Machine Learning For Laymen

To The Point

Asmin Rijal, Asal Kc

Copyright

Preface

This mini-book/handout whatever you call it is written assuming you have no prior knowledge about Machine Learning Algorithms. The purpose of this book is to give you fundamental concepts and intuition about how some Machine Learning algorithms work. All the concepts here, we believe (and we hope it is the case) are explained in a simple yet logical way.

A proper understanding of foundational Mathematics, simple analytical thinking, and a little bit of will to learn, are the prerequisites to start your journey with this thing.

Contents

Chapter 1

Introduction

So you've officially stepped foot into the world of ML. Maybe you had already done it before but let us tell you the purpose of this handout is to take you far enough to go far in your journey.

1.1 What is Machine Learning?

What is the first thought that comes to your mind when you think of this? Maybe something like intelligent beings. The term *machine learning* refers to a broad class of methods that enable machines to learn from data and improve their performance on specific tasks without being explicitly programmed for each task. Rather than following fixed instructions, machine learning systems use data to guide their decision-making processes.

But how do machines learn in the first place? This is where *data* becomes essential. In most machine learning approaches, data serves as the foundation from which the model extracts useful information. Without sufficient data, meaningful learning is generally not possible. Based on the data it is given, a machine attempts to make predictions or decisions by leveraging patterns it has observed. Much like humans learn from experience, machines rely on past data to inform future behavior.

So how does a machine actually learn from data? In many machine learning algorithms, learning occurs through the identification of patterns and relationships within the data. Depending on the algorithm, this may involve adjusting internal values, constructing decision rules, or organizing data in a way that supports prediction. These learning mechanisms differ across models, but they all aim to improve performance on the given task.

At the core of many machine learning methods are two important components: *features* and *model components*. *Features* represent the input information provided to the algorithm.

The model components such as *parameters*, *rules*, or *structures* determine how the algorithm processes this information when encountering new, unseen data.

In algorithms that rely on trainable parameters, learning involves adjusting these parameters as more data is processed in order to reduce prediction errors. In other approaches, learning may instead involve storing data, refining decision boundaries, or updating internal representations. Regardless of the specific method used, the overall objective is to construct a model that performs well on new data. While these concepts and terms may initially seem unfamiliar to you, don't let that sway you away as it'll all make sense at the end of the day cause they form the foundation of how many machine learning systems operate.

1.2 Types of Machine Learning Algorithms

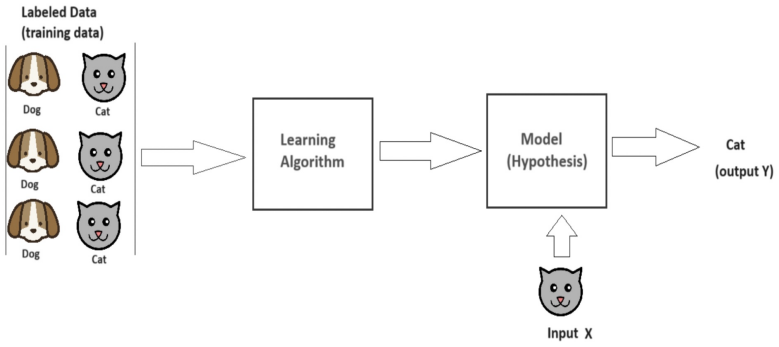
Somewhat analogous to humans—but again, not exactly—machines also learn through different techniques. These learning techniques can be broadly categorized based on how machines learn from data and the types of tasks they are designed to perform. Machine learning algorithms are commonly grouped into four categories based on their learning methodology: *supervised learning*, *unsupervised learning*, *semi-supervised learning*, and *reinforcement learning*.

In this book, we focus exclusively on *supervised learning* algorithms. We believe this provides a strong and accessible starting point for your machine learning journey.

Supervised Learning

Supervised learning is the most commonly used machine learning paradigm and the primary focus of this book. In supervised learning, the algorithm is trained on a labeled dataset, meaning that each input example is paired with a known output. The goal of the algorithm is to learn a mapping from inputs to outputs that can be used to make accurate predictions on new, unseen data.

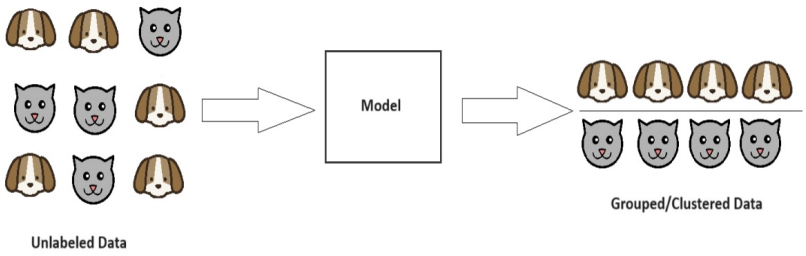
Typical supervised learning tasks include *classification*, where the output is a discrete label (such as dog or cat), and *regression*, where the output is a continuous value (such as predicting house prices) (we'll have a look it later). During training, the algorithm evaluates how far its predictions deviate from the true labels and adjusts its internal components accordingly to reduce this error.



Unsupervised Learning

Unsupervised learning is a type of machine learning that works with unlabeled data, where no explicit target outputs are provided. Instead of learning from predefined labels, unsupervised learning algorithms attempt to discover underlying structure, or relationships within the data.

In some unsupervised learning approaches, such as clustering, the algorithm groups similar data points together based on shared characteristics. For example, if raw images of cats and dogs are provided without labels, a clustering algorithm may group images with similar visual features, even though it does not know what a "cat" or a "dog" is.



However, not all unsupervised learning methods perform grouping. Some techniques focus on tasks such as dimensionality reduction, feature extraction, or anomaly detection. In these cases, the goal is not to assign data points to groups, but rather to transform, summarize, or analyze the data in a meaningful way.

Unsupervised learning is commonly used in applications such as exploratory data analysis, anomaly detection, and pattern discovery, and it often serves as a supporting tool within larger machine learning systems.

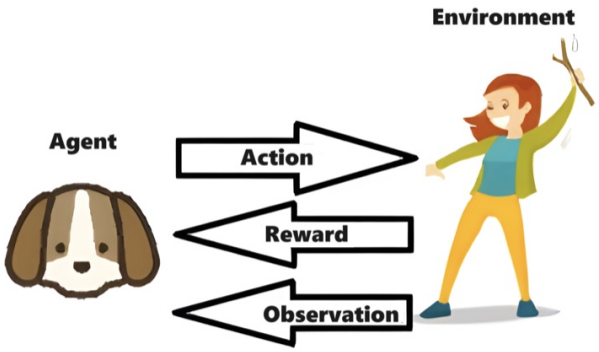
Semi-Supervised Learning

Semi-supervised learning, as the name suggests, combines aspects of both *supervised* and *unsupervised* learning by using a mixture of *labeled* and *unlabeled* data. This approach is particularly useful when only a small portion of the dataset is labeled while a large amount of *unlabeled* data is available.

By leveraging the limited labeled data alongside the structure present in the *unlabeled* data, semi-supervised learning can improve model performance without requiring extensive labeling. The model learns from the labeled examples and uses patterns and relationships within the unlabeled data to refine its understanding which ultimately improves its ability to generalize to new and unseen data.

Reinforcement Learning

Reinforcement learning is a type of machine learning in which an agent learns by interacting with an environment and receiving feedback in the form of rewards or penalties. Unlike supervised learning, reinforcement learning does not rely on labeled data. Instead, the agent learns by taking actions and observing the consequences of those actions over time.



In reinforcement learning, three key components exist: an agent, an environment, and a reward signal. The agent interacts with the environment by performing actions, and the environment responds by providing a reward that indicates how good or bad the action was. The goal of the agent is to learn a strategy, known as a policy, that maximizes the total reward over time.

A simple way to understand reinforcement learning is through training a dog. Here, the dog is the agent and its surroundings form the environment. When the dog performs a desired action, such as sitting on command, it receives a positive reward like a treat. Undesired actions receive little or no reward. Over time, through repeated feedback, the dog learns

which actions lead to better outcomes, just like a reinforcement learning agent learns to maximize rewards.

Chapter 2

Supervised Learning Algorithms

Data is what we'll be working with all the time. Given a dataset we'll need to know how to calculate some important information about numbers based on it. What will be the kind of data we work with then? Well, it can be anything from something like just an array of numbers to text files to big databases. So what are some basic things we must know by just looking at data then? There are often three values that immediately interest us. The measures of central tendency.

2.1 Light Stats Intro

2.1.1 Mean (Average)

The sum of all values divided by the number of values.

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Weakness: highly sensitive to **outliers**. If 9 dogs weigh $50kg$ but one dog weighs $500kg$, the average becomes useless.

```
import numpy

dogs_weight = [50,50,50,50,50,50,50,50,50,500]

x = numpy.mean(dogs_weight)

print(x)
```

```
95.0
```

You see we just used *numpy* library cause it makes life way easier. Could've done the same

thing without libraries too but why?

2.1.2 Median

The middle value when the data is sorted.

- If n is odd, it is the middle number.
- If n is even, it is the average of the two middle numbers.

Strength: very robust to outliers.

```
import numpy

dogs_weight = [50,50,50,50,50,50,50,50,50,500]

x = numpy.mean(dogs_weight)

print(x)
```

```
95.0
```

2.1.3 Mode

The most frequently occurring value in the dataset. Useful for categorical data (e.g., *"The most common breed of a dog"*). We had a look but *numpy* doesn't have a method for mode like *.mean()* or *.median()* but there is one in *SciPy*.

```
import numpy

dogs_weight = [50,50,50,50,50,50,50,50,50,500]

x = numpy.mean(dogs_weight)

print(x)
```

```
95.0
```

Then we have the measures of dispersion. entral tendency tells us where the data *is*, but dispersion tells us how *spread out* it is.

2.1.4 Range

The difference between the maximum and minimum values.

$$R = x_{max} - x_{min}$$

2.1.5 Variance

The average of the squared differences from the Mean. It measures how far a set of numbers is spread out from their average value.

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

```
import numpy

dogs_weight = [50,50,50,50,50,50,50,50,50,500]

x = numpy.var(dogs_weight)

print(x)
```

```
18225.0
```

2.1.6 Standard Deviation

The square root of the variance. It is preferred over variance because it is in the **same unit** as the original data.

$$\sigma = \sqrt{\sigma^2}$$

A low standard deviation means that most of the numbers are close to the mean value and a high standard deviation means that the values are spread out over a wider range.

```
import numpy

dogs_weight = [50,50,50,50,50,50,50,50,50,500]

x = numpy.std(dogs_weight)

print(x)
```

```
135.0
```

2.1.7 Coefficient of Variation (CV)

A standardized measure of dispersion, allowing comparison between datasets with different units or widely different means.

$$CV = \frac{\sigma}{\mu} \times 100\%$$

2.2 Some Python Libraries

2.2.1 Pandas

In machine learning, data rarely comes in a clean or ready-to-use format. One of the most commonly used Python libraries for data manipulation and analysis is `pandas`. Pandas provides powerful data structures that make it easy to load, clean, transform, and analyze structured data such as tables, spreadsheets, and CSV files. Here we'll only introduce you to the library briefly. You can always learn and practice yourself because these things don't really settle in your memory unless you do it yourself.
The two core data structures in Pandas are the `Series` and the `DataFrame`. A `Series` represents a one-dimensional labeled array, while a `DataFrame` represents a two-dimensional table consisting of rows and columns.

Importing Pandas

You probably know how to import libraries and also install them in the first place. This is what is conventionally done while important pandas.

```
import pandas as pd
```

Series

A pandas series looks like this. It is one-dimensional and the *.Series()* method takes a list as the parameter.

```
import pandas as pd

players = pd.Series(["Hazard", "Drogba", "KDB"])
print(players)
```

```
0    Hazard
1    Drogba
2       KDB
dtype: object
```

```
clubs = pd.Series(["Chelsea", "Yanited", "Sunderland"]) # sorry not sorry
print(clubs)
```

```
0       Chelsea
1       Yanited
2    Sunderland
dtype: object
```

Here we have two one-dimensional Panda series *players* and *clubs*. But often times, you'll have more than just one column of data. This is where *DataFrame* comes in.

DataFrame

A DataFrame is two-dimensional. It can be created from a *dictionary*, where *keys* represent column names and *values* represent the data:

```
import pandas as pd

players = pd.Series(["Hazard", "Drogba", "KDB"])
# print(players)

clubs = pd.Series(["Chelsea", "Yanited", "Sunderland"]) # sorry not sorry
# print(clubs)

c = {
    "Players": players,
    "Clubs": clubs
}
complete = pd.DataFrame(c)
print(complete)
```

```
   Players       Clubs
0  Hazard     Chelsea
1  Drogba     Yanited
2     KDB  Sunderland
```

We combined the two series we had and now we have a *DataFrame* called complete.

Importing Data

When working with real data, it comes in different formats. Pandas supports reading data from various file formats. The most common format in machine learning is CSV:

```
player_stats = pd.read_csv("players-stats.csv")
print(player_stats)
```

```
     Player         Team  Minutes Played  Contract Years            Price
0     Messi  Inter Miami            1500               2   $50,000,000.00
1  Ronaldo     Al Nassr            1620               2   $45,000,000.00
2   Mbappe          PSG            1410               3  $120,000,000.00
3  Haaland     Man City            1705               5  $110,000,000.00
4   Neymar     Al Hilal            1330               2   $60,000,000.00
5   Zidane  Real Madrid            2000               2   $30,000,000.00
```

We've used hypothetical data here. You can have a look at it in the final chapter.
As you can probably see, Pandas *DataFrames* like Python *Lists* start indexing at 0, and once we import a file as a DataFrame we can take advantages of all the tools that *pandas* leaves us at our hand.
Okay, so we trust your vocab for now and will assume that you understand the distinction between *rows* and *columns*. In the future, we'll start seeing a parameter called *axis*. For now remember that, *axis = 0* refers to *column-wise operation* that reduces it to a single row and *axis = 1* refers to *row-wise operation* that reduces it to a single column.

Exporting Data

You can export your dataframe into different file types once you're done with your side of the work.

```
player_stats = pd.read_csv("players-stats.csv")
# print(player_stats)


player_stats.to_csv("players-haru-ko-stats") # inside the quotes put what you want to name your file
                                             # when exported
```

Here, we've not added the parameter *index = False* to the *.to_csv()* method. So, when exporting the index also gets exported as an extra column as a result of which you will have to indexes when importing the same file (the one without the parameter).

Inspecting Data

Before using data in a machine learning model, it is important to understand its structure. There are some functions and attributes that are generally useful to know:

```
df.head()       # First five rows
df.tail()       # Last five rows
df.shape        # Number of rows and columns
df.columns      # Column names
df.info()       # Data types and missing values
df.describe()   # Statistical summary
```

```
      Player         Team  Minutes Played  Contract Years           Price
0      Messi  Inter Miami            1500               2   $50,000,000.00
…  1  Ronaldo     Al Nassr            1620               2   $45,000,000.00
   2   Mbappe          PSG            1410               3  $120,000,000.00
   3  Haaland     Man City            1705               5  $110,000,000.00
   4   Neymar     Al Hilal            1330               2   $60,000,000.00
      Player         Team  Minutes Played  Contract Years           Price
   1  Ronaldo     Al Nassr            1620               2   $45,000,000.00
   2   Mbappe          PSG            1410               3  $120,000,000.00
   3  Haaland     Man City            1705               5  $110,000,000.00
   4   Neymar     Al Hilal            1330               2   $60,000,000.00
   5   Zidane  Real Madrid            2000               2   $30,000,000.00
   (6, 5)
   Index(['Player', 'Team', 'Minutes Played', 'Contract Years', 'Price'], dtype='object')
   <class 'pandas.core.frame.DataFrame'>
   RangeIndex: 6 entries, 0 to 5
   Data columns (total 5 columns):
    #   Column          Non-Null Count  Dtype
   ---  ------          --------------  -----
    0   Player          6 non-null      object
    1   Team            6 non-null      object
    2   Minutes Played  6 non-null      int64
    3   Contract Years  6 non-null      int64
    4   Price           6 non-null      object
   dtypes: int64(2), object(3)
   memory usage: 372.0+ bytes
   None
          Minutes Played  Contract Years
   count        6.000000        6.000000
   mean      1594.166667        2.666667
   std        240.923570        1.211060
   min       1330.000000        2.000000
   25%       1432.500000        2.000000
   50%       1560.000000        2.000000
   75%       1683.750000        2.750000
   max       2000.000000        5.000000
```

This is how it looks on our data.

Selecting Data

Columns can be accessed using their names:

```
df["Player"]
```

Multiple columns can be selected together:

```
df[["Player", "Team"]]
```

Rows can be accessed using index-based selection:

```
df.iloc[0]     # First row
df.iloc[0:3]   # First three rows
```

Label-based selection can be done using `loc`:

```
df.loc[0, "Player"]
```

```
0      Messi
1    Ronaldo
2     Mbappe
3    Haaland
4     Neymar
5     Zidane
Name: Player, dtype: object

    Player         Team
0    Messi  Inter Miami
1  Ronaldo     Al Nassr
2   Mbappe          PSG
3  Haaland     Man City
4   Neymar     Al Hilal
5   Zidane  Real Madrid

Player                    Messi
Team                 Inter Miami
Minutes Played              1500
Contract Years                 2
Price              $50,000,000.00
Name: 0, dtype: object

    Player         Team  Minutes Played  Contract Years            Price
0    Messi  Inter Miami            1500               2   $50,000,000.00
1  Ronaldo     Al Nassr            1620               2   $45,000,000.00
2   Mbappe          PSG            1410               3  $120,000,000.00

Messi
```

This is what it would look like on our data. Also an important thing to note, *iloc* refers to *position* and *loc* refers to *index*. Cause you can set the index by yourself, it might not always be the standard 0-index with step 1. Therefore, it is important to note the difference.

```
color = pd.Series(["Red", "Blue", "Pink", "Yellow"], index = [1,2,5,8])
print(color)

print(color.loc[8])
print(color.loc[2])
print(color.iloc[2])
```

```
1       Red
2      Blue
5      Pink
8    Yellow
dtype: object
Yellow
Blue
Pink
```

This example should make things clearer for you. You can also have named indices and use *loc* to access them.

Filtering Data

Filtering allows us to select rows that meet specific conditions.

```
df[df["Points"] > 20]
df[df["Team"] == "Lakers"]
```

Multiple conditions can be combined:

```
df[(df["Points"] > 20) & (df["Assists"] > 5)]
```

```
                    Player       Team Position  Games  Points  Assists  \
3  Giannis Antetokounmpo      Bucks       PF     63    30.4      5.7
5           Luka Doncic  Mavericks       PG     58    32.4      8.0
7           Joel Embiid      76ers        C     50    33.1      4.2

   Rebounds      Salary
3      11.5  45600000.0
5       8.6  40000000.0
7      10.2         NaN

          Player    Team Position  Games  Points  Assists  Rebounds      Salary
0  LeBron James  Lakers       SF     55    25.3      7.4       7.8  47600000.0

                    Player       Team Position  Games  Points  Assists  \
0          LeBron James     Lakers       SF     55    25.3      7.4
1         Stephen Curry   Warriors       PG     60    29.1      6.3
2          Kevin Durant       Suns       SF     47    27.8      5.5
3  Giannis Antetokounmpo      Bucks       PF     63    30.4      5.7
4          Nikola Jokic    Nuggets        C     69    26.1      9.0
5           Luka Doncic  Mavericks       PG     58    32.4      8.0
8             Ja Morant  Grizzlies       PG     42    26.2      8.1
9          Jimmy Butler       Heat       SF     48    22.9      5.3

   Rebounds      Salary
0       7.8  47600000.0
1       5.2  51900000.0
2       6.6  44100000.0
3      11.5  45600000.0
4      12.3  47600000.0
5       8.6  40000000.0
8       5.9  34000000.0
9       6.1  37600000.0
```

This is the above three lines of code executed respectively.

Handling Missing Values

Real-world datasets often contain missing values. Some general methods in pandas look something like this.

```
df.isnull().sum()
df.dropna()
df.fillna(0)
```

Handling missing data is an important preprocessing step before training machine learning models.

```
...
Player      0
Team        0
Position    0
Games       0
Points      0
Assists     0
Rebounds    0
Salary      1
dtype: int64

                 Player       Team Position  Games  Points  Assists  \
0          LeBron James     Lakers       SF     55    25.3      7.4
1         Stephen Curry   Warriors       PG     60    29.1      6.3
2          Kevin Durant       Suns       SF     47    27.8      5.5
3  Giannis Antetokounmpo      Bucks       PF     63    30.4      5.7
4          Nikola Jokic    Nuggets        C     69    26.1      9.0
5           Luka Doncic  Mavericks       PG     58    32.4      8.0
6          Jayson Tatum    Celtics       SF     64    26.9      4.9
7           Joel Embiid      76ers        C     50    33.1      4.2
8             Ja Morant  Grizzlies       PG     42    26.2      8.1
9          Jimmy Butler       Heat       SF     48    22.9      5.3

   Rebounds        Salary
0       7.8  4.760000e+07
1       5.2  5.190000e+07
2       6.6  4.410000e+07
3      11.5  4.560000e+07
4      12.3  4.760000e+07
5       8.6  4.000000e+07
6       8.1  3.260000e+07
7      10.2  4.233333e+07
8       5.9  3.400000e+07
9       6.1  3.760000e+07
```

We ran the following code on our *nba-players-stats.csv* file to obtain the output above.

```
df = pd.read_csv("nba-players-stats.csv")


print(df.isnull().sum())
print()
df["Salary"] = df["Salary"].fillna(df["Salary"].mean())
print(df)
```

Creating New Columns

New columns can be created using existing ones:

```
df["Total"] = df["Points"] + df["Assists"] + df["Rebounds"]
```

```
df = pd.read_csv("nba-players-stats.csv")

df["Total"] = df["Points"] + df["Assists"] + df["Rebounds"]
print(df)
```

```
                  Player       Team Position  Games  Points  Assists  \
0          LeBron James     Lakers       SF     55    25.3      7.4
1         Stephen Curry   Warriors       PG     60    29.1      6.3
2          Kevin Durant       Suns       SF     47    27.8      5.5
3  Giannis Antetokounmpo      Bucks       PF     63    30.4      5.7
4          Nikola Jokic    Nuggets        C     69    26.1      9.0
5           Luka Doncic  Mavericks       PG     58    32.4      8.0
6          Jayson Tatum    Celtics       SF     64    26.9      4.9
7           Joel Embiid      76ers        C     50    33.1      4.2
8             Ja Morant  Grizzlies       PG     42    26.2      8.1
9          Jimmy Butler       Heat       SF     48    22.9      5.3

   Rebounds      Salary  Total
0       7.8  47600000.0   40.5
1       5.2  51900000.0   40.6
2       6.6  44100000.0   39.9
3      11.5  45600000.0   47.6
4      12.3  47600000.0   47.4
5       8.6  40000000.0   49.0
6       8.1  32600000.0   39.9
7      10.2         NaN   47.5
8       5.9  34000000.0   40.2
9       6.1  37600000.0   34.3
```

This process is known as feature engineering and is a key step in machine learning workflows.

Grouping Data

Grouping allows analysis across categories:

```
df.groupby("Team")["Points"].mean()
```

```
df = pd.read_csv("nba-players-stats.csv")

print(df.groupby("Team")["Points"].mean())
print()
print()
print(df.groupby("Position").size())
```

```
Team
76ers        33.1
Bucks        30.4
Celtics      26.9
Grizzlies    26.2
Heat         22.9
Lakers       25.3
Mavericks    32.4
Nuggets      26.1
Suns         27.8
Warriors     29.1
Name: Points, dtype: float64


Position
C     2
PF    1
PG    3
SF    4
dtype: int64
```

This is useful for summarizing and comparing different groups in a dataset.

There are many many different features that *pandas* equips you with. These are just some examples of how you could use them to clean your data and only touches the surface of what's possible. It is up to you guys to learn it on your own now, and make sure to practice with data by doing these things on your own.

2.2.2 NumPy

NumPy (Numerical Python) is the foundational library for numerical computing in Python. Almost every machine learning library you will encounter, including *Pandas*, *Scikit-learn*, *TensorFlow*, and *PyTorch*, is built on top of NumPy. Understanding NumPy is therefore essential for understanding how data is stored, manipulated, and computed efficiently in machine learning workflows.

Importing NumPy

For the sake of it, we're just putting it there.

```
import numpy as np
```

NumPy Arrays

The core object in NumPy is the `ndarray`. It represents a grid of values. It is the same as a list. The primary difference is that *NumPy* arrays are designed for efficient numerical operations on large, homogeneous (same type) data stored in contiguous memory, while *Python lists* are general-purpose, heterogeneous (mixed type) data structures with dynamic sizing capabilities.onk

Creating Arrays

From a Python list:

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

```
import numpy as np

l = [1,2,3,4,5]
arr = np.array(l)
print(arr)
type(arr)
```

```
[1 2 3 4 5]
numpy.ndarray
```

Two-dimensional array:

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

```
import numpy as np

l = [1,2,3]
m = [4,5,6]

arr = np.array([l,m])
print(arr)
print()
print(type(arr))
print()
print(arr.shape)
```

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

<class 'numpy.ndarray'>

(2, 3)
```

This might be easy to visualize as two rows and three columns. We like to think of it as two lists with 3 elements inside of them each, and there is always that one big list that contains them all.

Common Array Creation Functions

```
np.zeros((3, 3))     # Matrix of zeros
np.ones((2, 4))      # Matrix of ones
np.eye(3)            # Identity matrix
np.arange(0, 10, 2)  # Range with step size
np.linspace(0, 1, 5) # Evenly spaced values
```

```
import numpy as np

print(np.zeros((3, 3)))     # Matrix of zeros
print()
print(np.ones((2, 4)))      # Matrix of ones
print()
print(np.eye(3))           # Identity matrix
print()
print(np.arange(0, 10, 2))  # Range with step size
print()
print(np.linspace(0, 1, 5)) # Evenly spaced values
```

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

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

[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]

[0 2 4 6 8]

[0.   0.25 0.5  0.75 1.  ]
```

Array Attributes

Every NumPy array has useful attributes:

```
arr.shape    # Dimensions of the array
arr.ndim     # Number of dimensions
arr.size     # Total number of elements
arr.dtype    # Data type
```

These attributes become especially important when working with datasets and tensors.

```
import numpy as np

arr = np.eye(0, 10, 2)   # Range with step size

print(arr.shape)    # Dimensions of the array
print(arr.ndim)     # Number of dimensions
print(arr.size)     # Total number of elements
print(arr.dtype)    # Data type
```

```
(0, 10)
2
0
float64
```

Indexing and Slicing

Indexing works similarly to Python lists, but extends naturally to multiple dimensions.

One-Dimensional Indexing

```
arr[0]
arr[1:4]
```

Two-Dimensional Indexing

```
matrix[0, 1]      # Row 0, Column 1
matrix[:, 1]      # All rows, column 1
matrix[1, :]      # Row 1, all columns
```

Array Operations

One of NumPy's greatest strengths is *vectorization*. Operations are applied element-wise without explicit loops.

Arithmetic Operations

```
arr + 10
arr * 2
arr ** 2
```

```
import numpy as np

arr = np.array([1,2,3,4,5])
print(arr + 10)
print(arr*2)
print(arr**2)
```

```
[11 12 13 14 15]
[ 2  4  6  8 10]
[ 1  4  9 16 25]
```

Operations Between Arrays

```
import numpy as np

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

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

```
[5 7 9]
[ 4 10 18]
```

Broadcasting

Broadcasting is a powerful feature in NumPy that allows arithmetic operations between arrays of different shapes *without explicitly reshaping* them. This makes many elementwise operations fast and memory efficient because *Python* loops are avoided and computations happen internally in *optimized C code*.

What Is Broadcasting?

In NumPy, arrays usually need to have compatible shapes for element-by-element operations. Broadcasting relaxes this requirement by "stretching" the smaller array along dimensions with size 1 or missing dimensions so that it matches the shape of the larger array. The values are not actually copied in memory. This is handled efficiently behind the scenes.

Simple Broadcasting Examples

Scalar broadcast to a 2D array:

```
import numpy as np

a = np.array([[1, 2, 3],
              [4, 5, 6]])
x = 10

print(a + x)
```

```
[[11 12 13]
 [14 15 16]]
```

Broadcasting a 1D array to a 2D array:

```
import numpy as np

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

print(b + c)
```

```
[[ 5  7  9]
 [ 8 10 12]]
```

Here, `b` is broadcast across each row of `c` so that elementwise addition can be performed.

Practical Use Cases

Scaling rows of a matrix:

```
a = np.array([[10, 20],
              [40, 5]])
b = np.array([10, 20])

print(a * b)
```

```
[[100 400]
 [400 100]]
```

Each row of the matrix `a` is multiplied elementwise by the vector `b`.

Centering data:

```
d = np.array([[20, 20],
              [15, 50],
              [20, 20]])
mean = data.mean(axis=0)

print(d - mean)
```

```
[[  1.66666667 -10.        ]
 [ -3.33333333  20.        ]
 [  1.66666667 -10.        ]]
```

The feature-wise means are broadcast across all rows, enabling vectorized centering of data for machine learning preprocessing, in general.

Mathematical Functions

NumPy provides fast implementations of mathematical functions that you can check out. Here we only list some of them:

`np.mean(arr)`
`np.sum(arr)`
`np.std(arr)`
`np.min(arr)`
`np.max(arr)`

Element-wise functions:

`np.sqrt(arr)`
`np.log(arr)`
`np.exp(arr)`

Boolean Indexing

Boolean indexing allows filtering data based on conditions.

`arr[arr > 2]`

Multiple conditions:

`arr[(arr > 1) & (arr < 4)]`

This concept directly translates to dataset filtering in machine learning.

```
import numpy as np

arr = np.array([1, 2, 3, 9, -1, 2, 3, 4, 6,])
print(arr[arr > 2])

print(arr[(arr>2) & (arr != 6)])
```

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

Reshaping Arrays

Reshaping changes the structure without changing the data.

`arr.reshape(5, 1)`

`arr.reshape(-1, 1)`

The value `-1` tells NumPy to infer the correct dimension automatically.

```
import numpy as np

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

reshaped = arr.reshape(5, 1)

print(reshaped)
print(reshaped.shape)
```

```
[[1]
 [2]
 [3]
 [4]
 [5]]
(5, 1)
```

Random Numbers

Randomness plays a major role in machine learning and we use it all the time.

```
print(np.random.rand(3))
# Generates 3 random numbers drawn from a uniform distribution
# between 0 (inclusive) and 1 (exclusive)

print(np.random.randn(3))
# Generates 3 random numbers drawn from a standard normal
# distribution with mean 0 and variance 1

print(np.random.randint(0, 10, size=5))
# Generates 5 random integers between 0 (inclusive)
# and 10 (exclusive)
```

```
[0.70522026 0.76741169 0.9201164 ]
[ 0.67970796 -0.53999644 -0.41433428]
[2 2 3 8 2]
```

Also, there is a thing about something called "*seed*". Setting a *seed* ensures *reproducibility*. Setting a random seed ensures that the same sequence of random numbers is generated every time the code is run. This is especially important in machine learning for fair comparison of models.

```
np.random.seed(42)
np.random.rand(3)
# Always produces the same 3 numbers when the seed is set to 42
```

```
array([0.37454012, 0.95071431, 0.73199394])
```

Without setting a seed, *NumPy* uses a different starting point each time, resulting in different outputs on every run.

Random seeds are commonly used when:

- Initializing weights in neural networks
- Splitting datasets into training and testing sets
- Running experiments that must be reproducible so that no matter who is running the notebook, the same thing is outputted

Linear Algebra

NumPy includes a powerful linear algebra module.

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

print(np.dot(A, A))
print()
print(np.linalg.det(A))
print()
print(np.linalg.inv(A))
print()
print(np.linalg.eig(A))
```

```
[[ 7 10]
 [15 22]]

-2.0000000000000004

[[-2.   1. ]
 [ 1.5 -0.5]]

EigResult(eigenvalues=array([-0.37228132,  5.37228132]), eigenvectors=array([[-0.82456484, -0.41597356],
       [ 0.56576746, -0.90937671]]))
```

Linear algebra is the mathematical backbone of machine learning models and having a good handle on it definitely helps, though this is very simple and abstracts the meaning behind them. If possible, go into Linear Algebra courses because to get past a certain level, you definitely have to be very strong with the *Maths* behind things.

Pandas Dataframe out of NumPy arrays

2.2.3 Matplotlib

Visualizing data is a crucial step in understanding patterns, trends, and relationships. In Python, the most widely used library for creating plots and figures is **Matplotlib**. In this section, we will build a complete understanding of Matplotlib from the ground up, starting with basic plots and gradually moving toward customization and multiple plots.

What is Matplotlib?

Matplotlib is a Python library used for creating static, animated, and interactive visualizations. It is especially popular in data science and machine learning because it integrates seamlessly with NumPy and Pandas.
The most commonly used module in Matplotlib is `pyplot`, which provides a simple interface for creating figures and plots.

Importing Matplotlib

If Matplotlib is not already installed in your environment, it can be installed using:

```
pip install matplotlib
```

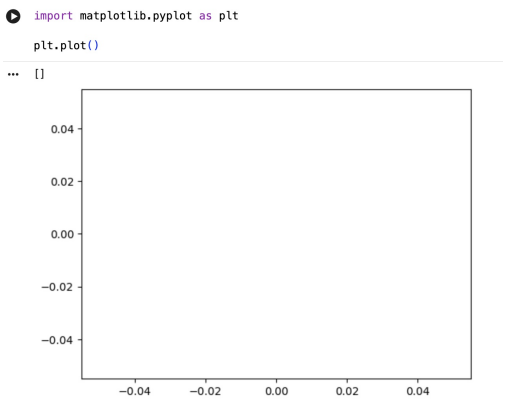
Once installed, we import it as follows:

```
import matplotlib.pyplot as plt
```

The alias `plt` is a widely accepted convention.

Your First Plot

First, we've created a plot without any data to show how it looks by default. We go on to add elements to this plot to add value to it all.



Let us now start with a simple line plot.

Create Data

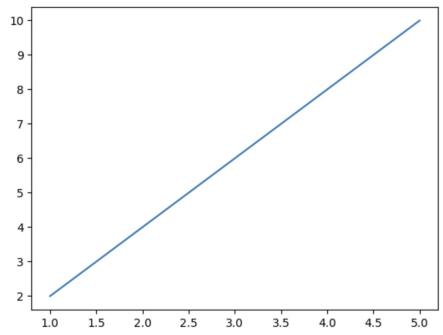
```
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]
```

Plot the Data

```
plt.plot(x, y)
plt.show()
```

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

plt.plot(x, y)
```

```
[<matplotlib.lines.Line2D at 0x7d308ecb79e0>]
```



Explanation:

- `plt.plot(x, y)` plots y versus x.
- `plt.show()` displays the figure. (we didn't have to use it cause it was done in *colab*.)

Adding Labels and Title

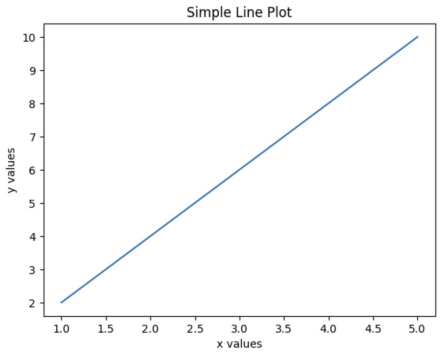
Plots are far more useful when they are properly labeled.

```
plt.plot(x, y)
plt.xlabel("x values")
plt.ylabel("y values")
plt.title("Simple Line Plot")
plt.show()
```

For the same x and y values as the list in the previous example:

```
plt.plot()
plt.plot(x, y)


plt.xlabel("x values")
plt.ylabel("y values")
plt.title("Simple Line Plot")
```

```
Text(0.5, 1.0, 'Simple Line Plot')
```

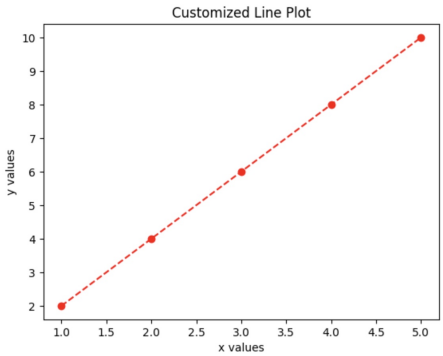


Customizing Plots

Matplotlib allows extensive customization that you can try out for yourself. Here we only present to you a simple customization

```
plt.plot(x, y, color="red", linestyle="--", marker="o")
plt.xlabel("x values")
plt.ylabel("y values")
plt.title("Customized Line Plot")
plt.show()
```

```
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

plt.plot(x, y, color="red", linestyle="--", marker="o")
plt.xlabel("x values")
plt.ylabel("y values")
plt.title("Customized Line Plot")
plt.show()
```



Common options:

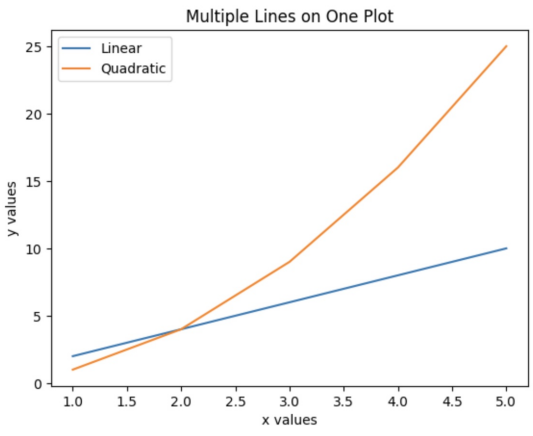
- Colors: `red`, `blue`, `green`, `black`
- Line styles: `-`, `--`, `:`
- Markers: `o`, `s`, `^`, `x`

Plotting Multiple Lines

We can plot more than one dataset on the same figure.

```
y2 = [1, 4, 9, 16, 25]

plt.plot(x, y, label="Linear")
plt.plot(x, y2, label="Quadratic")
plt.xlabel("x values")
plt.ylabel("y values")
plt.title("Multiple Lines on One Plot")
plt.legend()
plt.show()
```

```
plt.plot(x, y, label="Linear")
plt.plot(x, y2, label="Quadratic")
plt.xlabel("x values")
plt.ylabel("y values")
plt.title("Multiple Lines on One Plot")
plt.legend()
plt.show()
```



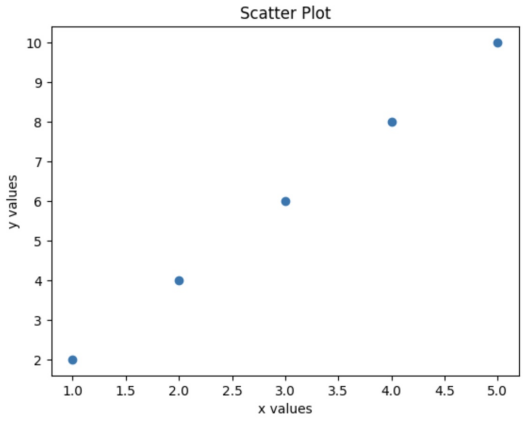
We're using the same x and y values as the previous list again here by the way.

Scatter Plots

Scatter plots are useful for visualizing relationships between variables. We can use *matplotlib* to plot all kinds of plots and scatter plots are no exceptions.

```
plt.scatter(x, y)
plt.xlabel("x values")
plt.ylabel("y values")
plt.title("Scatter Plot")
plt.show()
```

```
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

plt.scatter(x, y)
plt.xlabel("x values")
plt.ylabel("y values")
plt.title("Scatter Plot")
plt.show()
```

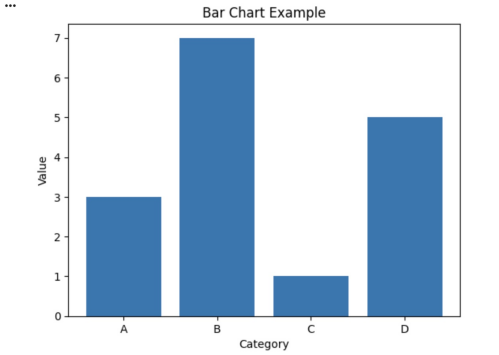


Bar Charts

Bar charts are commonly used to compare quantities across categories.

```
categories = ["A", "B", "C", "D"]
values = [3, 7, 1, 5]

plt.bar(categories, values)
plt.xlabel("Category")
plt.ylabel("Value")
plt.title("Bar Chart Example")
plt.show()
```

```
categories = ["A", "B", "C", "D"]
values = [3, 7, 1, 5]

plt.bar(categories, values)
plt.xlabel("Category")
plt.ylabel("Value")
plt.title("Bar Chart Example")
plt.show()
```



Histograms

Histograms show the distribution of numerical data.

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

plt.hist(data, bins=5)
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.title("Histogram Example")
plt.show()
```

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

plt.hist(data, bins=5)
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.title("Histogram Example")
plt.show()
```

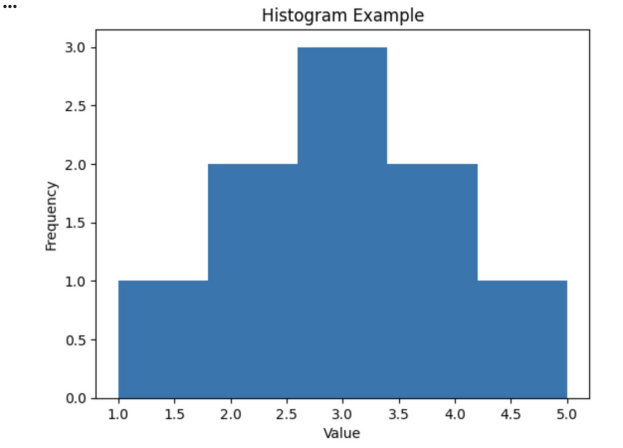


Figure Size and Resolution

You can control the size of the plot using `.figure()`.

```
plt.figure(figsize=(8, 5))
plt.plot(x, y)
```

Saving Figures

Plots can be saved to files instead of (or in addition to) displaying them.

```
plt.plot(x, y)
plt.savefig("plot.png")
plt.show()
```

The Object-Oriented (Axes) Approach

So far, we have used the `pyplot` interface, which is simple and convenient for quick plots. However, for more complex figures, Matplotlib provides a more powerful and flexible approach known as the **object-oriented (OO) or Axes approach**.
This approach gives explicit control over:

- Figures and Axes
- Multiple subplots
- Precise customization

In practice, this is the preferred method for professional and large-scale visualizations.

Core Idea

In Matplotlib,

- A **Figure** is the entire canvas (the window or page).
- An **Axes** is an individual plot within the figure.

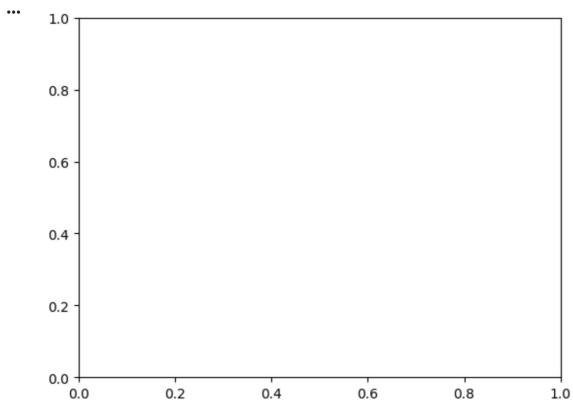
A figure can contain one or more Axes objects.

Creating a Figure and Axes

The standard way to begin is with `plt.subplots()`.

```
fig, ax = plt.subplots()
```

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
```



Here,

- `fig` is the Figure object
- `ax` is the Axes object

Basic Line Plot Using Axes

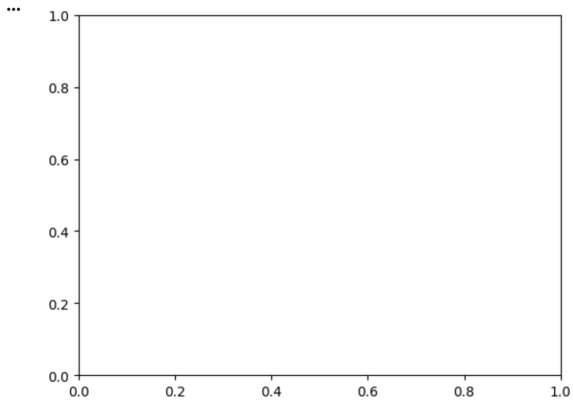
Like the last time, create Data

```
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]
```

Plot Using the Axes Object

```
fig, ax = plt.subplots()
ax.plot(x, y)
plt.show()
```

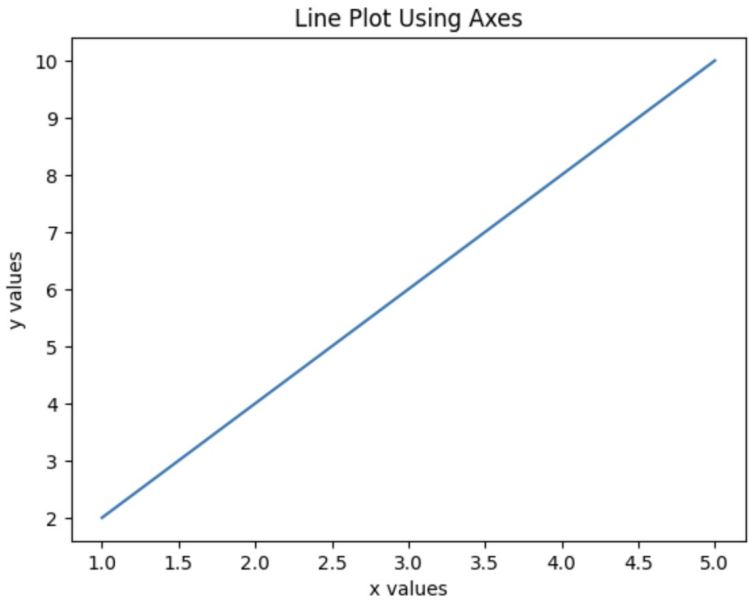
```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
```



Notice that we call `plot()` directly on the Axes object rather than using `plt.plot()`.

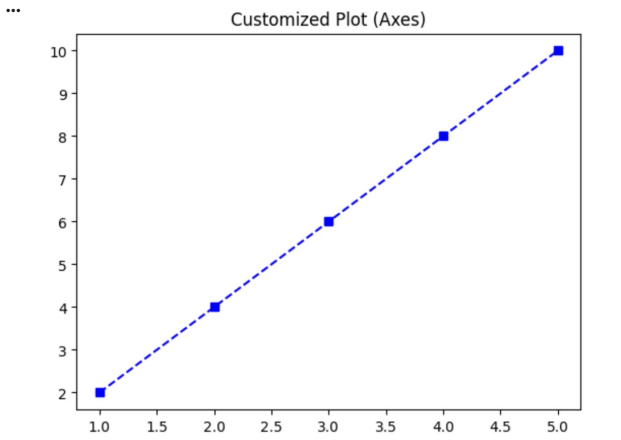
Adding Labels and Titles

```
fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_xlabel("x values")
ax.set_ylabel("y values")
ax.set_title("Line Plot Using Axes")
plt.show()
```



Customizing Lines with Axes

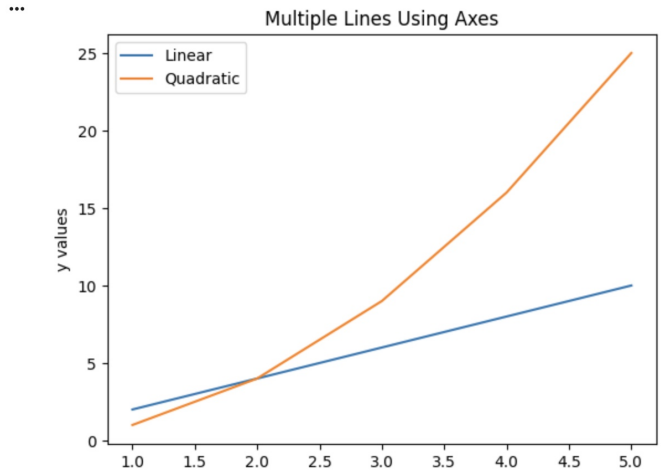
```
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]


fig, ax = plt.subplots()
ax.plot(x, y, color="blue", linestyle="--", marker="s")
ax.set_title("Customized Plot (Axes)")
plt.show()
```



All styling options available in `pyplot` work here as well.

Multiple Lines on the Same Axes

```
fig, ax = plt.subplots()
ax.plot(x, y, label="Linear")
ax.plot(x, y2, label="Quadratic")
ax.set_xlabel("x values")
ax.set_ylabel("y values")
ax.set_title("Multiple Lines Using Axes")
ax.legend()
plt.show()
```

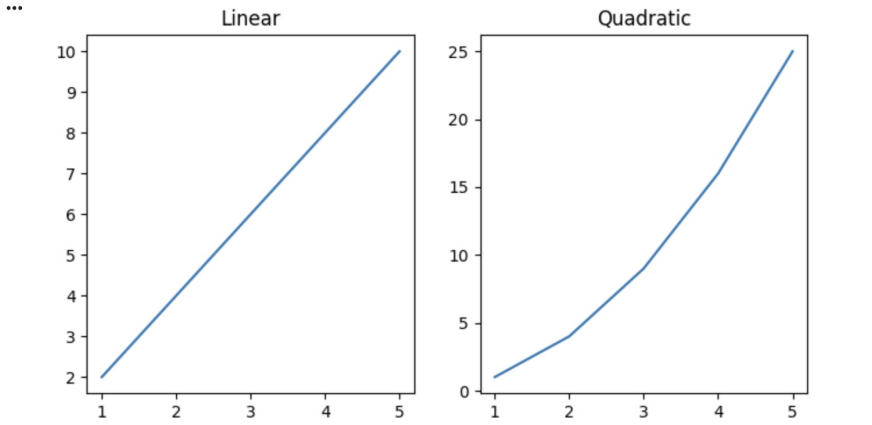


Multiple Subplots with Axes

This is really where it makes all the difference. The Axes approach truly shines when working with multiple plots.

```
fig, axs = plt.subplots(1, 2, figsize=(8, 4))

axs[0].plot(x, y)
axs[0].set_title("Linear")

axs[1].plot(x, y2)
axs[1].set_title("Quadratic")

plt.show()
```

```
fig, axs = plt.subplots(1, 2, figsize=(8, 4))

axs[0].plot(x, y)
axs[0].set_title("Linear")

axs[1].plot(x, y2)
axs[1].set_title("Quadratic")

plt.show()
```



Explanation:

- `axs` is an array of Axes objects
- Each subplot is controlled independently

Different Plot Types Using Axes

It's the same and not all that different from other methods. We've just put down what you can do and not the final chart itself for this. You guys can try it out on your own!

Scatter Plot

```
fig, ax = plt.subplots()
ax.scatter(x, y)
ax.set_title("Scatter Plot (Axes)")
plt.show()
```

Bar Chart

```
categories = ["A", "B", "C", "D"]
values = [3, 7, 1, 5]

fig, ax = plt.subplots()
ax.bar(categories, values)
ax.set_title("Bar Chart (Axes)")
plt.show()
```

Histogram

```
fig, ax = plt.subplots()
ax.hist(values, bins=5)
ax.set_title("Histogram (Axes)")
plt.show()
```

Pyplot vs Axes

Pyplot	Axes (OO)
Quick, simple plots	Complex and scalable plots
Implicit state	Explicit control
Harder to manage many plots	Ideal for multiple subplots

Take your time reviewing the material here, and also look for other resources. Mastering these fundamentals will allow you to visualize data effectively throughout your machine learning journey.

Sci-Kit Learn (sklearn)

Machine learning in practice is not just about understanding algorithms. Yes that is important, but it is about using reliable tools that allow us to implement ideas quickly, correctly, and at scale. In the Python ecosystem, `scikit-learn` (often abbreviated as `sklearn`) is the most widely used library for classical machine learning.
This chapter introduces `scikit-learn`, and the core workflow you will repeatedly use when it's your time.

What is `scikit-learn`?

It is an open-source Python library that provides:

- Simple and efficient implementations of machine learning algorithms
- A consistent and intuitive API across models
- Tools for data preprocessing, model evaluation, and model selection
- Seamless integration with `NumPy`, `SciPy`, and `pandas`

It is designed primarily for:

- Supervised learning (regression and classification)
- Unsupervised learning (clustering, dimensionality reduction)
- Model evaluation and validation

Why `scikit-learn`?

Unlike deep learning frameworks that focus on neural networks, `scikit-learn` excels at:

- Interpretable models
- Small to medium-sized datasets
- Rapid experimentation
- Clean, readable code

Note: For beginners, its greatest strength is consistency: once you learn how to use one model, you already know how to use most others.

Importing sklearn

For the sake of it again,
If you are using `pip`, install `scikit-learn` with:

```
pip install scikit-learn
```

and for sklearn the standard practice is to import the specific modules that we want to use in our program which you will see down the line.

Core `scikit-learn` Workflow

Almost every machine learning task in `scikit-learn` follows the same four-step pattern:

1. Import the model
2. Create an instance of the model
3. Fit the model to data
4. Use the model to make predictions

Note: This uniform structure is one of the most important ideas to internalize.

Your First Model

Let's have a look at a simple *regression* example.

Importing the Model

This is the standard way we import specific objects so that we don't have to keep calling the dot operator on the main library that we import.

```
from sklearn.linear_model import LinearRegression
```

Creating the Model

```
model = LinearRegression()
```

So far, at this stage, the model has learned nothing. It is an empty mathematical object waiting for data.

Fitting the Model

Suppose `X` contains input features and `y` contains target values.

```
model.fit(X, y)
```

Now, this is where the learning happens. The model estimates parameters that best explain the relationship between `X` and `y`.

Making Predictions

```
predictions = model.predict(X_new)
```

Once trained, the model can now make predictions on unseen data.

Key API Concepts

As we've titled the book, we're trying to keep the whole thing as clear and as simple as possilbe. So, the terms kinda mean the following in sense.

`fit`

The `fit` method always means,

> *Learn from data*

`predict`

The `predict` method always means,

> *Use what was learned to make predictions*

`transform`

Commonly used in preprocessing and dimensionality reduction, it means,

> *Change the representation of the data*

Some objects use both `fit` and `transform` together by using:

```
fit_transform()
```

Train–Test Split

To evaluate a model fairly, we split data into training and testing sets. As in the example below, generally, the training and testing samples are divided into 80/20 out of a 100 respectively.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

The model is trained on `X_train` and evaluated on `X_test`.

Model Evaluation

This typically depends on which model we want to evaluate.
For regression tasks, a common metric is mean squared error.

```
from sklearn.metrics import mean_squared_error

mse = mean_squared_error(y_test, model.predict(X_test))
```

For classification, accuracy is often used:

```
from sklearn.metrics import accuracy_score

accuracy = accuracy_score(y_test, y_pred)
```

Framework To Keep in Mind

Regardless of the what algorithm or technique you might end up using, always think in terms of:

$$\texttt{Data} \rightarrow \texttt{Model} \rightarrow \texttt{Fit} \rightarrow \texttt{Predict} \rightarrow \texttt{Evaluate}$$

Once this workflow feels natural, learning new models in `scikit-learn` should feel almost effortless.

By the end of this book, `scikit-learn` will feel less like a library and more like a language you speak fluently.

Examples with Real Data

Now that we understand the workflow, let's apply it to real datasets. We'll explore three different supervised learning techniques on publicly available data that you can access and experiment with yourself.

Iris Classification

It's a classic in machine learning. It contains measurements of 150 iris flowers from three species. If you wanna have a look at it outside, you check it out **here**.What our task is, for now, is to predict the species based on measurements.

Loading the Data

The beauty of this dataset is that it comes built into `scikit-learn`, so we can just load the data and play around for ourselves:

```
from sklearn.datasets import load_iris
import pandas as pd

# Load the dataset
iris = load_iris()
X = iris.data  # features: sepal length, sepal width,
               # petal length, petal width
y = iris.target  # Target: species (0, 1, or 2)

# converting to DataFrame for better visualization
df = pd.DataFrame(X, columns=iris.feature_names)
df['species'] = iris.target_names[y]
print(df.head())
print(df.tail())
print("Feature names:", iris.feature_names)
print("Target names:", iris.target_names)
```

```
…    sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

   species
0  setosa
1  setosa
2  setosa
3  setosa
4  setosa
     sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
145                6.7               3.0                5.2               2.3
146                6.3               2.5                5.0               1.9
147                6.5               3.0                5.2               2.0
148                6.2               3.4                5.4               2.3
149                5.9               3.0                5.1               1.8

        species
145  virginica
146  virginica
147  virginica
148  virginica
149  virginica
Feature names: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
Target names: ['setosa' 'versicolor' 'virginica']
```

K-Nearest Neighbors (KNN)

KNN (K-Nearest Neighbors) is one of the simplest machine learning algorithms. Here's how it works:

Imagine you're trying to identify a flower species based on its petal and sepal measurements. KNN looks at the k training examples that are most similar to your new flower (the *nearest neighbors*), and predicts the species based on what those neighbors are.

For example, if $k = 3$ and you're classifying a new flower:

1. Find the 3 training flowers with the most similar measurements (using distance)
2. Look at what species those 3 flowers are (say 2 are setosa, 1 is versicolor)
3. Predict the most common species among them (setosa wins with 2 votes)

Something to note in your mind:

The k in *KNN* is a parameter you choose. A smaller k (like 3) makes the model sensitive to individual data points, while a larger k (like 10) makes predictions smoother but potentially less accurate.

Let's have a look at a sample code in the following page:

KNN

```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, classification_report

# Splitting the data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# creating and training the models
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

# making predictions
y_pred = knn.predict(X_test)

# evaluating our model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
print(classification_report(y_test, y_pred,
                            target_names=iris.target_names))
```

```
Accuracy: 1.00
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        10
  versicolor       1.00      1.00      1.00         9
   virginica       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

We've not played around with the dataset a lot here. If you want to have a look at how things actually work from scratch, you can check this out ***here***.

California Housing Prices Dataset

This dataset contains information about housing districts in California from the 1990 census. Our task is to predict median house prices based on features like location, population, and median income. What do you think would be the idea algorithm to go over this? Let's first load the data and convert it into a *pandas* dataframe.

Loading the Data

```
# loading the dataset
from sklearn.datasets import fetch_california_housing
import pandas as pd
import numpy as np

cali = fetch_california_housing()
cali_df = pd.DataFrame(cali.data, columns=cali.feature_names)
cali_df['MedHouseVal'] = cali.target

cali_df.head()
```

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude	MedHouseVal
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23	4.526
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22	3.585
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24	3.521
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25	3.413
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25	3.422

```
print("Dataset Description:")
cali_df.describe()
```

Dataset Description:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude	MedHouseVal
count	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000
mean	3.870671	28.639486	5.429000	1.096675	1425.476744	3.070655	35.631861	-119.569704	2.068558
std	1.899822	12.585558	2.474173	0.473911	1132.462122	10.386050	2.135952	2.003532	1.153956
min	0.499900	1.000000	0.846154	0.333333	3.000000	0.692308	32.540000	-124.350000	0.149990
25%	2.563400	18.000000	4.440716	1.006079	787.000000	2.429741	33.930000	-121.800000	1.196000
50%	3.534800	29.000000	5.229129	1.048780	1166.000000	2.818116	34.260000	-118.490000	1.797000
75%	4.743250	37.000000	6.052381	1.099526	1725.000000	3.282261	37.710000	-118.010000	2.647250
max	15.000100	52.000000	141.909091	34.066667	35682.000000	1243.333333	41.950000	-114.310000	5.000010

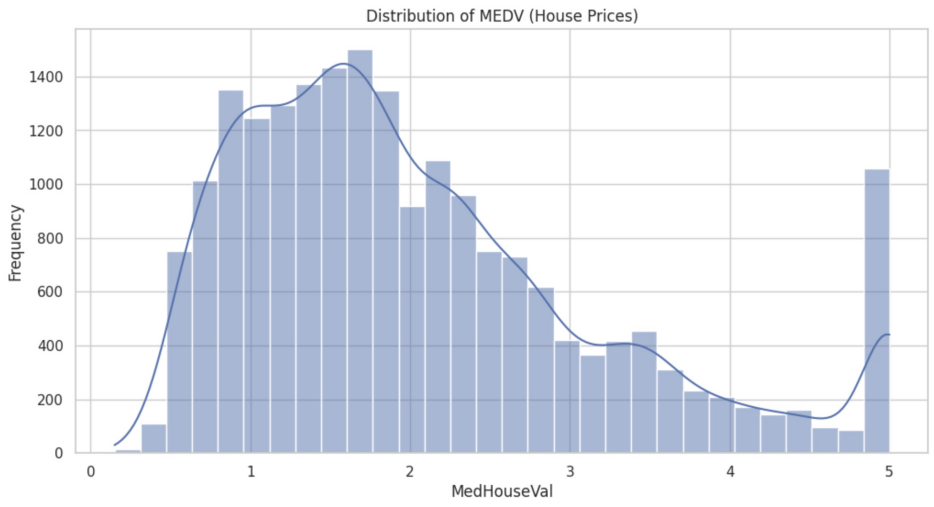
We know that this dataset has no missing values, but it is always a good practice to check that out first.

```
print("Missing Values in the Data")
print(cali_df.isnull().sum())
```

```
Missing Values in the Data
MedInc          0
HouseAge        0
AveRooms        0
AveBedrms       0
Population      0
AveOccup        0
Latitude        0
Longitude       0
MedHouseVal     0
dtype: int64
```

Let's visualize our dataset:

```
import matplotlib.pyplot as plt
import seaborn as sns

sns.set(style='whitegrid')
plt.figure(figsize=(12, 6))
# since, we've not talked about seaborn a lot in the book,
# what the single line of code down below does is:
# Plots a histogram of the 'MedHouseVal' column from cali_df
# - bins=30 splits the data into 30 intervals
# - kde=True overlays a Kernel Density Estimate to show the distribution shape
sns.histplot(cali_df['MedHouseVal'], kde=True, bins=30)
plt.title('Distribution of MEDV (House Prices)')
plt.xlabel('MedHouseVal')
plt.ylabel('Frequency')
plt.show()
```

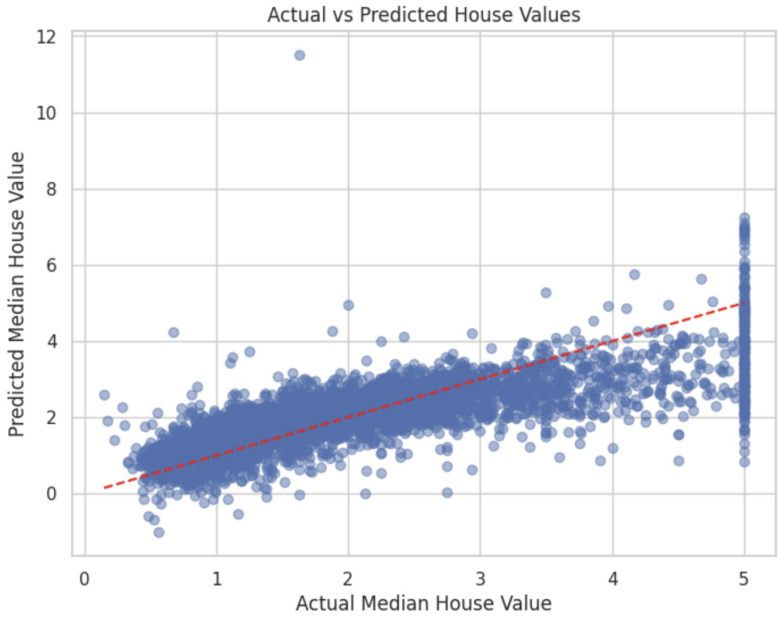


```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

# separate input features (X) and target variable (y)
X = cali_df.drop(columns=['MedHouseVal'])
y = cali_df['MedHouseVal']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# initializing the linear regression model
model = LinearRegression()

# fitting the model on the training data
model.fit(X_train, y_train)

# predicting house values for the test set
y_pred = model.predict(X_test)
```

```
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# mean absolute error: average absolute difference between predictions and true values
mae = mean_absolute_error(y_test, y_pred)

# mean Squared Error: penalizes larger errors more strongly
mse = mean_squared_error(y_test, y_pred)

# R^2 score: proportion of variance explained by the model
r2 = r2_score(y_test, y_pred)

print("Model Evaluation:")
print("MAE:", mae)
print("MSE:", mse)
print("R^2:", r2)
```

```
Model Evaluation:
MAE: 0.5332001304956553
MSE: 0.5558915986952444
R^2: 0.5757877060324508
```

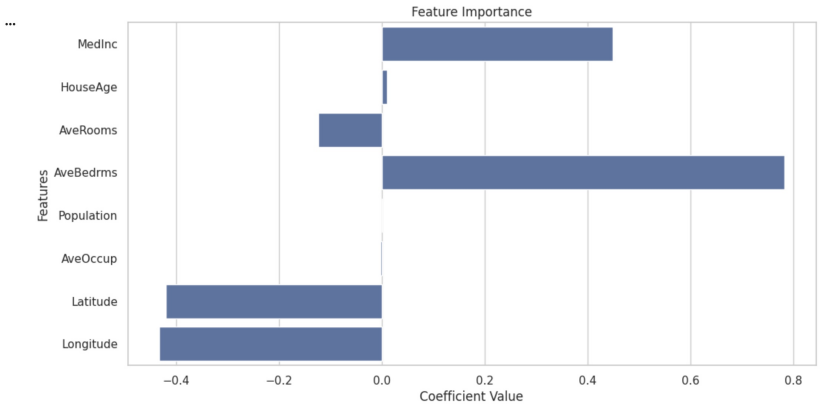
What we get when we compare the prediction and the actual values:



The important thing that we learn is the relation between the different features:

```
# storing the feature names and their learned coefficients
coefficients = pd.DataFrame({
    'Feature': X.columns,
    'Coefficient': model.coef_
})

print(coefficients)
```

```
      Feature  Coefficient
0      MedInc     0.448675
1    HouseAge     0.009724
2    AveRooms    -0.123323
3   AveBedrms     0.783145
4  Population    -0.000002
5    AveOccup    -0.003526
6    Latitude    -0.419792
7   Longitude    -0.433708
```



Here, the linear regression model provided some valuable insights into what factors affect house prices and demonstrated its power in predictive modeling. While our model achieved decently reasonable performance, further improvements could be made by exploring more complex models for this specific dataset or additional data preprocessing techniques.

This is how much we'll have a look so far. We make iterations based on what feels right and necessary in the book. Till then keep learning!

Chapter 3

Python Refresher

Python is a high-level, interpreted programming language known for its readability and simplicity. These qualities make it especially well suited for data analysis and machine learning applications. Here we have some *Python* concepts as a refresher whenever you need it.

Datatypes

In Python, variables are used to store data values. Unlike some other programming languages, Python does not require explicit type declarations. The type of a variable is inferred from the value it is assigned. What do I mean by this?

```
n = 1
hi = "I love you"
print(type(n))
print(type(hi))
```

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

Did we explicitly specify the datatype here? No. If you have a slight exposure to any of the other low-level languages, you probably know that it is not as convenient and you need to specify it.

Some of the most commonly used basic data types include integers (`int`), floating-point numbers (`float`), boolean values (`bool`), and strings (`str`). These data types form the foundation upon which more complex data structures and computations are built.

```
n = 1
hi = "I love you"
p = 3.14
flag = True
print(type(n))
print(type(hi))
print(type(p))
print(type(flag))
```

```
<class 'int'>
<class 'str'>
<class 'float'>
<class 'bool'>
```

Basic Arithmetic and Comparisons

Below are examples of basic arithmetic and comparison operations in Python.

```
# Assigning numerical values to variables
x = 10
y = 3

# Basic arithmetic operations
sum_value = x + y        # Addition
difference = x - y       # Subtraction
product = x * y          # Multiplication
quotient = x / y         # Division (returns a float)

# Print results
print(sum_value)
print(difference)
print(product)
print(quotient)

# Comparison operations
print(x > y)             # Checks if x is greater than y
print(x < y)             # Checks if x is less than y
print(x == y)            # Checks if x is equal to y
print(x != y)            # Checks if x is not equal to y
```

```
13
7
30
3.3333333333333335
True
False
False
True
```

Arithmetic operations are frequently used in machine learning to compute quantities such as averages, distances, or error values. Comparison operations return boolean values (`True` or `False`) and are commonly used in conditional statements.

```python
# Assigning numerical values to variables
x = 10
y = 3

modulo = x % y           # Modulus (remainder)
exponent = x ** y        # Exponentiation

print(modulo)
print(exponent)
```

```
1
1000
```

Comments and Docstrings

Comments and docstrings help make code way easier to understand and maintain for you when you visit back after some time or for someone who's seeing your code for the first time. They are not printed in the output console as you probably saw in the previous few snippets. Single-line comments begin with the # symbol and are ignored by Python when the program

```python
# This is a single-line comment.
# It is used to explain what the code is doing.

def calculate_average(numbers):
    """
    This function takes a list of numbers
    and returns their average value.
    """
    total = sum(numbers)      # Add all values in the list
    count = len(numbers)      # Count how many values there are
    return total / count      # Compute and return the average

# Call the function with sample data
values = [2, 4, 6, 8]
average = calculate_average(values)

print(average)
```

```
5.0
```

runs. Docstrings are enclosed in triple quotes and are used to document the purpose and behavior of functions, classes, or modules. Very useful!!

Print Function

The standard *print* function. Let's look at some basic different ways you can use it.

1) the standard way - this prints out the string inside the quotes "" as it is.

```
print("Hello World")
```

```
Hello World
```

2) printing variables - we do not use quotes "" whenever we want to print out the variables as they're just containers that contain what we want to actually output.

```
hi = "Hello World"
print(hi)
```

```
Hello World
```

3) printing different datatypes in the same print function - you can use comma (,) to separate them.

```
hi = "I love you"
print(hi, 1000)
```

```
I love you 1000
```

4) using f-string

```
age = 18
# what we could do
print("The person is", age, "years old")

# we use an f-string to embed variables directly in the string
print(f"The person is {age} years old")
```

```
The person is 18 years old
The person is 18 years old
```

User-Defined Functions

So far, we have looked at functions such as `print()` and `type()`, which are examples of *built-in functions* in Python. Built-in functions are readily available and perform common tasks without requiring us to define their behavior. While these functions are extremely useful, they only cover a small portion of what we can do with Python.

In practice, especially in machine learning workflows, we often need to perform the same operation repeatedly or organize our code in a clean and reusable way. This is where *user-defined functions* come into play. A user-defined function allows us to group a set of instructions under a single name so that it can be executed whenever needed.

In Python, functions are defined using the `def` keyword, followed by the function name and a pair of parentheses. Any input values that the function needs are passed as *parameters* inside these parentheses.

```
def greet():
    print("Hello, welcome to our machine learning handout!")
```

Once a function is defined, it does not run automatically. To execute the function, we must explicitly call it using its name followed by parentheses.

```
def greet():
    print("Hello, welcome to our machine learning handout!")

greet()
```

```
Hello, welcome to our machine learning handout!
```

Functions can also take input values and return outputs. These returned values are especially important in machine learning, where functions often compute quantities such as losses, predictions, or gradients. Just putting it out there cause this is specific to machine learning...

```
def square(x):
    return x * x

result = square(5)
print(result)
```

```
25
```

Also, a key thing to note is that your program is executed line by line. Therefore, the function must be defined before it is called.

Control Flow

Control flow statements allow us to execute code conditionally and repeatedly.

Conditional Statements: if, elif, else

Use `if`, `elif`, and `else` to execute code blocks based on conditions.

```
x = 10
if x > 10:
    print("x is greater than 10")
elif x == 10:
    print("x is exactly 10")
else:
    print("x is less than 10")
```

```
x is exactly 10
```

The thing to note is all the *elif*(s) and the *else* that follow up only run if the previous condition fails. These conditional statements come in very handy and are fundamental to one's programming knowledge.

Loops: for and while

Loops are used to iterate over a sequence or execute a block of code multiple times. Here's a simple example of doing the same thing with *for* and *while* loops.

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

```
count = 0
while count < 5:
    print(count)
    count += 1
```

```
0
1
2
3
4
```

When to use a for loop:

- When iterating over a dataset or list of values
- When the number of iterations is known
- When processing each element in a collection (if you don't get what this means you'll eventually get it if you follow the refresher)

```
# Loop over a list of values
values = [10, 20, 30]

for v in values:
    print(v)
```

```
10
20
30
```

You might not get this just yet if you're not familiar with *lists* before you've read this but just know that this is very very handy.

When to use a while loop:

- When looping depends on a condition rather than a count
- When the stopping point is not known beforehand
- When iterating until convergence or a threshold is reached

Key Differences

- A `for` loop iterates over a sequence or collection.
- A `while` loop continues as long as a condition is true.
- `for` loops are generally safer and less prone to infinite loops.
- `while` loops require careful condition updates to avoid running indefinitely.

Choosing the Right Loop

As a general rule, use a `for` loop whenever possible. It is more readable and less error-prone. Use a `while` loop when the problem naturally depends on a condition rather than a fixed number of iterations.

range() Function

The `range` function is commonly used with `for` loops to generate a sequence of numbers. It does not create a list directly, but instead produces values one at a time, which makes it memory-efficient. When `range` is given a single argument, it generates numbers starting from 0 up to (but not including) that value as you might have seen in the previous example of a *for* loop. This loop prints the numbers 0 through 4.

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

Specifying a Start and End Value

The `range` function can take two arguments: a starting value and an ending value.

```
for i in range(2, 6):
    print(i)
```

```
2
3
4
5
```

This prints the numbers 2, 3, 4, and 5.

Using a Step Size

A third argument can be used to specify the step size, which controls how much the value increases (or decreases) each time.

```
for i in range(0, 10, 2):
    print(i)
```

```
0
2
4
6
8
```

This prints the even numbers between 0 and 8.

Counting Backwards

The `range` function can also generate values in reverse by using a negative step size.

```
for i in range(5, 0, -1):
    print(i)
```

```
5
4
3
2
1
```

This prints the numbers 5, 4, 3, 2, and 1.

Using `range` with Lists

`range` is often used to access list elements by index.

```
values = [10, 20, 30]

for i in range(len(values)):
    print(values[i])
```

```
10
20
30
```

This iterates over the list using indices. Again, don't worry we will definitely go over lists.

Break and Continue

`break` exits the loop immediately, while `continue` skips the rest of the current iteration and moves to the next one. These come in very handy in specific use cases so it's good if you have a good grasp on them early on.

```
for i in range(10):
    if i == 5:
        break  # Stops the loop when i is 5
    if i % 2 == 0:
        continue  # Skips printing even numbers
    print(i)
```

```
1
3
```

Data Structures

Python provides some powerful built-in data structures to store collections of data.

Lists

Lists are ordered, mutable collections. If you are familiar with arrays, they're almost the same except you can have different data types stored in the same list which is not doable with arrays.

```
fruits = ["apple", "banana", "cherry"]
fruits.append("date")  # Adding an element to the end of the list
fruits.remove("banana")  # Removing an element
print(fruits[0])  # Accessing the first element
```

```
apple
```

Key Features of Python Lists

- Lists are **ordered**, meaning elements have a fixed position.
- Lists are **mutable**, so elements can be modified after creation.
- Lists can store **different data types** in the same structure.
- Lists support indexing and slicing.
- Lists grow and shrink dynamically.

```
a = ["hello", 2, True, 2.5, "GTA"]
print(a[1:3])
```

```
[2, True]
```

You can see an example of list slicing here where you can get parts of a list. It behaves almost the same like a range function. The *first index* is where it starts and it stops at *last index - 1*.

```
numbers = [0, 1, 2, 3, 4, 5]
print(numbers[1:4])  # Output: [1, 2, 3]
print(numbers[::-1]) # Reverse list: [5, 4, 3, 2, 1, 0]
```

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

You can see that if we add another *colon(:)* we can define the step-size like a range function. You can access elements using indices and create sub-lists using slicing. Slicing is very very handy. Make sure to get enough practice with slicing lists.

To determine if a specified item is present in a list, we can use the in keyword and this is very very handy too:

```
fruits = ["orange","apple", "banana", "mango"]
if "banana" in fruits:
  print("Yep, 'banana' is in the fruits list")
```

```
Yep, 'banana' is in the fruits list
```

As we've previously discussed in the control flow section, you can use a *for* loop to iterate over a list if you don't want to use indexing. This comes in handy too.

```
fruits = ["orange","apple", "banana", "mango"]
for fruit in fruits:
  print(fruit)
```

```
orange
apple
banana
mango
```

List Comprehensions

Just another concise way to create lists to keep things quick and simple.

```
squares = [x**2 for x in range(10)]
print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

It also offers a shorter syntax when you want to create a new list based on the values of an existing list. If you didn't use list comprehension you would use something along this line:

```
players = ["Messi", "Hazard", "Terry", "Drogba", "KDB"]
newlist = []

for player in players:
  if "a" in player:
    newlist.append(player)

print(newlist)
```

```
['Hazard', 'Drogba']
```

whereas using list comprehension gives you shorter concise way of doing the same thing.

```
players = ["Messi", "Hazard", "Terry", "Drogba", "KDB"]
newlist = [player for player in players if "a" in player]

print(newlist)
```

```
['Hazard', 'Drogba']
```

This is a very basic introduction to list comprehension and you can add more details along the way but just get the basic idea and you get used to it with practice.
Just think of this as the basic outline and work your way up.

```
newlist = [expression for item in iterable if condition == True]
```

where iterable can be any *iterable* object, like a list, tuple, set etc.

Copying a List

You cannot copy a list by just typing *lista = listb* cause *lista* will only be a reference to *listb*, and changes made in *lista* will automatically also be made in *listb*. Therefore, here are some techniques you can use to do it:
1) using *copy()* method

```
players = ["Messi", "Hazard", "Terry", "Drogba", "KDB"]
mylist = players.copy()
players.remove("Messi")
print(mylist)
print(players)
```

```
['Messi', 'Hazard', 'Terry', 'Drogba', 'KDB']
['Hazard', 'Terry', 'Drogba', 'KDB']
```

2) using *list()* method

```
players = ["Messi", "Hazard", "Terry", "Drogba", "KDB"]
mylist = list(players)
players.remove("Messi")
print(mylist)
print(players)
```

```
['Messi', 'Hazard', 'Terry', 'Drogba', 'KDB']
['Hazard', 'Terry', 'Drogba', 'KDB']
```

3) using list slicing

```
players = ["Messi", "Hazard", "Terry", "Drogba", "KDB"]
mylist = players[:]
players.remove("Messi")
print(mylist)
print(players)
```

```
['Messi', 'Hazard', 'Terry', 'Drogba', 'KDB']
['Hazard', 'Terry', 'Drogba', 'KDB']
```

Joining Lists

From what we've learned so far, you must be able to come up with at least one way to do it. The easiest way is to just use the + operator and concatenate two lists.

```
players = ["Messi", "Hazard", "Terry"]
players2 = ["Drogba", "KDB"]
players = players + players2
print(players)
```

```
['Messi', 'Hazard', 'Terry', 'Drogba', 'KDB']
```

We can also use the *append()* method and pass a list there, but that appends the list inside the list which is not something we want. It looks like this.

```
players = ["Messi", "Hazard", "Terry"]
players2 = ["Drogba", "KDB"]
players.append(players2)
print(players)
```

```
['Messi', 'Hazard', 'Terry', ['Drogba', 'KDB']]
```

What we can instead do is loop over each item in the other list and append them one by one:

```
players = ["Messi", "Hazard", "Terry"]
players2 = ["Drogba", "KDB"]
for player in players2:
  players.append(player)
print(players)
```

```
['Messi', 'Hazard', 'Terry', 'Drogba', 'KDB']
```

We can also use the *extend()* method to do it.

```
players = ["Messi", "Hazard", "Terry"]
players2 = ["Drogba", "KDB"]
players.extend(players2)
print(players)
```

```
['Messi', 'Hazard', 'Terry', 'Drogba', 'KDB']
```

There are many methods you can learn on your own when you need it and it's not possible to remember every one of them. It is also not feasible. Therefore, the true way of learning programming is doing it yourself. Just make use of them and you'll get them.

Tuples

Tuples are ordered, immutable collections.

```
point = (10, 20)
# point[0] = 5  # this would raise an error
x, y = point  # unpacking
print(x, y)
```

```
10 20
```

```
point = (10, 20)
point[0] = 5  # this would raise an error
x, y = point  # unpacking
print(x, y)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipython-input-2794492565.py in <cell line: 0>()
      1 point = (10, 20)
----> 2 point[0] = 5  # this would raise an error
      3 x, y = point  # unpacking
      4 print(x, y)

TypeError: 'tuple' object does not support item assignment
```

Accessing items in a tuple is also the same as a list. You can do it using indices or loops.

How do we add items to a tuple then?

We know that they are immutable, but there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

```
players = ("Messi", "Hazard", "Terry")
players2 = list(players)
players2.append("Drogba")
players = tuple(players2)

print(players)
```

```
('Messi', 'Hazard', 'Terry', 'Drogba')
```

Here, we only appended a single item but since you've converted it into a list, you can perform all actions on it that you would perform on a list and turn it back into a tuple in the end. Other way you could do it is just add another tuple to it; that is allowed.

```
players = ("Messi", "Hazard", "Terry")
players2 = ("Drogba", "KDB")
players = players + players2

print(players)
```

```
('Messi', 'Hazard', 'Terry', 'Drogba', 'KDB')
```

Sets

Sets are unordered collections of unique unindexed elements. Once you create a set you cannot change(modify) an element within it, but you can definitely add or remove elements.

```
unumbers = {1, 2, 2, 3}
print(unumbers) # {1, 2, 3}
```

```
{1, 2, 3}
```

In a set, *False* and 0 are treated the same and *True* and 1 are treated the same.
So we said you elements in a set are unindexed, so how do we access them then?
1) we can use a *for*-loop and *in*

```
players = {"Messi", "Hazard", "Terry"}
for playa in players:
  print(playa)
```

```
Hazard
Messi
Terry
```

Adding items to a set

1) can use an *add()* method (just to add one item though)

```
players = {"Messi", "Hazard", "Terry"}
players.add("Drogba")

print(players)
```

```
{'Hazard', 'Drogba', 'Messi', 'Terry'}
```

2) can use *update()* method to add elements from another set/any other iterable to existing set to which this method is applied

```
players = {"Messi", "Hazard", "Terry"}
players2 = {"Drogba", "KDB"}

players.update(players2)

print(players)
```

```
{'KDB', 'Terry', 'Drogba', 'Hazard', 'Messi'}
```

```
players = {"Messi", "Hazard", "Terry"}
players2 = ("Drogba", "KDB")

players.update(players2)

print(players)
```

```
{'Hazard', 'KDB', 'Drogba', 'Terry', 'Messi'}
```

Dictionaries

Dictionaries store data in key-value pairs. It is a collection which is ordered, changeable and does not allow duplicates.

```
person = {"name": "Alice", "age": 25}
print(person["name"])
person["city"] = "New York"  # Adding a new key-value pair
print(person)
```

```
Alice
{'name': 'Alice', 'age': 25, 'city': 'New York'}
```

```
# Creating a dictionary
student = {
    "name": "Arjun",
    "age": 18,
    "major": "CS",
    "gpa": 3.8
}

# Accessing values using keys
print(student["name"])    # Alex
print(student["gpa"])     # 3.8

# Updating a value
student["age"] = 22

# Adding a new key-value pair
student["graduated"] = False

# Looping through key-value pairs
for key, value in student.items():
    print(key, ":", value)
```

```
Arjun
3.8
name : Arjun
age : 22
major : CS
gpa : 3.8
graduated : False
```

Dictionaries are pretty important. Above we had some examples of what one can do with it. Practice stuff yourself too. Get your hands dirty. Also, note that something *immutable* cannot be the *key* of the list.

Working with Files

Handling file input and output is crucial for data processing, especially when you go to action in the real world. Things go beyond inputting in the console, so treat this part as something really important.

The main function for working with files is the *open()* function which takes two parameters: *filename*, and *mode*. There are 4 different modes to open a file:
1) r - Read - Default value. Opens a file for reading, error if the file does not exist
2) a - Append - Opens a file for appending, creates the file if it does not exist
3) w - Write - Opens a file for writing, creates the file if it does not exist
4) x - Create - Creates the specified file, returns an error if the file exists
Source: [1]
In addition you can specify if the file should be handled as binary or text mode.
You can learn more about file handling from other sources and try it out yourself too.

OOP(Object Oriented Programming) is necessary and you should definitely have a look at it. We decided not to cover it for now. Might come back to this and do it ourselves, but for now we'll be listing down some resources for people to learn further and OOP resources will be there.
We never introduced you to this but you should be able to download packages using `pip`. As projects grow in complexity, especially in machine learning, it becomes impractical to write all functionality from scratch. Python addresses this by providing a package manager called `pip`, which allows users to install, update, and manage external libraries. `pip` gives access to thousands of open-source packages from the Python Package Index (PyPI), including widely used machine learning libraries such as `numpy`, `pandas`, `matplotlib`, and `scikit-learn`. Instead of manually downloading and configuring these libraries, `pip` automates the process with simple commands.

To install a package, you just execute the following in the terminal or command prompt:

```
pip install numpy
```

Once installed, the package can be imported and used directly within a Python script or notebook:

```
import numpy
```

Chapter 4

What next?

If you went over all the chapters. Great job! You're set to continue on a journey of your own. We might come up with something just like this in the indefinite future, but there are already a lot of great resources to learn things by yourself. We highly recommend that you check out the sources in *bibliography* which will be really helpful. Learning from the source of the source is great for you. This is completely optional by the way and we're just putting it out there for those who might want a direction going forward.

Bibliography

[1] W3Schools. *Python Tutorial*. Available at: https://www.w3schools.com/python/

[2] Daniel Bourke. YouTube/Udemy Available at: https://www.youtube.com/@mrdbourke

[3] Ericka42. *Kaggle*. Available at: https://www.kaggle.com/code/ericka42/linear-regression-with-the-california-housing/notebook

Chapter 5

Image Credits

https://medium.com/analytics-vidhya/a-beginners-guide-to-reinforcement-learning-88a330d8d94e

Chapter 6

Resources To Check Out

https://damiantgordon.com/Courses/OOP/OOP-Workbook.pdf
https://youtu.be/eWRfhZUzrAc?si=GQleHK2m1lqNqjbV
https://users.cs.duke.edu/ cynthia/teaching.html
https://www.youtube.com/@mrdbourke

Chapter 7

Data Used

Unfortunately couldn't figure out a way to make downloadable files. Might make changes in the future. As of now, you can do the same or something similar in *Excel* and export it as a csv file for practice purposes.

players-stats.csv :

A	B	C	D	E
Player	Team	Minutes Played	Contract Years	Price
Messi	Inter Miami	1500	2	$50,000,000.00
Ronaldo	Al Nassr	1620	2	$45,000,000.00
Mbappe	PSG	1410	3	$120,000,000.00
Haaland	Man City	1705	5	$110,000,000.00
Neymar	Al Hilal	1330	2	$60,000,000.00
Zidane	Real Madrid	2000	2	$30,000,000.00

nba-players-stats.csv :

A	B	C	D	E	F	G	H
Player	Team	Position	Games	Points	Assists	Rebounds	Salary
LeBron James	Lakers	SF	55	25.3	7.4	7.8	47600000
Stephen Curry	Warriors	PG	60	29.1	6.3	5.2	51900000
Kevin Durant	Suns	SF	47	27.8	5.5	6.6	44100000
Giannis Antetokounmpo	Bucks	PF	63	30.4	5.7	11.5	45600000
Nikola Jokic	Nuggets	C	69	26.1	9	12.3	47600000
Luka Doncic	Mavericks	PG	58	32.4	8	8.6	40000000
Jayson Tatum	Celtics	SF	64	26.9	4.9	8.1	32600000
Joel Embiid	76ers	C	50	33.1	4.2	10.2	
Ja Morant	Grizzlies	PG	42	26.2	8.1	5.9	34000000
Jimmy Butler	Heat	SF	48	22.9	5.3	6.1	37600000