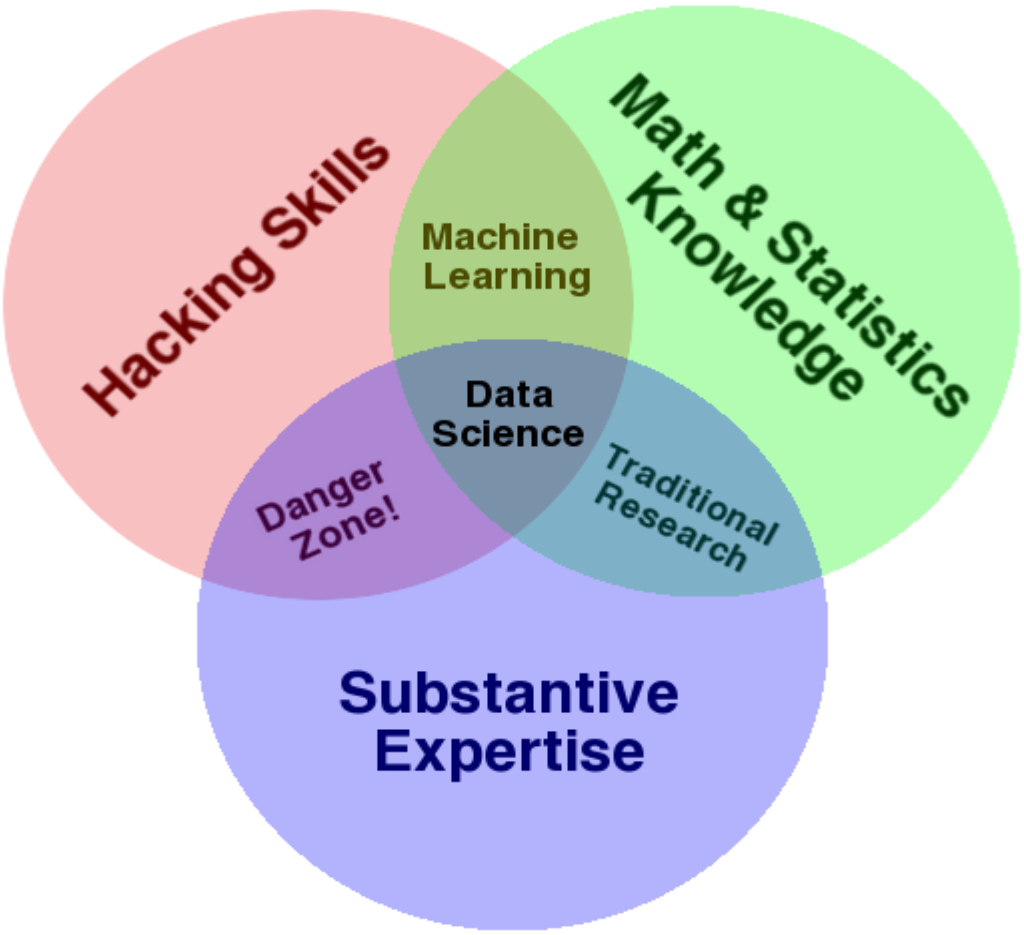
Unit -4:

1a)Data science: It is the science of analysing the raw data using statistical and machine learning skills with the purpose of drawing conclusions about the information.



Explanation of Drew Conway's Venn Diagram

In Drew Conway's Venn Diagram of Data Science, the primary colours of data are

- Hacking skills,
- Math and stats knowledge, and
- Substantive expertise

But the question is why has he highlighted these three? So let's understand the term why!!

- It is known to everyone that data is the key part of data science. And data is a commodity traded electronically; so, in order to be in

this market, one needs to speak **hacker**. So what does this line means? Being able to manage text files at the command-line, learning vectorized operations, thinking algorithmically; are the **hacking skills** that make for a successful **data hacker**.

- Once you have collected and cleaned the data, the next step is to actually obtain insight from it. In order to do this, you need to use appropriate **mathematical and statistical methods**, that demand at least a baseline familiarity with these tools. This is not to say that a PhD in statistics is required to be a skilled data scientist, but it does need understanding what an ordinary least squares regression is and how to explain it.
- The third important part is **Substantive expertise.** According to Drew Conway, "**data plus math and statistics only gets you machine learning"**, which is excellent if that is what you are interested in, but not if you are doing data science. Science is about experimentation and building knowledge, which demands some motivating questions about the world and hypotheses that can be brought to data and tested with statistical methods.
- On the other hand, "substantive expertise + knowledge in mathematics and statistics are where maximum traditional researcher falls". Doctoral level researchers use most of their time getting expertise in these areas, but very little time acquiring technology. Part of this is the culture of academia, which does not compensate researchers for knowing technology.

1b) Big Data and Datafication are two important concepts in the field of Data Science:

Big Data: Big Data refers to large and complex datasets that are beyond the capabilities of traditional data processing methods. It is characterized by the three V's: Volume, Velocity, and Variety.

- Volume: Big Data involves massive amounts of data, often in terabytes, petabytes, or even exabytes. This data is generated from various sources such as social media, sensors, transaction records, and more.
- Velocity: Big Data is generated at high speed and requires real-time or near-real-time processing. The data flows in rapidly from sources like social media feeds, IoT devices, and online transactions.
- Variety: Big Data comes in diverse formats and structures, including structured data (databases, spreadsheets), unstructured data (text, images, videos), and semi-structured data (log files, XML). It also includes data from different sources and platforms.

Big Data presents both challenges and opportunities for Data Science. Data scientists employ advanced techniques like distributed computing, cloud computing, and machine learning algorithms to extract insights, uncover patterns, and make predictions from large and complex datasets.

Datafication: Datafication refers to the process of converting various aspects of the world into quantifiable data that can be collected, analyzed, and used for decision-making and insights.

- Data Collection: Datafication involves collecting data from diverse sources, including structured databases, unstructured text, social media, sensors, and more.
- Data Analysis: Datafication enables data scientists to perform analysis and extract insights. Techniques like statistical analysis, data mining, machine learning, and visualization are used to uncover patterns, correlations, and trends within the data.
- Data-driven Decision-making: Datafication empowers organizations to make informed decisions by leveraging quantitative evidence and insights derived from data analysis. It supports evidence-based decision-making across various domains.

Datafication plays a crucial role in Data Science as it provides the raw material for analysis, modeling, and decision-making. It allows data scientists to transform real-world phenomena into quantifiable data, uncover insights, and drive data-driven strategies and actions.

2a) **Exploratory Data Analysis (EDA)** is an approach that is used to analyze the data and discover trends, patterns, or check assumptions in data with the help of statistical summaries and graphical representations.

Types of EDA

Depending on the number of columns we are analyzing we can divide EDA into two types.

1. **Univariate Analysis** – In univariate analysis, we analyze or deal with only one variable at a time. The analysis of univariate data is thus the simplest form of analysis since the information deals with only one quantity that changes. It does not deal with causes or relationships and the main purpose of the analysis is to describe the data and find patterns that exist within it.
2. **Bi-Variate analysis** – This type of data involves two different variables. The analysis of this type of data deals with causes and

relationships and the analysis is done to find out the relationship between the two variables.

3. **Multivariate Analysis** – When the data involves three or more variables, it is categorized under multivariate.

Depending on the type of analysis we can also subcategorize EDA into two parts.

1. Non-graphical Analysis – In non-graphical analysis, we analyze data using statistical tools like mean median or mode or skewness
2. Graphical Analysis – In graphical analysis, we use visualizations charts to visualize trends and patterns in the data

EDA using python:

```
import pandas as pd

import numpy as np

# read datasdet using pandas

df = pd.read_csv('employees.csv')

df.head()
```

Output:

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
0	Douglas	Male	8/6/1993	12:42 PM	97308	6.945	True	Marketing
1	Thomas	Male	3/31/1996	6:53 AM	61933	4.170	True	NaN
2	Maria	Female	4/23/1993	11:17 AM	130590	11.858	False	Finance
3	Jerry	Male	3/4/2005	1:00 PM	138705	9.340	True	Finance
4	Larry	Male	1/24/1998	4:47 PM	101004	1.389	True	Client Services

First five rows of the dataframe

```
df.shape
```

Output:

```
(1000, 8)
```

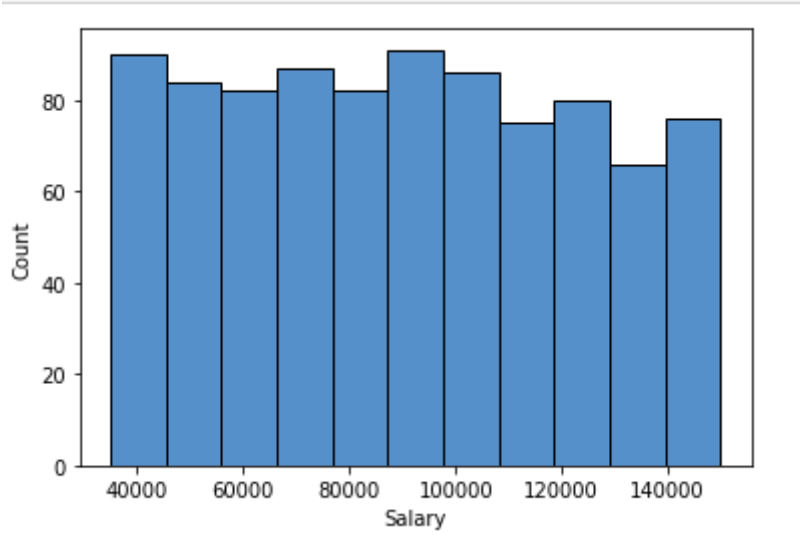
This means that this dataset has 1000 rows and 8 columns.

```
df.tail()#gives bottom 5 rows of the data set
```

```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt


sns.histplot(x='Salary', data=df, )
plt.show()
```

Output:



2b) A Data Scientist's Role in data science Process

This model so far seems to suggest this will all magically happen without human intervention. By "human" here, we mean "data scientist." Someone has to make the decisions about what data to collect, and why. That person needs to be formulating questions and hypotheses and making a plan for how the problem will be attacked. And that someone is the data scientist .

Roles & Responsibilities of a Data Scientist

- **Management:** The Data Scientist plays an insignificant managerial role where he supports the construction of the base of futuristic and technical abilities within the Data and Analytics field in order to assist various planned and continuing data analytics projects.
- **Analytics:** The Data Scientist represents a scientific role where he plans, implements, and assesses high-level statistical models and strategies for application in the business's most complex issues. The

Data Scientist develops econometric and statistical models for various problems including projections, classification, clustering, pattern analysis, sampling, simulations, and so forth.

- **Strategy/Design:** The Data Scientist performs a vital role in the advancement of innovative strategies to understand the business's consumer trends and management as well as ways to solve difficult business problems, for instance, the optimization of product fulfillment and entire profit.
- **Collaboration:** The role of the Data Scientist is not a solitary role and in this position, he collaborates with superior data scientists to communicate obstacles and findings to relevant stakeholders in an effort to enhance drive business performance and decision-making.
- **Knowledge:** The Data Scientist also takes leadership to explore different technologies and tools with the vision of creating innovative data-driven insights for the business at the most agile pace feasible. In this situation, the Data Scientist also uses initiative in assessing and utilizing new and enhanced data science methods for the business, which he delivers to senior management of approval.
- **Other Duties:** A Data Scientist also performs related tasks and tasks as assigned by the Senior Data Scientist, Head of Data Science, Chief Data Officer, or the Employer.

3a) **NumPy** is the fundamental package for scientific computing in Python. Numpy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences. Numpy is not another programming language but a Python extension module. It provides fast and efficient operations on arrays of homogeneous data.

Some important points about Numpy arrays:

- We can create an N-dimensional array in python using Numpy.array().
- The array is by default Homogeneous, which means data inside an array must be of the same Datatype. (Note you can also create a structured array in python).
- Element-wise operation is possible.
- Numpy array has various functions, methods, and variables, to ease our task of matrix computation.

- Elements of an array are stored contiguously in memory. For example, all rows of a two-dimensioned array must have the same number of columns. Or a three dimensioned array must have the same number of rows and columns on each card.
- Multi-dimensional Numpy Array

- Python3

```
import numpy as np

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

Output:

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

A **Python [list](#)** is a collection that is ordered and changeable. In Python, lists are written with square brackets.

```
Var = ["Geeks", "for", "Geeks"]
print(Var)
```

Output:

```
["Geeks", "for", "Geeks"]
```

Advantages of using Numpy Arrays Over Python Lists:

- Consumes less memory.
- Fast as compared to the python List.
- Convenient to use.
- convenient to perform complex computations and manipulations on arrays.

3b) To install the NumPy package, you can use the following syntax in your Python environment:

pip install numpy

This command will install the NumPy package using pip, the package installer for Python.

To create a NumPy array in Python, you can use the `numpy.array()` function. Here's an example Python program that demonstrates how to create a NumPy array:

#1d array

```
import numpy as np

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

print(arr)

o/p:[1 2 3 4 5]

#2d array

import numpy as np

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

print(arr)

o/p: [[1 2 3]

 [4 5 6]]

#3d array

import numpy as np

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

print(arr)

o/p:[[[1 2 3]

  [4 5 6]]



 [[1 2 3]

  [4 5 6]]]
```

4a) NumPy has some extra data types, and refer to data types with one character, like `i` for integers, `u` for unsigned integers etc.

Below is a list of all data types in NumPy and the characters used to represent them.

- `i` - integer
- `b` - boolean
- `u` - unsigned integer
- `f` - float
- `c` - complex float
- `m` - timedelta
- `M` - datetime
- `O` - object
- `S` - string
- `U` - unicode string
- `V` - fixed chunk of memory for other type (void)

```
import numpy as np

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

print(arr.dtype)
```

o/p:int64

```
import numpy as np

arr = np.array(['apple', 'banana', 'cherry'])

print(arr.dtype)
```

o/p:U6

```
import numpy as np

arr = np.array([1, 2, 3, 4], dtype='S')

print(arr)

print(arr.dtype)
```

o/p:[b'1' b'2' b'3' b'4']

|S1

4b) an example program that demonstrates how to create ndarrays using NumPy:

```
import numpy as np

# Create a 1D array

arr_1d = np.array([1, 2, 3, 4, 5])

print("1D Array:")

print(arr_1d)

# Create a 2D array

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

print("\n2D Array:")
```

```
print(arr_2d)

# Create a 3D array

arr_3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])

print("\n3D Array:")

print(arr_3d)
```

In this program, we import the `numpy` module as `np` and use it to create ndarrays.

After creating each array, we print it to the console using the `print()` function.

Outputs of the above programs are:

```
1D Array:

[1 2 3 4 5]

2D Array:

[[1 2 3]

 [4 5 6]]

3D Array:

[[[1 2]

  [3 4]]

 [[5 6]

  [7 8]]]
```

The program demonstrates the creation of 1D, 2D, and 3D ndarrays using NumPy. You can modify the program by providing your own values or dimensions to create ndarrays of different shapes and sizes.

5a) Here's an example program that demonstrates scalar operations on NumPy arrays:

```
import numpy as np

# Create a NumPy array

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

# Scalar addition

addition_result = arr + 2

print("Scalar Addition:")

print(addition_result)

# Scalar subtraction
```

```
subtraction_result = arr - 2
print("\nScalar Subtraction:")
print(subtraction_result)
# Scalar multiplication
multiplication_result = arr * 2
print("\nScalar Multiplication:")
print(multiplication_result)
# Scalar division
division_result = arr / 2
print("\nScalar Division:")
print(division_result)
# Scalar exponentiation
exponentiation_result = arr ** 2
print("\nScalar Exponentiation:")
print(exponentiation_result)
```

When you run this program, the output will be:

```
Scalar Addition:
[3 4 5 6 7]
Scalar Subtraction:
[-1  0  1  2  3]
Scalar Multiplication:
[ 2  4  6  8 10]
Scalar Division:
[0.5 1.  1.5 2.  2.5]
Scalar Exponentiation:
[ 1  4  9 16 25]
```

The program demonstrates how scalar operations can be applied to NumPy arrays, resulting in element-wise operations on the array elements.

5b) Here's an example program that creates a NumPy array and demonstrates the **`resize()`** and **`reshape()`** functions:

```
import numpy as np
```

```
# Create a NumPy array

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

# Print the original array

print("Original Array:")

print(arr)

# Resize the array using resize()

resized_array = np.resize(arr, (3, 4))

# Print the resized array

print("\nResized Array:")

print(resized_array)

# Reshape the array using reshape()

reshaped_array = arr.reshape((2, 3))

# Print the reshaped array

print("\nReshaped Array:")

print(reshaped_array)
```

the use of `resize()` and `reshape()` functions:

1. `resize()`: We use the `resize()` function to resize the original array to the specified shape `(3, 4)`. The `resize()` function can adjust the size of the array by repeating or truncating elements as necessary to fit the desired shape. The resulting array is stored in the `resized_array` variable.
2. `reshape()`: We use the `reshape()` function to reshape the original array to the specified shape `(2, 3)`. The `reshape()` function returns a new array with the specified shape without modifying the original array. The resulting array is stored in the `reshaped_array` variable.

o/p:Original Array:

[1 2 3 4 5 6]

Resized Array:

[[1 2 3 4]

[5 6 1 2]

[3 4 5 6]]

Reshaped Array:

[[1 2 3]

[4 5 6]]

6) Array indexing is the same as accessing an array element.

You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

Ex:

import numpy as np

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

print(arr[0])

o/p:1

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.

Think of 2-D arrays like a table with rows and columns, where the dimension represents the row and the index represents the column.

Example

Access the element on the first row, second column:

```
import numpy as np

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

print('2nd element on 1st row: ', arr[0, 1])
```

o/p:

2nd element on 1st dim: 2

Access 3-D Arrays

To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.

Example

Access the third element of the second array of the first array:

```
import numpy as np

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

```
print(arr[0, 1, 2])
```

o/p:6

Negative Indexing

Use negative indexing to access an array from the end.

Example

Print the last element from the 2nd dim:

```
import numpy as np

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

print('Last element from 2nd dim: ', arr[1, -1])
```

o/p:

```
Last element from 2nd dim:  10
```

Slicing in python means taking elements from one given index to another given index.

We pass slice instead of index like this: `[start:end]`.

We can also define the step, like this: `[start:end:step]`.

If we don't pass start its considered 0

If we don't pass end its considered length of array in that dimension

If we don't pass step its considered 1

Example

Slice elements from index 1 to index 5 from the following array:

```
import numpy as np

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

print(arr[1:5])
```

o/p:[2 3 4 5]

Slice elements from index 4 to the end of the array:

```
import numpy as np

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

print(arr[4:])
```

o/p;[5 6 7]

Slice elements from the beginning to index 4 (not included):

```
import numpy as np

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

print(arr[:4])
```

o/p:[1 2 3 4]

Negative Slicing

Use the minus operator to refer to an index from the end:

Example

Slice from the index 3 from the end to index 1 from the end:

```
import numpy as np

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

print(arr[-3:-1])
```

o/p:[5 6]

Use the `step` value to determine the step of the slicing:

Example

Return every other element from index 1 to index 5:

```
import numpy as np

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

print(arr[1:5:2])
```

o/p:[2 4]

Slicing 2-D Arrays

Example

From the second element, slice elements from index 1 to index 4 (not included):

```
import numpy as np

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

print(arr[1, 1:4])
```

o/p:[7 8 9]

From both elements, return index 2:

```
import numpy as np

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

print(arr[0:2, 2])
```

o/p;[3 8]

From both elements, slice index 1 to index 4 (not included), this will return a 2-D array:

```
import numpy as np

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

print(arr[0:2, 1:4])
```

o/p: [[2 3 4]

[7 8 9]]

7) NumPy is a powerful Python library that provides support for efficient numerical operations on large arrays and matrices. It offers various data processing tasks that can be performed on arrays using its extensive set of functions. Here are some examples of data processing tasks using the NumPy package:

1. Array Creation: NumPy allows you to create arrays of different shapes and data types. For example, you can create a one-dimensional array using the `numpy.array()` function:

```
import numpy as np
```

```
# Create a one-dimensional array

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

print(arr)

# Output: [1 2 3 4 5]
```

2. Array Arithmetic: NumPy provides element-wise arithmetic operations on arrays. You can perform operations like addition, subtraction, multiplication, and division on arrays. For example:

```
import numpy as np

# Create two arrays

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

# Perform addition

result = arr1 + arr2

print(result)

# Output: [5 7 9]

# Perform multiplication

result = arr1 * arr2

print(result)

# Output: [4 10 18]
```

3. Array Indexing and Slicing: NumPy provides indexing and slicing operations to access and modify specific elements or subsets of an array. For example:

```
import numpy as np


# Create a one-dimensional array

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

# Access elements using indexing

print(arr[0])

# Output: 1

# Modify an element

arr[2] = 10

print(arr)

# Output: [1 2 10 4 5]
```

```
# Perform slicing

print(arr[1:4])

# Output: [2 10 4]
```

4. Array Aggregation: NumPy provides functions to aggregate data in arrays, such as calculating the sum, mean, minimum, maximum, and standard deviation. For example:

```
import numpy as np

# Create a one-dimensional array

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

# Calculate the sum

sum_result = np.sum(arr)

print(sum_result)

# Output: 15

# Calculate the mean

mean_result = np.mean(arr)

print(mean_result)

# Output: 3.0

# Find the minimum and maximum

min_result = np.min(arr)

max_result = np.max(arr)

print(min_result, max_result)

# Output: 1 5

# Calculate the standard deviation

std_result = np.std(arr)

print(std_result)

# Output: 1.41421356
```

These are just a few examples of the many data processing tasks that can be performed using the NumPy package. NumPy provides a wide range of functions and capabilities for efficient data manipulation, mathematical operations, and statistical analysis on arrays.

8a) i) Boolean Indexing: Boolean indexing allows you to select elements from an array based on a certain condition. You can create a Boolean mask, which is an array of the same shape as the original array, where each element is either `True` or `False` based on whether the corresponding element in the original array satisfies the condition. Let's see an example:

```
import numpy as np

# Create a one-dimensional array

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

# Create a Boolean mask

mask = arr > 2

print(mask)

# Output: [False False  True  True  True]

# Select elements based on the Boolean mask

selected_elements = arr[mask]

print(selected_elements)

# Output: [3 4 5]
```

In the above example, we create an array `arr` and then create a Boolean mask `mask` by checking if each element of `arr` is greater than 2. The resulting Boolean mask is `[False False True True True]`. We then use this mask to select only the elements from `arr` that correspond to `True` values in the mask, resulting in the array `[3 4 5]`.

ii) Fancy Indexing: Fancy indexing refers to indexing an array using another array or a list of indices. It allows you to select elements from an array based on a specified set of indices, even if they are not contiguous. Let's see an example:

```
import numpy as np

# Create a one-dimensional array

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

# Create an array of indices

indices = np.array([1, 3, 4])

# Select elements using fancy indexing

selected_elements = arr[indices]

print(selected_elements)

# Output: [2 4 5]
```

In this example, we have an array `arr`, and we create another array `indices` that contains the indices `[1, 3, 4]`. We then use this `indices` array to select the elements at those positions in the `arr` array. As a result, we get the array `[2 4 5]`, which consists of the elements at indices 1, 3, and 4 in the original array `arr`.

Fancy indexing can also be used for multidimensional arrays, where you can specify separate arrays for each dimension to select elements from different dimensions simultaneously.

8b) Here's a Python program that creates a NumPy array, sorts the elements into ascending order, and then into descending order:

```
import numpy as np

# Create a NumPy array

arr = np.array([5, 2, 9, 1, 7])

# Sort the array in ascending order

sorted_arr_ascending = np.sort(arr)

print("Array sorted in ascending order:")

print(sorted_arr_ascending)

# Sort the array in descending order

sorted_arr_descending = np.sort(arr)[::-1]

print("Array sorted in descending order:")

print(sorted_arr_descending)
```

In this program, we start by creating a NumPy array `arr` with some arbitrary values. Then, we use the `np.sort()` function to sort the array in ascending order, and we store the result in the variable `sorted_arr_ascending`. We print this sorted array, which will display the elements in ascending order.

Next, we sort the array in descending order by using the `[::-1]` slicing syntax, which reverses the order of elements in the array `sorted_arr_ascending`. We store the result in the variable `sorted_arr_descending` and print it, which will display the elements in descending order.

Here's an example output:

```
Array sorted in ascending order:[1 2 5 7 9]

Array sorted in descending order:[9 7 5 2 1]
```

9a) There are many NumPy array functions available but here are some of the most commonly used ones.

Array Operations	Functions
Array Creation Functions	`np.array()`, `np.zeros()`, `np.ones()`, `np.empty()`, etc.
Array Manipulation Functions	`np.reshape()`, `np.transpose()`, etc.
Array Mathematical Functions	`np.add()`, `np.subtract()`, `np.sqrt()`, `np.power()`, etc.
Array Statistical Functions	`np.median()`, `np.mean()`, `np.std()`, and `np.var()`.
Array Input and Output Functions	`np.save()`, `np.load()`, `np.loadtxt()`, etc.

Array creation functions allow us to create new NumPy arrays. For example,

```
import numpy as np

# create an array using np.array()
array1 = np.array([1, 3, 5])
print("np.array():\n", array1)

# create an array filled with zeros using np.zeros()
array2 = np.zeros((3, 3))
print("\nnp.zeros():\n", array2)

# create an array filled with ones using np.ones()
array3 = np.ones((2, 4))
print("\nnp.ones():\n", array3)
```

Output

```
np.array():
[1 3 5]
np.zeros():
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
np.ones():
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

NumPy array manipulation functions allow us to modify or rearrange NumPy arrays. For example,

```
import numpy as np

# create a 1D array
array1 = np.array([1, 3, 5, 7, 9, 11])

# reshape the 1D array into a 2D array
array2 = np.reshape(array1, (2, 3))

# transpose the 2D array
array3 = np.transpose(array2)

print("Original array:\n", array1)
print("\nReshaped array:\n", array2)
print("\nTransposed array:\n", array3)
```

Output

```
Original array:
[ 1  3  5  7  9 11]

Reshaped array:
 [[ 1  3  5]
 [ 7  9 11]]

Transposed array:
 [[ 1  7]
 [ 3  9]
 [ 5 11]]
```

In NumPy, there are tons of mathematical functions to perform on arrays. For example,

```
import numpy as np

# create two arrays
array1 = np.array([1, 2, 3, 4, 5])
array2 = np.array([4, 9, 16, 25, 36])

# add the two arrays element-wise
arr_sum = np.add(array1, array2)
```

```
# subtract the array2 from array1 element-wise
arr_diff = np.subtract(array1, array2)

# compute square root of array2 element-wise
arr_sqrt = np.sqrt(array2)


print("\nSum of arrays:\n", arr_sum)
print("\nDifference of arrays:\n", arr_diff)
print("\nSquare root of first array:\n", arr_sqrt)
```

Output

```
Sum of arrays:
[ 5 11 19 29 41]

Difference of arrays:
 [ -3  -7 -13 -21 -31]

Square root of first array:
 [2. 3. 4. 5. 6.]
```

NumPy provides us with various statistical functions to perform statistical data analysis.

```
import numpy as np

# create a numpy array
marks = np.array([76, 78, 81, 66, 85])

# compute the mean of marks
mean_marks = np.mean(marks)
print("Mean:",mean_marks)

# compute the median of marks
median_marks = np.median(marks)
print("Median:",median_marks)

# find the minimum and maximum marks
min_marks = np.min(marks)
print("Minimum marks:", min_marks)

max_marks = np.max(marks)
print("Maximum marks:", max_marks)
```

Output

```
Mean: 77.2
Median: 78.0
Minimum marks: 66
Maximum marks: 85
```

NumPy Array Input/Output Functions

```
import numpy as np

# create an array
array1 = np.array([[1, 3, 5], [2, 4, 6]])

# save the array to a text file
np.savetxt('data.txt', array1)

# load the data from the text file
loaded_data = np.loadtxt('array1.txt')

# print the loaded data
print(loaded_data)
```

Output

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

9b) Random numbers can be generated using the random module in Python's standard library or using the NumPy package. Here's an example of how to generate random numbers between 1 and 100 using the NumPy package:

import numpy as np

random_numbers = np.random.randint(1, 101, size=10)

print(random_numbers)

In the above code, `np.random.randint(1, 101, size=10)` generates an array of 10 random integers between 1 and 100. The `randint` function takes three arguments: the start value, the end value (exclusive), and the size of the output array.

o/p:[74 14 93 36 47 82 53 95 70 10]

Each time you run the code, a new set of random numbers will be generated