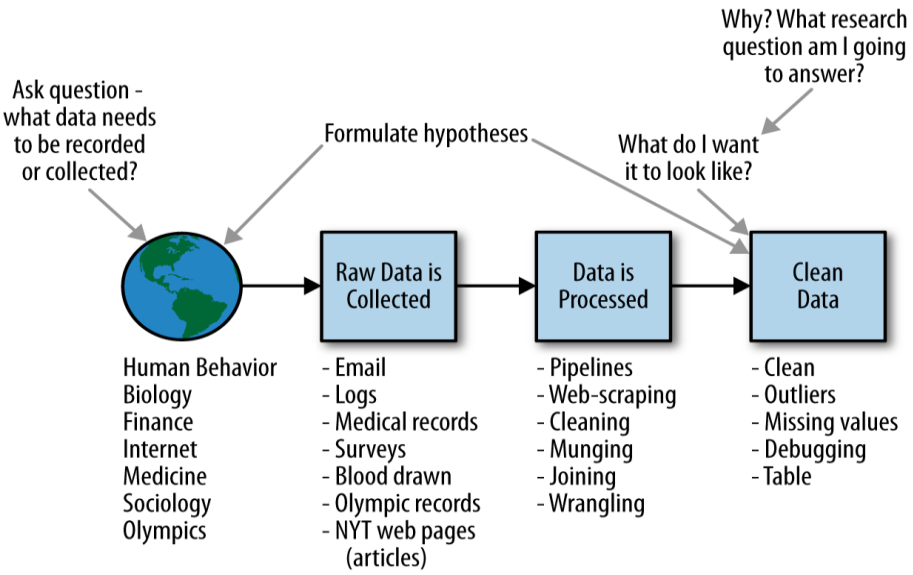
The final step is to communicate the results to stakeholders. This is crucial for making informed business decisions. Effective communication involves:

- **Data visualization:** Creating clear and intuitive visualizations to convey findings.
- **Reporting:** Writing comprehensive reports that explain the methodology, findings, and recommendations.
- **Presentations:** Delivering presentations to stakeholders to discuss the insights and potential actions.

A data scientist role in this process

- Ask a question.
- Do background research.
- Construct a hypothesis.
- Test your hypothesis by doing an experiment.
- Analyze your data and draw a conclusion.
- Communicate your results.

The data scientist is involved in every part of this process



NumPy Basics:

The NumPy ndarray: A Multidimensional Array Object:

Creating ndarrays:

The array function in NumPy is used to create an array.

array(object, dtype=None, copy=True, order='K', subok=False, ndmin=0)

1.object

- This is the data or object to be converted into a NumPy array. It can be a list, tuple, ndarray, or any other array-like object.

Ex : np.array([1, 2, 3]) # Creates an array from a list

2. dtype (optional)

- Specifies the desired data type of the array elements. If not provided, the data type is inferred from the input data.
- Common Values: int, float, complex, bool, etc.

Ex: np.array([1, 2, 3], dtype=float) # Creates an array of floats

3. copy (optional, default=True)

- If True, a new array is always created. If False, a new array is created only if needed .

  Ex : a = np.array([1, 2, 3])

b = np.array(a, copy=False) # `b` may share memory with `a` if no changes are needed

4. order (optional, default='K')

- **Description**: Specifies the memory layout of the array.
  - 'C': Row-major (C-style)
  - 'F': Column-major (Fortran-style)
  - 'A': Preserve the input's layout
  - 'K': Preserve the input's layout unless a copy is made

Ex: np.array([[1, 2], [3, 4]], order='F') # Creates a Fortran-style array

5. subok (optional, default=False)

- If True, subclasses of ndarray (like masked arrays) are passed through without being cast to a base ndarray. If False, the result is always a base ndarray.

Ex: class MyArray(np.ndarray):

  pass

a = np.array([1, 2, 3]).view(MyArray)

b = np.array(a, subok=True) # Retains subclass

The .view() method in NumPy creates a new view of the array with the same data but potentially a different interpretation of that data. When you call .view() with a class as an argument (e.g., MyArray), it creates a new object of the specified class while retaining the data of the original array.

6. ndmin (optional, default=0)

- **Description**: Specifies the minimum number of dimensions the resulting array should have. Extra dimensions of size 1 are prepended if necessary.

Ex: np.array([1, 2, 3], ndmin=2) # Creates a 2D array: [[1, 2, 3]]

A list can be used to create an array object.

In [13]: data1 = [6, 7.5, 8, 0, 1]

In [14]: arr1 = np.array(data1)

In [15]: arr1

Out[15]: array([6. , 7.5, 8. , 0. , 1.])

Nested sequences, like a list of equal-length lists, will be converted into a multidimensional array:

In [16]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]

In [17]: arr2 = np.array(data2)

In [18]: arr2

Out[18]:

array([[1, 2, 3, 4],

[5, 6, 7, 8]])

In [19]: arr2.ndim # gives number of dimensions

Out[19]: 2

In [20]: arr2.shape # gives the size of array with rows and columns

Out[20]: (2, 4)

The data type is stored in a special **dtype** object.

For example,in the above two examples we have:

In [21]: arr1.dtype

Out[21]: dtype('float64')

zeros and **ones** create arrays of 0's or 1's, respectively, with a given length or shape. **empty** creates an array without initializing its values to any particular value.

➔ zeros(shape, dtype=float, order='C')

Return a new array of given shape and type, filled with zeros.

Parameters

---

shape : int or tuple of ints

Shape of the new array, e.g., ``(2, 3)`` or ``2``.

dtype : data-type, optional

The desired data-type for the array, e.g., `numpy.int8`. Default is `numpy.float64`.

order : {'C', 'F'}, optional, default: 'C'

Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

Returns

----

out : ndarray

Array of zeros with the given shape, dtype, and order.

```
In [46]: al=np.zeros((2,3),float)
         al

Out[46]: array([[0., 0., 0.],
                [0., 0., 0.]])

In [47]: al=np.zeros(3,float)
         al

Out[47]: array([0., 0., 0.])
```

➔ ones(shape, dtype=**None**, order=**'C'**)

Return a new array of given shape and type, filled with ones.

```
In [50]: ao=np.ones(5, dtype=int)
         ao

Out[50]: array([1, 1, 1, 1, 1])
```

➔ empty(shape, dtype=float, order='C')

Return a new array of given shape and type, without initializing entries.

```
In [51]: ae=np.empty([2, 2])
         ae

Out[51]: array([[1.33733641e+160, 8.81904788e+199],
                [9.26733849e+242, 7.29489530e+175]])
```

arange is an array-valued version of the built-in Python range function.

Syntax: arange([start,] stop [, step,], dtype=None)

Return evenly spaced values within a given interval. Values are generated within the half-open interval ``[start, stop)`` (in other words, the interval including `start` but excluding `stop`).

For integer arguments the function is equivalent to the Python built-in `range` function, but returns an ndarray rather than a list.

Parameters

---

start : number, optional

Start of interval. The interval includes this value. The default start value is 0.

stop : number

End of interval. The interval does not include this value.

step : number, optional

Spacing between values. For any output `out`, this is the distance between two adjacent values, ``out[i+1] - out[i]``. The default step size is 1. If `step` is specified as a position argument, `start` must also be given.

dtype : dtype

The type of the output array. If `dtype` is not given, infer the datatype from the other input arguments.

When using a non-integer step, such as 0.1, the results will often not be consistent. It is better to use `numpy.linspace` for these cases.

```
In [17]: b=np.arange(5,51,5)

In [18]: b
```

Function name	Description
array	Convert input data (list, tuple, array, or other sequence type) to a ndarray either by inferring a dtype or explicitly specifying a dtype. Copies the input data by default.
asarray	Convert input to ndarray, but do not copy if the input is already an ndarray
arange	Like the built-in range but returns an ndarray instead of a list.
ones, ones_like	Produce an array of all 1's with the given shape and dtype. ones_like takes another array and produces a ones array of the same shape and dtype.
zeros, zeros_like	Like ones and ones_like but producing arrays of 0's instead
empty, empty_like	Create new arrays by allocating new memory, but do not populate with any values like ones and zeros
eye, identity	Return a 2-D array with ones on the diagonal and zeros elsewhere. np**.** identity **(**n**,** dtype=**None)** Return the identity array. np.eye **(**N**,** M=**None,** k=**0,** dtype=**'Float',** order=**'C')** **k ->positive -upper diagonal** ->negative -lower diagonal
full	np**.** full **(**shape**,** fill _value**,** dtype=**None,** order=**'C')** Return a new array of given shape and type, filled with `fill_value`.

Data Types for ndarrays:

The *data type* or dtype is a special object containing the information the ndarray needs to interpret a chunk of memory as a particular type of data.

Dtypes map directly onto an underlying machine representation, which makes it easy to read and write binary streams of data to disk and also to connect to code written in a low-level language like C or Fortran.

The numerical dtypes are named the same way: a type name, like float or int, followed by a number indicating the number of bits per element. A standard double-precision floating point value (what's used under the hood in Python's float object) takes up 8 bytes or 64 bits. Thus, this type is known in NumPy as float64.

```
In [62]: arr1 = np.array([1, 2, 3], dtype=np.float64)
         arr1.dtype

Out[62]: dtype('float64')
```

int8, uint8	i1, u1	Signed and unsigned 8-bit (1 byte) integer types
int16, uint16	i2, u2	Signed and unsigned 16-bit integer types
int32, uint32	i4, u4	Signed and unsigned 32-bit integer types
int64, uint64	i8, u8	Signed and unsigned 32-bit integer types
float16	f2	Half-precision floating point
float32	f4 or f	Standard single-precision floating point. Compatible with C float
float64, float128	f8 or d	Standard double-precision floating point. Compatible with C double and Python float object
float128	f16 or g	Extended-precision floating point
complex64,complex128, complex256	c8, c16, c32	Complex numbers represented by two 32, 64, or 128 floats, respectively
bool object string_	? O S	Boolean type storing True and False values Python object typeFixed-length string type (1 byte per character).

Explicit conversion from one dtype to another using ndarray's **astype** method

```
In [64]: arr = np.array([1, 2, 3, 4, 5])
         print(arr.dtype)
         float_arr = arr.astype(np.float64)
         float_arr.dtype

         int32

Out[64]: dtype('float64')
```

Casting some floating-point numbers to be of integer dtype, the decimal part will be truncated.

```
In [65]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
         print(arr)

         arr.astype(np.int32)
         arr
```

```
[ 3.7 -1.2 -2.6  0.5 12.9 10.1]
```

```
Out[65]: array([ 3.7, -1.2, -2.6,  0.5, 12.9, 10.1])
```

An array of strings representing numbers, we can use astype to convert them to numeric form

```
In [66]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)
         numeric_strings.astype(float)
```

```
Out[66]: array([ 1.25, -9.6 , 42.  ])
```

Note :Calling astype *always* creates a new array (a copy of the data), even if the new dtype is the same as the old dtype.

Operations between Arrays and Scalars:

Arrays are important as can be used for batch operations on data without writing any for loops. This is called ***vectorization.***

Any arithmetic operations between equal-size arrays applied with the operation elementwise.

```
In [68]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])
         arr
```

```
Out[68]: array([[1., 2., 3.],
                [4., 5., 6.]])
```

```
In [69]: arr * arr
```

```
Out[69]: array([[ 1.,  4.,  9.],
                [16., 25., 36.]])
```

Arithmetic operations with scalars are as applying with the value to each element

```
In [70]: 1 / arr
```

```
Out[70]: array([[1.        , 0.5       , 0.33333333],
                [0.25      , 0.2       , 0.16666667]])
```

*Operations between differently sized arrays is called ***broadcasting.***

Basic Indexing and Slicing:

NumPy indexing and slicing are similar to python Lists.

```
In [72]: arr = np.arange(10)
         arr
Out[72]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [73]: arr[5:8]
Out[73]: array([5, 6, 7])

In [74]: arr[5:8]=12
         arr
Out[74]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

If a scalar value is assigned to a slice, as in arr[5:8] = 12, the value is propagated (or *broadcasted* henceforth) to the entire selection. An important first distinction from lists is that array slices are *views* on the original array. This means that the data is not copied, and any modifications to the view will be reflected in the source array:

```
In [75]: arr_slice = arr[5:8]

In [76]: arr_slice[1] = 12345
         arr
Out[76]: array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,
                    9])

In [77]: arr_slice[:] = 64
         arr
Out[77]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

*If we want a copy of a slice of an ndarray instead of a view, we need to explicitly copy the array; for example arr[5:8].copy().

In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays:

Ex: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

arr2d[2]

array([7, 8, 9])

To access individual elements in 2d array:

```
In [80]: arr2d[0][2]
Out[80]: 3
```

We can pass a comma-separated list of indices to select individual elements.

```
In [81]: arr2d[0, 2]
Out[81]: 3
```

In multidimensional arrays, if you omit later indices, the returned object will be a lower dimensional ndarray consisting of all the data along the higher dimensions. So in the 2 × 2 × 3 array arr3d

```
In [82]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
         arr3d

Out[82]: array([[[ 1,  2,  3],
                 [ 4,  5,  6]],

                [[ 7,  8,  9],
                 [10, 11, 12]]])
```

arr3d[0] is a 2 × 3 array:

```
In [83]: arr3d[0]

Out[83]: array([[1, 2, 3],
                [4, 5, 6]])
```

Both scalar values and arrays can be assigned to arr3d[0]:

```
In [84]: old_values = arr3d[0].copy()

In [86]: arr3d[0] = 42
         arr3d

Out[86]: array([[[42, 42, 42],
                 [42, 42, 42]],

                [[ 7,  8,  9],
                 [10, 11, 12]]])

In [88]: arr3d[0] = old_values
         arr3d

Out[88]: array([[[ 1,  2,  3],
                 [ 4,  5,  6]],

                [[ 7,  8,  9],
                 [10, 11, 12]]])
```

Similarly, arr3d[1, 0] gives you all of the values whose indices start with (1, 0), forming a 1-dimensional array:

```
In [89]: arr3d[1, 0]

Out[89]: array([7, 8, 9])
```

Boolean Indexing

In NumPy, Boolean indexing allows us to filter elements from an array based on a specific condition.

Performing data analysis using Python, a common operation is filtering the data. It allows us to extract relevant patterns and insights from the data. One way to filter data is through Boolean vectors. The process of doing this is commonly known as Boolean indexing.

Boolean indexing in NumPy uses a Boolean array to select elements from an array that meet a certain condition. The Boolean array is a binary mask that indicates whether each element in the array should be selected or not.

```
In [17]: #Boolean indexing
         data=np.array([1,2,3,4])
         bdata=data>=3
         filldata=data[bdata]
         print(filldata)

         [3 4]
```

Example: # create an array of numbers
array1 = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# create a boolean mask
boolean_mask = array1 % 2 != 0

# boolean indexing to filter the odd numbers
result = array1[boolean_mask]

print(result)

Output: [1 3 5 7 9]

Modify Elements Using Boolean Indexing
In NumPy, we can use boolean indexing to modify elements of the array. For example,
import numpy as np

Example:
# Create an array of numbers
numbers = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

Make a copy of the array
numbers_copy = numbers.copy()

change all even numbers to 0 in the copy
numbers_copy[numbers % 2 == 0] = 0

# Print the modified copy
print(numbers_copy)

Output: [1 0 3 0 5 0 7 0 9 0]

2D Boolean Indexing in NumPy

Boolean indexing can also be applied to multi-dimensional arrays in NumPy.

create a 2D array

array1 = np.array([[1, 7, 9],
[14, 19, 21],
[25, 29, 35]])

```
# create a boolean mask based on the condition
# that elements are greater than 9
boolean_mask = array1 > 9

# select only the elements that satisfy the condition
result = array1[boolean_mask]

print(result)
```

output:

[14 19 21 25 29 35]

Fancy Indexing:

In NumPy, fancy indexing allows us to use an array of indices to access multiple array elements at once. Fancy indexing can perform more advanced and efficient array operations, including conditional filtering, sorting, and so on.

Example:

```
# Create a NumPy array
array1 = np.array([1, 2, 3, 4, 5, 6, 7, 8])

# select elements at index 1, 2, 5, 7
select_elements = array1[[1, 2, 5, 7]]

print(select_elements)
```

Output: [2 3 6 8]

Fancy Indexing to Assign New Values to Specific Elements:

We can also assign new values to specific elements of a NumPy array using fancy indexing.

For example,

```
array1 = np.array([3, 2, 6, 1, 8, 5, 7, 4])

# create a list of indices to assign new values
indices = [1, 3, 6]

# create a new array of values to assign
new_values = [10, 20, 30]

# Use fancy indexing to assign new values to specific elements
array1[indices] = new_values

print(array1)

# Output: [ 3 10  6 20  8  5 30  4]
```

Reshaping of Arrays:

Reshaping numpy array simply means changing the shape of the given array, shape basically tells the number of elements and dimension of array, by reshaping an array we can add or remove dimensions or change number of elements in each dimension.
In order to reshape a numpy array we use reshape method with the given array.

Example:

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr = arr.reshape(4, 3)

print(newarr)
```

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

The opposite operation of reshape from one-dimensional to a higher dimension is known as ***flattening*** **or** ***raveling*****:**

```
arr = np.arange(15).reshape((5, 3))

array([[ 0, 1, 2],
[ 3, 4, 5],
[ 6, 7, 8],
[ 9, 10, 11],
[12, 13, 14]])
```

Data Processing Using Arrays:

Data Processing is a way of organizing the data to get the required results.

Expressing Conditional Logic as Array Operations:

In NumPy, we can replace explicit loops and if conditions with efficient array operations using boolean indexing, numpy.where(), and logical functions.

Using numpy.where()

The numpy.where() function is useful for element-wise conditional selection.

Syntax: where(condition, [x, y])
Condition True->x
False->y
Return elements chosen from `x` or `y` depending on `condition`.

condition : array_like, bool

Where True, yield `x`, otherwise yield `y`.
x, y : array_like
Values from which to choose. `x`, `y` and `condition` need to be broadcastable to some shape.

Example:
1.Replacing negative values in an array with 0:

```
arr = np.array([-3, -2, 5, -1, 7])
result = np.where(arr < 0, 0, arr)
print(result)

# Output: [0 0 5 0 7]
```

2. Replace Values Greater Than 10

```
arr = np.array([5, 12, 8, 20, 3])
result = np.where(arr > 10, 10, arr)
print(result)

 # Output: [5 10 8 10 3]
```

3. Replace Even Numbers with Square, Odd with Cube

```
arr = np.array([1, 2, 3, 4, 5])
result = np.where(arr % 2 == 0, arr**2, arr**3)
print(result)

# Output: [1 4 27 16 125]
```

4. Replace all positive values with 2 and all negative values with -2.

If all the arrays are 1-D, `where` is equivalent to:

```
[xv if c else yv

 for c, xv, yv in zip(condition, x, y)]
```

Example:
Let us consider 3 arrays:

```
xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
cond = np.array([True, False, True, True, False])
```

Take a value from xarr whenever the corresponding value in cond is True otherwise take the value from yarr.

```
result = [(x if c else y)
          for x, y, c in zip(xarr, yarr, cond)]
```

The numpy module supports the ***logical_and*** operator. It is used to relate between two variables. If two variables are 0 then output is 0, if two variables are 1 then output is 1 and if one variable is 0 and another is 1 then output is 0.

Syntax:
numpy.logical_and(var1,var2)
Where, var1 and var2 are a single variable or a list/array.
Return type: *Boolean value (True or False)*

```
list1 = [True, False, True, False]

# list 2 represents an array with boolean values
list2 = [True, True, False, True]

# logical operations between boolean values
print('Operation between two lists =  ', np.logical_and(list1, list2))
```

The NumPy module supports the ***logical_or*** operator. It is also used to relate between two variables. If two variables are 0 then output is 0, if two variables are 1 then output is 1 and if one variable is 0 and another is 1 then output is 1.

Syntax:
numpy.logical_or(var1,var2)
Where, var1 and var2 are a single variable or a list/array.
Return type: *Boolean value (True or False)*

Example:

```
print('logical_or operation =  ',  np.logical_or(True, False))

a = 2
b = 6
print('logical or Operation between two variables =  ', np.logical_or(a, b))
a = 0
b = 0
print('logical or Operation between two variables =  ', np.logical_or(a, b))


# list 1 represents an array with integer values
list1 = [1, 2, 3, 4, 5, 0]

# list 2 represents an array with integer values
list2 = [0, 1, 2, 3, 4, 0]

# logical operations between integer values
print('Operation between two lists =  ',  np.logical_or(list1, list2))
```

Methods for Boolean Arrays:

Boolean values are coerced to 1 (True) and 0 (False) in the above methods. Thus, sum is often used as a means of counting True values in a boolean array:

```
arr = np.random.randn(100)
 (arr > 0).sum() # Number of positive values
```

Output:

```
44
```

There are two additional methods, **any** and **all**, useful especially for boolean arrays. any tests whether one or more values in an array is True, while all checks if every value is True:
Example:

```
bools = np.array([False, False, True, False])
bools.any()
```
output:
True

```
bools.all()
```

Output: False
These methods also work with non-boolean arrays, where non-zero elements evaluate to True.

Sort:

NumPy arrays can be sorted in-place using the sort method:
1. numpy.sort() – Sort an Array
This function returns a sorted copy of the array without modifying the original.

```
arr = np.array([5, 2, 8, 1, 3])
sorted_arr = np.sort(arr)
print(sorted_arr) #
```
Output: [1 2 3 5 8]

2. numpy.argsort() – Get Sorted Indices
If you want the indices of the sorted elements:
Example:
```
arr = np.array([5, 2, 8, 1, 3])
sorted_indices = np.argsort(arr)
print(sorted_indices)
# Output: [3 1 4 0 2] # Sorting the array using the indices
print(arr[sorted_indices])
```
Output: [1 2 3 5 8]

3. Sorting Multi-Dimensional Arrays:
Sorting Along Rows (axis=1)

```
arr = np.array([[3, 1, 4], [9, 2, 8]])
sorted_arr = np.sort(arr, axis=1)
print(sorted_arr)
# Output: # [[1 3 4] # [2 8 9]]
```

Sorting Along Columns (axis=0)
```
sorted_arr = np.sort(arr, axis=0)
print(sorted_arr)
```
#Output: # [[3 1 4] # [9 2 8]]

4. Sorting in Descending Order
NumPy doesn't have a direct descending sort, but you can reverse a sorted array:

```
arr = np.array([5, 2, 8, 1, 3])
desc_sorted = np.sort(arr)[::-1]
print(desc_sorted)
# Output: [8 5 3 2 1]
```

5. numpy.lexsort() – Sorting by Multiple Keys
Sort using multiple columns (useful for structured data).

Example:

```
names = np.array(['Alice', 'Bob', 'Charlie', 'David'])
scores = np.array([85, 92, 85, 75]) # Primary sort key (ascending)
ages = np.array([23, 20, 22, 21]) # Secondary sort key (ascending)
sorted_indices = np.lexsort((ages, scores)) # Sort by scores first, then by ages
print(names[sorted_indices])

# Output: ['David' 'Charlie' 'Alice' 'Bob']
```

6. numpy.sort() vs numpy.ndarray.sort()

- np.sort(arr): Returns a sorted copy.
- np.arr.sort(): Sorts **in-place** (modifies original array).

Example:

```
arr = np.array([5, 2, 8, 1, 3])
arr.sort() print(arr)
 # Output: [1 2 3 5 8] (Original array modified)
```

Unique:

np.unique, which returns the sorted unique values in an array.

Example:

```
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
np.unique(names)

output:
array(['Bob', 'Joe', 'Will'],dtype='|S4')
```

np.in1d, tests membership of the values in one array in another, returning a boolean array:

example:

```
values = np.array([6, 0, 0, 3, 2, 5, 6])
np.in1d(values, [2, 3, 6])

Output: array([ True, False, False, True, True, False, True], dtype=bool)
```