Introduction to Data Science

What is Data Science?

Welcome to this course on Data Science. If you are active on Fb/Insta/X, **share your Data Science journey on these platforms and tag me**. I would love to see your progress.

I am so happy you decided to learn Data Science with me in my style. Lets step back and talk a bit about why do we need data science. What kind of tasks we will do as a data scientist in an organization

What is Data Science?

At its core, **Data Science** is the field of study that uses **mathematics, statistics, programming, and domain knowledge** to extract **meaningful insights** from data. Well that sounds like a bookish definition which you are free to write in your semester exams but Data Science pretty much blends various techniques from different disciplines to analyze large amounts of information and solve real-world problems.

Simple Definition:

Let me give you a very simple non fancy definition of Data Science

> Data Science is the process of collecting, cleaning, analyzing, and interpreting data to make informed decisions.

Why is Data Science Important?

In today's world, **data is everywhere**—from your online shopping habits to the sensors in smart devices. Companies use this data to:

- **Make better decisions** (e.g., which product to launch next).
- **Predict outcomes** (e.g., weather forecasts or stock prices).
- **Automate processes** (e.g., self-driving cars).
- **Personalize experiences** (e.g., Netflix recommendations, YouTube recommendations).

Key Steps in Data Science:

1. **Data Collection** – Gathering raw data from various sources (websites, databases, IoT devices, etc.).
2. **Data Cleaning** – Fixing errors and handling missing values (this takes up **80% of a data scientist's time**).
3. **Data Analysis** – Using statistical methods to find patterns and insights.
4. **Model Building** – Applying **machine learning** to make predictions.
5. **Interpretation & Communication** – Presenting findings in a clear way to help decision-making.

Where Do We See Data Science in Action?

- **Healthcare:** Predicting disease outbreaks and improving diagnoses.
- **E-commerce:** Personalized product recommendations.
- **Finance:** Detecting fraudulent transactions by recognizing patterns.
- **Entertainment:** Content recommendations on platforms like YouTube and Netflix.

Why Should You Care About Data Science?

- **High Demand:** There's a global shortage of skilled data scientists.
- **Great Pay:** The average salary of a data scientist in the US is **$120,000+ per year**; in India, ₹10-30 LPA for experienced professionals.
- **Future-Proof Career:** It powers everything from AI to business strategy—this field is only growing.

Note: The content you just read will be supplied to you as a downloadable PDF. Since I am writing a summarized version of the video content which will serve as revision notes, I recommend you try to read them along with watching the videos. Thanks and see you in the next lecture!

Data Science Lifecycle

The **Data Science Lifecycle** refers to the structured process used to extract insights from data. It involves several stages, from gathering raw data to delivering actionable insights. Here is a breakdown of each step:

1. Problem Definition

> Understanding the problem you want to solve.

- Identify business objectives and define the question to answer.
- Later, we will do a very interesting project called "Coders of Delhi" where we start by understanding the business objective.
- Example: "Can we predict customer churn?" or "What factors drive sales?". In "Coders of Delhi" project, the problem is how to find potential friends of a given person in a social network

Key Activities:

- Collaborate with stakeholders.
- Define success metrics.
- Set project goals and deliverables.

---

2. Data Collection

> Gathering relevant data from multiple sources.

- Sources may include databases, APIs, web scraping, or third-party datasets. Sometimes if this step is taken care of by another team or it's a data dump given by another team, we don't care where it came from.

Key Activities in data collection:

- Identify data sources (structured vs. unstructured).
- Collect data using SQL, Python, or automated pipelines.
- Ensure data relevance and completeness.

---

3. Data Cleaning (Data Preprocessing)

> Preparing raw data for analysis.

- This step addresses **missing values, duplicates, and inconsistencies**.

Key Activities:

- Handle missing or incorrect data.
- Standardize formats and remove duplicates.
- Manage outliers and inconsistencies.

Fun Fact: Data scientists spend **80% of their time** cleaning data!

---

4. Data Exploration (EDA – Exploratory Data Analysis)

> Analyzing data patterns and relationships.

- Understand data distributions and detect anomalies using visualizations.

Key Activities:

- Summarize data using statistics (mean, median, etc.).

- Visualize patterns (using **Matplotlib**, **Seaborn**, etc.).
- Identify correlations and outliers. (correlation is how two variables move in relation to each other and outlier is a data point that stands out as unusually different from the rest. Eg. A 190 Kg heavyweight person is an outlier)

---

5. Model Building

> Creating and training machine learning models.

- Use algorithms to predict outcomes or classify data.

Key Activities:

- Choose appropriate models (e.g., regression, decision trees, neural networks).
- Split data into training and testing sets.
- Train and fine-tune models.

Common Tools: Scikit-learn, TensorFlow, PyTorch.

---

6. Model Evaluation

> Measuring model performance and accuracy.

- Evaluate models using metrics to ensure reliability.

Key Activities:

- Use performance metrics (e.g., accuracy, RMSE, ROC curve) to answer questions like - *"How often is my model correct?"*
- Perform cross-validation for robustness. Train using some part of the data and test using some part and average out the accuracy. We will study this in detail later
- Compare multiple models for best outcomes.

Key Metrics:

- Classification: Accuracy, Precision, Recall, F1-Score.

- Regression: RMSE, R-squared.

---

7. Deployment

> Integrating the model into production systems.

- Deliver actionable results through APIs or dashboards.

Key Activities:

- Package the model for deployment (Usually done using web frameworks like Flask, and FastAPI).
- Automate pipelines for continuous learning (MLOps).
- Monitor performance post-deployment.

---

8. Communication & Reporting

> Sharing insights with stakeholders. At the end of the day the ML model solves a problem and proper reporting it to the concerned department is very important

Key Activities:

- Create dashboards
- Present findings clearly and concisely.
- Document the process and results.

---

9. Maintenance & Iteration

> Keeping the model accurate and up-to-date.

Key Activities:

- Monitor model performance.

- Update models with new data.
- Refine features and parameters.

Summary

The **Data Science Lifecycle** is a continuous, iterative process involving:

1. Problem Definition
2. Data Collection
3. Data Cleaning
4. Data Exploration
5. Model Building
6. Model Evaluation
7. Deployment
8. Communication & Reporting
9. Maintenance & Iteration

By following this lifecycle, data scientists transform raw data into meaningful insights that drive better decision-making.

Data Science Tools

I am enjoying teaching so far. When working in data science, the right tools make your work easier, faster, and more efficient. When I started my data science journey at IIT Kharagpur, I used to code using Pycharm and regular Python installation. I knew about Jupyter but wasn’t familiar with its capabilities. From writing code to visualizing data, there are many options to choose from. Here is a breakdown of popular data science tools and why **Anaconda** with **Jupyter Notebook** is an excellent choice for beginners and advanced users.

How to run Python programs

The easiest way to run Python programs is by installing VS Code and using pip to install packages but we will use Anaconda and Jupyter notebooks

1. Jupyter Notebook (with Anaconda Distribution)

> An open-source web application that allows you to create and share documents with live code, equations, visualizations, and text.

Why Use Anaconda with Jupyter Notebook?

- **User-Friendly:** Interactive coding and easy-to-follow outputs, perfect for beginners.
- **All-in-One Package:** Anaconda includes essential libraries (NumPy, Pandas, Matplotlib + 1,500 other popular packages) pre-installed.
- **Ideal for Data Science:** Quick prototyping, data visualization, and exploratory analysis.
- **Environment Management:** Easily create isolated environments to manage package dependencies.

Common Use Cases:

- Data exploration and visualization
- Machine learning experiments
- Sharing research and reports

Installation: Althought we will do a detailed installation of Anaconda in the later sections, you can download and install the Anaconda distribution from anaconda.com. It includes Jupyter Notebook by default.

Command to Launch Jupyter Notebook:

```
jupyter notebook
```

Don't worry, we will do all these things step by step in the next section

---

2. Google Colab

> A free, cloud-based Jupyter Notebook environment provided by Google.

Why Use Google Colab?

- **Free GPU/TPU Access:** Great for deep learning without requiring expensive hardware.
- **Cloud-Based:** No local setup—just log in and start coding.
- **Collaboration:** Share notebooks via links for easy collaboration.

Common Use Cases:

- Machine learning and deep learning projects
- Quick experiments without local setup
- Collaborative projects

Access: Use Google Colab directly in your browser at colab.research.google.com.

---

3. VS Code (Visual Studio Code)

> A lightweight and powerful code editor by Microsoft with robust extensions.

Why Use VS Code?

- **Customizable:** Extensive extensions for Python and data science (e.g., Python extension by Microsoft).
- **Integrated Jupyter Support:** You can run Jupyter Notebooks directly in VS Code.
- **Debugging Tools:** Advanced debugging capabilities.

Common Use Cases:

- Large-scale data science projects
- Working with multiple languages (Python, R, etc.)
- Integrated development (data pipelines, APIs)

Installation: Can be downloaded from code.visualstudio.com but we will install and use Anaconda distribution throughout this Data Science course.

---

4. PyCharm

> A powerful, professional IDE for Python development by JetBrains.

Why Use PyCharm?

- **Professional Features:** Advanced code navigation, refactoring, and debugging.
- **Environment Management:** Virtual environment and package management built-in.
- **Scientific Mode:** Built-in support for Jupyter notebooks.

Common Use Cases:

- Large, production-level data science projects
- Building Python-based machine learning applications

Installation: Download from jetbrains.com/pycharm.

---

5. Cursor AI

> An AI-powered code editor designed for enhanced productivity with machine learning assistance.

Why Use Cursor AI?

- **AI Integration:** Code suggestions and completions for faster development.
- **Context-Aware:** Understands complex data science workflows.
- **Collaborative:** Works well with team-based projects.

Common Use Cases:

- Assisted coding for data science
- Accelerating research and prototyping
- Team collaboration

Access: You can visit cursor.so to download cursor AI but I don't recommend using it just yet. Its important to understand the basics of data science and programming before you use such AI assistants

---

Which Tool Should You Choose?

Tool	Best For	Key Advantage
Jupyter Notebook	Interactive analysis, education	Easy to use and visualize data
Google Colab	Deep learning, cloud-based projects	Free GPU/TPU and no local setup
VS Code	Large projects, debugging	Lightweight with advanced features
PyCharm	Enterprise-level, complex applications	Professional IDE with deep features
Cursor AI	AI-assisted coding, productivity	AI-enhanced code suggestions
Spyder	Academic research, scientific computing	MATLAB-like interface

Recommendation: If you're starting out or want a hassle-free experience, **Anaconda with Jupyter Notebook** is the best choice. For advanced AI and big data projects, **Google Colab** is an excellent free alternative. For robust, large-scale development, **VS Code** or **PyCharm** provides advanced capabilities. Since we are just starting our learning journey, I will be using Anaconda and Jupyter for the most part of this course

---

Summary

1. **Anaconda + Jupyter Notebook:** Best for beginners and interactive analysis.
2. **Google Colab:** Ideal for cloud-based work and deep learning.

3. **VS Code:** Perfect for integrated, large-scale projects.
4. **PyCharm:** A professional-grade IDE for Python development.
5. **Cursor AI:** AI-assisted productivity for fast development.

Choosing the right tool depends on your project size, complexity, and hardware needs. For most data science workflows, **Anaconda with Jupyter Notebook** offers the best balance of simplicity, flexibility, and power.

Installing Anaconda

Anaconda is a popular Python distribution designed for data science and machine learning. It comes pre-installed with essential libraries like NumPy, pandas, and scikit-learn, along with tools like Jupyter Notebook. This makes it easy to set up a consistent environment without worrying about dependencies or compatibility issues. It’s beginner-friendly and saves time by simplifying package management and project setup.

How to Install Anaconda

1. **Download Anaconda:** Go to https://www.anaconda.com/products/distribution. It might ask you for your email but there is also a “Skip Registration” button to directly download anaconda right away without email.
2. **Choose Your OS:** Select your operating system (Windows, macOS, or Linux).
3. **Download the Installer:** Click the appropriate download button (64-bit recommended).
4. **Run the Installer**
    1. **Windows/macOS**: Open the downloaded file and follow the setup instructions.
    2. **Linux**: Dowload the bash file and run it in terminal like this:

       ```
       bash Anaconda3-*.sh
       ```

       Dont forget to replace Anaconda3-* with the correct filename
5. **Verify Installation:** Open terminal or Anaconda Prompt and type:

   ```
   conda --version
   ```

You're done! It should output the version of conda you just installed

Anaconda vs Miniconda

Miniconda is a lightweight Python distribution with just conda and a few basics, ideal for custom setups. Anaconda is a full package with Python and popular data science libraries pre-installed, great for getting started quickly. Choose Miniconda for flexibility, and Anaconda for convenience.

1. Miniconda:

- **Lightweight version** of Anaconda.
- Comes with **only** `conda`, `pip`, and a few essential packages.
- When you run `conda list` after installing Miniconda, packages like **NumPy will not be present** unless you install it manually using:

  ```
  conda install numpy
  ```

2. Anaconda:

- **Full distribution** with **hundreds of packages** for data science and machine learning.
- Includes **NumPy** and many other popular libraries (like pandas, matplotlib, etc.).
- After installing Anaconda, running `conda list` will **show NumPy** by default.

Summary: Miniconda is minimal and does not come with NumPy, while Anaconda is a larger package that includes it.

Adding conda to Path (Windows)

When you install **Anaconda** or **Miniconda**, the `conda` command-line tool is available in the terminal or shell where you ran the installation script. However, it may not work in other terminals or shells unless you add it to your **environment variables**. To **add `conda` to your environment** (so it works in any terminal or shell), follow these steps based on your operating system:

You can run this command in **Anaconda Prompt** to initialize `conda` for **Command Prompt (cmd)** or **PowerShell**:

```
conda init
```

- This modifies your shell's configuration to recognize `conda`.
- Restart your terminal after running the command.
- The process is automatic and you should be able to use conda from any directory in the terminal.

Manually (If Needed): Add these paths to the **System Environment Variables**:

1. Open:
    1. **Win + R** → Type `sysdm.cpl` to open system properties → Click **Environment Variables**.
2. Find and edit the `Path` variable in **System Variables**.
3. Add these entries:

    ```
    C:\Users\<YourUsername>\Anaconda3\Scripts
    C:\Users\<YourUsername>\Anaconda3\bin
    ```

    Dont forget to replace "<YourUsername>" with your actual username

To verify:

```
conda --version
```

Understanding Conda Workflow

What is Conda?

Conda is a **package manager** and **environment manager** for Python (and other languages). Think of it like a **recipe manager** for your projects!

The Recipe Analogy:

Imagine you're making **mashed potatoes**. The type of **potato** you use changes the taste – in the same way, **package versions** change how your code works.

For example:

- **Python 3.11** is like using a **Shimla Aloo**.
- **Python 3.8** is like using a **Kufri Sindhuri Aloo**.

Conda lets you **control the ingredients** (package versions) for each project.

---

1. Managing Environments with Conda

By default, you are in the **base** environment.

Check Available Environments

Run the following command to list the available environments

```
conda env list
```

2. Create a New Environment

Use this command to create a new conda environment

```
conda create -n myenv
```

Tip: You can create an environment with a specific Python version:

```
conda create -n myenv python=3.11
```

3. Activate an Environment

```
conda activate myenv
```

4. Install Packages (Example: from conda-forge Channel)

```
conda install -c conda-forge numpy
```

- `-c` flag specifies the **channel** (a source for packages).
- **conda-forge** is a popular community-maintained channel with cutting-edge packages.

5. Deactivate the Environment

```
conda deactivate
```

Summary

- **Conda** is a package and environment manager for Python.
- It lets you manage different environments for different projects.
- You can create, activate, and deactivate environments using `conda`.
- Use `conda install` to add packages to an environment.
- **Conda-forge** is a community-maintained channel with many packages.

- **Tip:** Always activate an environment before installing packages.

Anaconda Navigator - Quick Tour

What is Anaconda Navigator

Anaconda Navigator is a graphical user interface (GUI) that comes with the Anaconda distribution. It lets you launch tools like Jupyter Notebook, Spyder, and manage environments and packages without using the command line.

To open Navigator in:

- **Windows:** Search for **Anaconda Navigator** in the Start menu.
- **macOS/Linux:** Run `anaconda-navigator` in the terminal.

It’s beginner-friendly and useful for managing your data science workflow visually.

Jupyter Notebook vs. JupyterLab

Jupyter is an open-source tool for creating and sharing documents that contain **live code**, **visualizations**, **equations**, and **text**.

What is Jupyter Notebook?

- A simple, **interactive environment** to write and run code.
- Each document (called a **notebook**) is saved as a `.ipynb` file.
- Useful for **quick prototyping**, **data exploration**, and **teaching**.
- A **cell** in Jupyter Notebook is a container where you can write and run code, markdown, or raw text.

Start Jupyter Notebook by typing:

```
conda install jupyter
```

followed by:

```
jupyter notebook
```

What is JupyterLab?

JupyterLab is the **next-generation interface** for Jupyter, offering a **more advanced** and **flexible** workspace.

Why We Use JupyterLab:

- **Multi-Panel Interface**: Open notebooks, terminals, text editors, and outputs **side by side**.
- **Better Navigation**: File browsing and workspace management are easier.
- **Extensions**: Customize with themes and advanced plugins.
- **Integrated Tools**: Combine notebooks with terminals and markdown in one window.

Start JupyterLab:

```
conda install jupyterlab
jupyter lab
```

Key Differences:

Feature	Jupyter Notebook	JupyterLab
Interface	Single document view	Multi-tab, multi-panel
Customization	Limited	Extensive via extensions
Performance	Lightweight	Slightly heavier
Use Case	Quick tasks, tutorials	Large projects, workflows

When to Use What:

- **Jupyter Notebook**: Simple analysis, tutorials, and quick checks.
- **JupyterLab**: Complex projects, interactive dashboards, and better organization.

We will use JupyterLab because it provides:

1. **Better Workflow** – Manage multiple notebooks and files in one view.
2. **Scalability** – Ideal for **data science** and **machine learning** projects.
3. **Future-Proofing** – JupyterLab is actively developed and might eventually **replace** Jupyter Notebook (My personal opinion).

As already told you in the video, one of the organizers of the conference I attended in Orlando told me that Jupyter Notebook is a blog where you can execute code. It is an information-rich coding interface. I like this way of looking at it. Hope you are enjoying this course so far!

Python Refresher

Why choose Python

In this section, we will learn why Python is a popular and powerful choice for data science.

Variables, Data Types, and Typecasting

In this section, we will learn about variables, data types, and typecasting in Python to store and convert data effectively. ## Variables

- Containers for storing data values.
- No need to declare data type explicitly.

```
name = "Alice"
age = 25
is_student = True
```

Data Types

Type	Example	Description
`int`	`10` , `-5`	Integer numbers
`float`	`3.14` , `-0.5`	Decimal numbers
`str`	`"hello"`	Text (string)
`bool`	`True` , `False`	Boolean values
`list`	`[1, 2, 3]`	Ordered, mutable collection
`tuple`	`(1, 2, 3)`	Ordered, immutable collection

Type	Example	Description
`dict`	`{"a": 1}`	Key-value pairs

Typecasting (Type Conversion)

- Convert data from one type to another using built-in functions:

```python
# str to int
x = int("10")       # 10

# int to str
y = str(25)         # "25"

# float to int
z = int(3.9)        # 3 (truncates, not rounds)

# list from string
lst = list("abc")  # ['a', 'b', 'c']
```

Quick Tips

- Use `type(variable)` to check a variable's data type.
- Typecasting errors can happen if the value isn't compatible:

  ```python
  int("hello")  # ValueError
  ```

String and String Methods

In this section, we will learn about strings and string methods in Python to work with and manipulate text data.

What is a String?

- A string is a sequence of characters enclosed in single (`'`) or double (`"`) quotes.

```
name = "Alice"
greeting = 'Hello'
```

Multiline Strings

- Use triple quotes (`'''` or `"""`) for multiline text.

```
message = """This is
a multiline
string."""
```

String Indexing and Slicing

- Indexing starts at 0.

```
text = "Python"
text[0]    # 'P'
text[-1]   # 'n' (last character)
text[0:2]  # 'Py'
text[:3]   # 'Pyt'
text[3:]   # 'hon'
```

String Immutability

- Strings cannot be changed after creation.

```
text[0] = 'J'  # Error
```

Common String Methods

Method	Description
`str.lower()`	Converts to lowercase
`str.upper()`	Converts to uppercase
`str.strip()`	Removes leading/trailing spaces
`str.replace(old, new)`	Replaces substring
`str.split(sep)`	Splits string into a list
`str.join(list)`	Joins list into string
`str.find(sub)`	Returns index of first occurrence
`str.count(sub)`	Counts occurrences of substring
`str.startswith(prefix)`	Checks if string starts with value
`str.endswith(suffix)`	Checks if string ends with value
`str.isdigit()`	Checks if all chars are digits
`str.isalpha()`	Checks if all chars are letters
`str.isalnum()`	Checks if all chars are letters/digits

Examples

```python
"hello".upper()            # 'HELLO'
" Hello ".strip()          # 'Hello'
"hello world".split()      # ['hello', 'world']
"-".join(["2025", "04", "14"])  # '2025-04-14'
"python".find("th")        # 2
```

String Formatting (f-strings)

```
name = "Alice"
age = 30
f"Hello, {name}. You are {age} years old."
# 'Hello, Alice. You are 30 years old.'
```

Operators in Python

In this section, we will learn about different types of operators in Python and how they are used in expressions.
## 1. Arithmetic Operators Used for basic mathematical operations.

Operator	Description	Example (`a=10, b=5`)
`+`	Addition	`a + b # 15`
`-`	Subtraction	`a - b # 5`
`*`	Multiplication	`a * b # 50`
`/`	Division	`a / b # 2.0`
`//`	Floor Division	`a // b # 2`
`%`	Modulus	`a % b # 0`
`**`	Exponentiation	`a ** b # 100000`

2. Comparison Operators

Compare values and return `True` or `False`.

Operator	Description	Example (`a=10, b=5`)
`==`	Equal to	`a == b # False`
`!=`	Not equal to	`a != b # True`

Operator	Description	Example (`a=10, b=5`)
`>`	Greater than	`a > b # True`
`<`	Less than	`a < b # False`
`>=`	Greater or equal	`a >= b # True`
`<=`	Less or equal	`a <= b # False`

3. Logical Operators

Used to combine conditional statements.

Operator	Description	Example (`x=True, y=False`)
`and`	Both True	`x and y # False`
`or`	Either True	`x or y # True`
`not`	Negation	`not x # False`

4. Bitwise Operators

Perform bit-level operations.

Operator	Description	Example (`a=5, b=3`)
`&`	AND	`a & b # 1`
\\|	OR	a \\| b # 7

5. Assignment Operators

Used to assign values to variables.

Operator	Example (`a=10`)	Equivalent to
`=`	`a = 5`	`a = 5`

Operator	Example (`a=10`)	Equivalent to
`+=`	`a += 5`	`a = a + 5`
`-=`	`a -= 5`	`a = a - 5`
`*=`	`a *= 5`	`a = a * 5`
`/=`	`a /= 5`	`a = a / 5`
`//=`	`a //= 5`	`a = a // 5`
`%=`	`a %= 5`	`a = a % 5`
`**=`	`a **= 5`	`a = a ** 5`

6. Membership & Identity Operators

Check for presence and object identity.

Operator	Description	Example (`lst=[1,2,3]` , `x=2`)
`in`	Present in sequence	`x in lst # True`
`not in`	Not present	`x not in lst # False`
`is`	Same object	`a is b # False`
`is not`	Different object	`a is not b # True`

Taking input from the user

In this section, we will learn how to take input from the user in Python and use it in our programs. ## Basic Usage

```
name = input("Enter your name: ")
print("Hello", name)
```

Note: `input()` always returns a string.

Type Conversion

You can always convert the string output of input() function to other supported data types

```
age = int(input("Enter your age: "))
price = float(input("Enter the price: "))
```

Operator Precedence

Python follows **PEMDAS** (Parentheses, Exponents, Multiplication/Division, Addition/Subtraction). The order of operations in Python is:

1. **Parentheses** `()` – Highest precedence, operations inside parentheses are evaluated first.
2. **Exponents** `**` – Power calculations (e.g., `2 ** 3` → 8).
3. **Multiplication** `*`, **Division** `/`, **Floor Division** `//`, **Modulus** `%` – Evaluated from left to right.
4. **Addition** `+`, **Subtraction** `-` – Evaluated from left to right.

Example:

```
result = 10 + 2 * 3  # Multiplication happens first: 10 + (2 * 3)
print(result)

result = (10 + 2) * 3  # Parentheses first: (10 + 2) * 3 = 36
print(result)

result = 2 ** 3 ** 2  # Right-to-left exponentiation: 2 ** (3 **
print(result)
```

If else statements

In Python, conditional statements (`if`, `elif`, and `else`) are used to control the flow of a program based on conditions. These are essential in data science for handling different scenarios in data processing, decision-making, and logic execution.

Basic `if` Statement

The `if` statement allows you to execute a block of code only if a condition is `True`.

```python
x = 10
if x > 5:
    print("x is greater than 5")
```

Explanation:

- The condition `x > 5` is checked.
- If `True`, the indented block under `if` runs.
- If `False`, nothing happens.

`if-else` Statement

The `else` block executes when the `if` condition is `False`.

```python
x = 3
if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")
```

Explanation:

- If `x > 5`, it prints the first message.
- Otherwise, the `else` block executes.

`if-elif-else` Statement

When multiple conditions need to be checked sequentially, use `elif` (short for "else if").

```
x = 5
if x > 10:
    print("x is greater than 10")
elif x > 5:
    print("x is greater than 5 but not more than 10")
elif x == 5:
    print("x is exactly 5")
else:
    print("x is less than 5")
```

Explanation:

- The conditions are checked from top to bottom.
- The first `True` condition executes, and the rest are skipped.

Using `if-else` in Data Science

Conditional statements are widely used in data science for filtering, cleaning, and decision-making.

Example: Categorizing Data

```
age = 25
if age < 18:
    category = "Minor"
elif age < 65:
    category = "Adult"
else:
    category = "Senior Citizen"

print("Category:", category)
```

Example: Applying Conditions on Pandas DataFrame

```
import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie'], 'Score': [85, 40, 75
df = pd.DataFrame(data)

df['Result'] = df['Score'].apply(lambda x: 'Pass' if x >= 50 else
print(df)
```

Summary

- `if` : Executes if the condition is `True` .
- `if-else` : Adds an alternative block if the condition is `False` .
- `if-elif-else` : Handles multiple conditions.
- Useful in data science for logic-based decision-making.

Match Case Statements

The `match-case` statement, introduced in Python 3.10, provides pattern matching similar to `switch` statements in other languages.

Syntax

```
def http_status(code):
    match code:
        case 200:
            return "OK"
        case 400:
            return "Bad Request"
        case 404:
            return "Not Found"
        case 500:
            return "Internal Server Error"
        case _:
            return "Unknown Status"

print(http_status(200))  # Output: OK
print(http_status(404))  # Output: Not Found
```

Features:

- The `_` (underscore) acts as a default case.
- Patterns can include literals, variable bindings, and even structural patterns.

Example: Matching Data Structures

Lets try to match a tuple using Match-Case statements

```
point = (3, 4)

match point:
    case (0, 0):
        print("Origin")
    case (x, 0):
        print(f"X-Axis at {x}")
    case (0, y):
        print(f"Y-Axis at {y}")
    case (x, y):
        print(f"Point at ({x}, {y})")
```

String Formatting and F-Strings

String is arguably the most used immutable data types in Python. Python provides multiple ways to format strings, including the `format()` method and f-strings (introduced in Python 3.6).

1. Using `format()`

```
name = "Alice"
age = 25
print("My name is {} and I am {} years old.".format(name, age))
# Output: My name is Alice and I am 25 years old.
```

Positional and Keyword Arguments

```
print("{0} is learning {1}".format("Alice", "Python"))  # Using p
print("{name} is learning {language}".format(name="Alice", langua
```

2. Using f-Strings (Recommended)

F-strings provide a cleaner and more readable way to format strings.

```
name = "Alice"
age = 25
print(f"My name is {name} and I am {age} years old.")
# Output: My name is Alice and I am 25 years old.
```

Expressions Inside f-Strings

```
a = 5
b = 10
print(f"Sum of {a} and {b} is {a + b}")
# Output: Sum of 5 and 10 is 15
```

Formatting Numbers

```
pi = 3.14159
print(f"Pi rounded to 2 decimal places: {pi:.2f}")
# Output: Pi rounded to 2 decimal places: 3.14
```

Padding and Alignment

```python
print(f"{'Python':<10}")  # Left-align
print(f"{'Python':>10}")  # Right-align
print(f"{'Python':^10}")  # Center-align
```

:<10 → The < symbol means left-align the text within a total width of 10 characters.

F-strings are the most efficient and recommended way to format strings in modern Python!

Loops in Python

Python has two main loops: `for` and `while`.

1. For Loop

Used to iterate over sequences like lists, tuples, and strings.

```python
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
    print(fruit)
```

Using `range()`

```python
for i in range(3):
    print(i)  # Output: 0, 1, 2
```

2. While Loop

Runs as long as a condition is `True`.

```python
count = 0
while count < 3:
    print(count)
    count += 1
```

Output:

```
0
1
2
```

3. Loop Control Statements

- `break` → Exits the loop.
- `continue` → Skips to the next iteration.
- `pass` → Does nothing (used as a placeholder).

```
for i in range(5):
    if i == 3:
        break  # Stops the loop at 3
    print(i)
```

List and List Methods

A **list** in Python is an ordered, mutable collection of elements. It can contain elements of different types.

Creating a List:

```
# Empty list
my_list = []

# List with elements
numbers = [1, 2, 3, 4, 5]
```

```python
# Mixed data types
mixed_list = [1, "Hello", 3.14, True]
```

Common List Methods

Method	Description	Example
`append(x)`	Adds an element `x` to the end of the list.	`my_list.append(10)`
`extend(iterable)`	Extends the list by appending all elements from an iterable.	`my_list.extend([6, 7, 8])`
`insert(index, x)`	Inserts `x` at the specified `index`.	`my_list.insert(2, "Python")`
`remove(x)`	Removes the first occurrence of `x` in the list.	`my_list.remove(3)`
`pop([index])`	Removes and returns the element at `index` (last element if index is not provided).	`my_list.pop(2)`
`index(x)`	Returns the index of the first occurrence of `x`.	`my_list.index(4)`
`count(x)`	Returns the number of times `x` appears in the list.	`my_list.count(2)`
`sort()`	Sorts the list in ascending order.	`my_list.sort()`
`reverse()`	Reverses the order of the list.	`my_list.reverse()`
`copy()`	Returns a shallow copy of the list.	`new_list = my_list.copy()`
`clear()`	Removes all elements from the list.	`my_list.clear()`

Example Usage:

```
fruits = ["apple", "banana", "cherry"]
fruits.append("orange")
print(fruits)  # ['apple', 'banana', 'cherry', 'orange']

fruits.sort()
print(fruits)  # ['apple', 'banana', 'cherry', 'orange']
```

Tuples and Tuple Methods

A **tuple** in Python is an ordered, immutable collection of elements. It is similar to a list, but once created, its elements cannot be modified.

Creating a Tuple:

```
# Empty tuple
empty_tuple = ()

# Tuple with elements
numbers = (1, 2, 3, 4, 5)

# Mixed data types
mixed_tuple = (1, "Hello", 3.14, True)

# Single element tuple (comma is necessary)
single_element = (42,)
```

Common Tuple Methods

Method	Description	Example
`count(x)`	Returns the number of times `x` appears in the tuple.	`my_tuple.count(2)`
`index(x)`		`my_tuple.index(3)`

Method	Description	Example
	Returns the index of the first occurrence of `x`.	

Tuple Characteristics

- **Immutable**: Once created, elements cannot be changed.
- **Faster than lists**: Accessing elements in a tuple is faster than in a list.
- **Can be used as dictionary keys**: Since tuples are immutable, they can be used as keys in dictionaries.

Accessing Tuple Elements

```
my_tuple = (10, 20, 30, 40)

# Indexing
print(my_tuple[1])  # 20

# Slicing
print(my_tuple[1:3])  # (20, 30)
```

Tuple Packing and Unpacking

```
# Packing
person = ("Alice", 25, "Engineer")

# Unpacking
name, age, profession = person
print(name)  # Alice
print(age)   # 25
```

When to Use Tuples?

- When you want an **unchangeable** collection of elements.
- When you need a **faster** alternative to lists.
- When storing **heterogeneous data** (e.g., database records, coordinates).

Set and Set Methods

A **set** in Python is an **unordered**, **mutable**, and **unique** collection of elements. It does not allow duplicate values.

Creating a Set:

```
# Empty set (must use set(), not {})
empty_set = set()

# Set with elements
numbers = {1, 2, 3, 4, 5}

# Mixed data types
mixed_set = {1, "Hello", 3.14, True}

# Creating a set from a list
unique_numbers = set([1, 2, 2, 3, 4, 4, 5])
print(unique_numbers)  # {1, 2, 3, 4, 5}
```

Common Set Methods

Method	Description	Example
`add(x)`	Adds an element `x` to the set.	`my_set.add(10)`
`update(iterable)`	Adds multiple elements from an iterable.	`my_set.update([6, 7, 8])`
`remove(x)`	Removes `x` from the set (raises an error if not found).	`my_set.remove(3)`

Method	Description	Example
`discard(x)`	Removes `x` from the set (does not raise an error if not found).	`my_set.discard(3)`
`pop()`	Removes and returns a random element.	`my_set.pop()`
`clear()`	Removes all elements from the set.	`my_set.clear()`
`copy()`	Returns a shallow copy of the set.	`new_set = my_set.copy()`

Set Operations

Operation	Description	Example
`union(set2)`	Returns a new set with all unique elements from both sets.	`set1.union(set2)`
`intersection(set2)`	Returns a set with elements common to both sets.	`set1.intersection(set2)`
`difference(set2)`	Returns a set with elements in `set1` but not in `set2`.	`set1.difference(set2)`
`symmetric_difference(set2)`		`set1.symmetric_difference(set2)`

Operation	Description	Example
	Returns a set with elements in either `set1` or `set2`, but not both.	
`issubset(set2)`	Returns `True` if `set1` is a subset of `set2`.	`set1.issubset(set2)`
`issuperset(set2)`	Returns `True` if `set1` is a superset of `set2`.	`set1.issuperset(set2)`

Example Usage:

In Python, sets support intuitive operators for common operations like union (`|`), intersection (`&`), difference (`-`), and symmetric difference (`^`). These have equivalent method forms too, like `.union()`, `.intersection()`, etc. Here's a quick example:

```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}

# Union – combines all unique elements
print(set1 | set2)            # {1, 2, 3, 4, 5, 6}
print(set1.union(set2))       # same result

# Intersection – common elements
print(set1 & set2)            # {3, 4}
```

```python
print(set1.intersection(set2))# same result

# Difference - in set1 but not in set2
print(set1 - set2)           # {1, 2}
print(set1.difference(set2))  # same result

# Symmetric Difference - in either set, but not both
print(set1 ^ set2)                    # {1, 2, 5, 6}
print(set1.symmetric_difference(set2))# same result
```

Key Properties of Sets:

- **Unordered**: No indexing or slicing.
- **Unique Elements**: Duplicates are automatically removed.
- **Mutable**: You can add or remove elements.

Dictionary and Dictionary Methods

A **dictionary** in Python is an **unordered**, **mutable**, and **key-value** pair collection. It allows efficient data retrieval and modification. Dictionaries in Python are ordered as of Python 3.7

Creating a Dictionary:

```python
# Empty dictionary
empty_dict = {}

# Dictionary with key-value pairs
student = {
    "name": "Alice",
    "age": 25,
    "grade": "A"
}

# Using dict() constructor
person = dict(name="John", age=30, city="New York")
```

Accessing Dictionary Elements

```
# Using keys
print(student["name"])  # Alice

# Using get() (avoids KeyError if key doesn't exist)
print(student.get("age"))  # 25
print(student.get("height", "Not Found"))  # Default value
```

Common Dictionary Methods

Method	Description	Example
`keys()`	Returns all keys in the dictionary.	`student.keys()`
`values()`	Returns all values in the dictionary.	`student.values()`
`items()`	Returns key-value pairs as tuples.	`student.items()`
`get(key, default)`	Returns value for `key`, or `default` if key not found.	`student.get("age", 0)`
`update(dict2)`	Merges `dict2` into the dictionary.	`student.update({"age": 26})`
`pop(key, default)`	Removes key and returns its value (or `default` if key not found).	`student.pop("grade")`
`popitem()`	Removes and returns the last inserted key-value pair.	`student.popitem()`

Method	Description	Example
`setdefault(key, default)`	Returns value for `key`, else sets it to `default`.	`student.setdefault("city", "Unknown")`
`clear()`	Removes all items from the dictionary.	`student.clear()`
`copy()`	Returns a shallow copy of the dictionary.	`new_dict = student.copy()`

Example Usage:

```
student = {"name": "Alice", "age": 25, "grade": "A"}

# Adding a new key-value pair
student["city"] = "New York"

# Updating an existing value
student["age"] = 26

# Removing an item
student.pop("grade")

# Iterating over a dictionary
for key, value in student.items():
    print(key, ":", value)

# Output:
# name : Alice
# age : 26
# city : New York
```

Dictionary Comprehension:

```
# Creating a dictionary using comprehension
squares = {x: x**2 for x in range(1, 6)}
print(squares)  # {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Key Properties of Dictionaries:

- **Unordered** (Python 3.6+ maintains insertion order).
- **Keys must be unique and immutable** (e.g., strings, numbers, tuples).
- **Values can be mutable** and of any type.

File Handling in Python

File handling allows Python programs to **read, write, and manipulate files** stored on disk. Python provides built-in functions for working with files.

Opening a File

Python uses the `open()` function to open a file.

Syntax

```
file = open("filename", mode)
```

- **`filename`** → The name of the file to open.
- **`mode`** → Specifies how the file should be opened.

File Modes

Mode	Description
`'r'`	Read (default) – Opens file for reading, **raises an error if file does not exist**.

Mode	Description
`'w'`	Write – Opens file for writing, **creates a new file if not found**, and **overwrites existing content**.
`'a'`	Append – Opens file for writing, **creates a new file if not found**, and appends content instead of overwriting.
`'x'`	Create – Creates a new file, but **fails if the file already exists**.
`'b'`	Binary mode – Used with `rb`, `wb`, `ab`, etc., for working with non-text files (e.g., images, PDFs).
`'t'`	Text mode (default) – Used for text files (e.g., `rt`, `wt`).

Reading Files

Using `read()` – Read Entire File

```
file = open("example.txt", "r")
content = file.read()
print(content)
file.close()  # Always close the file after use
```

Using `readline()` – Read Line by Line

```
file = open("example.txt", "r")
line1 = file.readline()  # Reads first line
print(line1)
file.close()
```

Using `readlines()` – Read All Lines as List

```
file = open("example.txt", "r")
lines = file.readlines()  # Reads all lines into a list
print(lines)
file.close()
```

Writing to Files

Using `write()` – Overwrites Existing Content

```
file = open("example.txt", "w")  # Opens file in write mode
file.write("Hello, World!")  # Writes content
file.close()
```

Using `writelines()` – Write Multiple Lines

```
lines = ["Hello\n", "Welcome to Python\n", "File Handling\n"]

file = open("example.txt", "w")
file.writelines(lines)  # Writes multiple lines
file.close()
```

Appending to a File

The `a` **(append) mode** is used to add content to an existing file without erasing previous data.

```
file = open("example.txt", "a")
file.write("\nThis is an additional line.")
file.close()
```

Using `with` Statement (Best Practice)

Using `with open()` ensures the file is **automatically closed** after execution.

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)  # No need to manually close the file
```

Checking if a File Exists

Use the `os` module to check if a file exists before opening it.

```
import os

if os.path.exists("example.txt"):
    print("File exists!")
else:
    print("File not found!")
```

Deleting a File

Use the `os` module to delete a file.

```
import os

if os.path.exists("example.txt"):
    os.remove("example.txt")
    print("File deleted.")
else:
    print("File does not exist.")
```

Working with Binary Files

Binary files (`.jpg` , `.png` , `.pdf` , etc.) should be opened in **binary mode** (`'b'`).

Reading a Binary File

```
with open("image.jpg", "rb") as file:
    data = file.read()
    print(data)  # Outputs binary content
```

Writing to a Binary File

```
with open("new_image.jpg", "wb") as file:
    file.write(data)  # Writes binary content to a new file
```

Summary of File Operations

Operation	Description	Example
Open file	Open a file	`file = open("example.txt", "r")`
Read file	Read all content	`file.read()`
Read line	Read one line	`file.readline()`
Read lines	Read all lines into list	`file.readlines()`
Write file	Write content (overwrite)	`file.write("Hello")`
Append file	Add content to the end	`file.write("\nMore text")`
		`os.path.exists("file.txt")`

Operation	Description	Example
Check file existence	Check before opening/deleting	
Delete file	Remove a file	`os.remove("file.txt")`

JSON module in Python

JSON (**JavaScript Object Notation**) is a lightweight data format used for data exchange between servers and applications. It is widely used in **APIs, web applications, and configurations**.

Python provides the `json` module to work with JSON data. You can import the json module like this:

```
import json
```

Converting Python Objects to JSON (Serialization)

Serialization (also called **encoding or dumping**) is converting a Python object into a JSON-formatted string.

`json.dumps()` – Convert Python object to JSON string

```
import json

data = {"name": "Alice", "age": 25, "city": "New York"}

json_string = json.dumps(data)
print(json_string)  # Output: {"name": "Alice", "age": 25, "city"
print(type(json_string))  # <class 'str'>
```

`json.dump()` – Write JSON data to a file

```
with open("data.json", "w") as file:
    json.dump(data, file)
```

Converting JSON to Python Objects (Deserialization)

Deserialization (also called **decoding or loading**) is converting JSON-formatted data into Python objects.

`json.loads()` – Convert JSON string to Python object

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

python_obj = json.loads(json_data)
print(python_obj)  # Output: {'name': 'Alice', 'age': 25, 'city':
print(type(python_obj))  # <class 'dict'>
```

`json.load()` – Read JSON data from a file

```
with open("data.json", "r") as file:
    python_data = json.load(file)

print(python_data)  # Output: {'name': 'Alice', 'age': 25, 'city'
```

Formatting JSON Output

You can format JSON for better readability using **indentation**.

```
formatted_json = json.dumps(data, indent=4)
print(formatted_json)
```

Output:

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

Summary of Common JSON Methods

Method	Description	Example
`json.dumps(obj)`	Converts Python object to JSON string	`json.dumps(data)`
`json.dump(obj, file)`	Writes JSON to a file	`json.dump(data, file)`
`json.loads(json_string)`	Converts JSON string to Python object	`json.loads(json_data)`
`json.load(file)`	Reads JSON from a file	`json.load(file)`

Object Oriented Programming in Python

Object-Oriented Programming (OOP) is a **programming paradigm** that organizes code into objects that contain both **data (attributes)** and **behavior (methods)**.

Key Concepts of OOP

Concept	Description
Class	A blueprint for creating objects.

Concept	Description
Object	An instance of a class with specific data and behavior.
Attributes	Variables that store data for an object.
Methods	Functions inside a class that define object behavior.
Encapsulation	Restricting direct access to an object's data.
Inheritance	Creating a new class from an existing class.
Polymorphism	Using the same method name for different classes.

1. Defining a Class and Creating an Object

Creating a Class

```python
class Car:
    def __init__(self, brand, model):
        self.brand = brand  # Attribute
        self.model = model  # Attribute

    def display_info(self):  # Method
        return f"{self.brand} {self.model}"

# Creating an Object (Instance)
car1 = Car("Toyota", "Camry")
print(car1.display_info())  # Output: Toyota Camry
```

2. Encapsulation (Data Hiding)

Encapsulation prevents direct modification of attributes and allows controlled access using **getter and setter methods**.

```python
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private Attribute

    def get_balance(self):  # Getter
        return self.__balance

    def deposit(self, amount):  # Setter
        if amount > 0:
            self.__balance += amount

# Using Encapsulation
account = BankAccount(1000)
account.deposit(500)
print(account.get_balance())  # Output: 1500
```

◇ **Why use encapsulation?**

It protects data by restricting direct modification.

---

3. Inheritance (Reusing Code)

Inheritance allows a class (**child**) to inherit attributes and methods from another class (**parent**).

Example of Single Inheritance

```python
class Animal:
    def speak(self):
        return "Animal makes a sound"

class Dog(Animal):  # Inheriting from Animal
    def speak(self):
        return "Bark"

dog = Dog()
print(dog.speak())  # Output: Bark
```

◇ **Why use inheritance?**
It promotes **code reusability** and maintains a cleaner code structure.

---

4. Multiple Inheritance

A class can inherit from multiple parent classes.

```
class A:
    def method_a(self):
        return "Method A"

class B:
    def method_b(self):
        return "Method B"

class C(A, B):  # Multiple Inheritance
    pass

obj = C()
print(obj.method_a())  # Output: Method A
print(obj.method_b())  # Output: Method B
```

◇ **Why use multiple inheritance?**
It allows a class to inherit **features from multiple parent classes**.

---

5. Polymorphism (Same Method, Different Behavior)

Polymorphism allows different classes to use the **same method name**.

Method Overriding Example

```
class Bird:
    def fly(self):
        return "Birds can fly"
```

```python
class Penguin(Bird):
    def fly(self):
        return "Penguins cannot fly"

bird = Bird()
penguin = Penguin()

print(bird.fly())       # Output: Birds can fly
print(penguin.fly())    # Output: Penguins cannot fly
```

◇ **Why use polymorphism?**
It provides **flexibility** by allowing different classes to define the same method differently.

---

6. Abstraction (Hiding Implementation Details)

Abstraction is used to define a method **without implementing it** in the base class. It is achieved using **abstract base classes** (`ABC` module).

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass  # No implementation

class Square(Shape):
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side * self.side  # Implemented in child clas

square = Square(4)
print(square.area())  # Output: 16
```

🔹 **Why use abstraction?**

It enforces **consistent implementation** across child classes.

---

7. Magic Methods (Dunder Methods)

Magic methods allow objects to behave like **built-in types**.

Example: `__str__()` and `__len__()`

Have a look at the code below:

```python
class Book:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages

    def __str__(self):  # String representation
        return f"Book: {self.title}"

    def __len__(self):  # Define behavior for len()
        return self.pages

book = Book("Python Basics", 300)
print(str(book))  # Output: Book: Python Basics
print(len(book))  # Output: 300
```

---

8. Class vs. Static Methods

Method Type	Description	Uses `self` ?	Uses `cls` ?
Instance Method	Works with instance attributes	✅	❌

Method Type	Description	Uses `self`?	Uses `cls`?
Class Method	Works with class attributes	✖	☑
Static Method	Does not use class or instance variables	✖	✖

Example

```python
class Example:
    class_var = "I am a class variable"

    def instance_method(self):
        return "Instance Method"

    @classmethod
    def class_method(cls):
        return cls.class_var

    @staticmethod
    def static_method():
        return "Static Method"

obj = Example()
print(obj.instance_method())  # Output: Instance Method
print(Example.class_method()) # Output: I am a class variable
print(Example.static_method()) # Output: Static Method
```

Summary of OOP Concepts

Concept	Description	Example
Class	A blueprint for creating objects	`class Car:`

Concept	Description	Example
Object	An instance of a class	`car1 = Car()`
Encapsulation	Restrict direct access to data	`self.__balance`
Inheritance	A class inherits from another class	`class Dog(Animal)`
Polymorphism	Using the same method in different ways	`def fly(self)`
Abstraction	Hiding implementation details	`@abstractmethod`
Magic Methods	Special methods like `__str__()`	`def __len__(self)`
Class Methods	Works with class variables	`@classmethod`
Static Methods	Independent of class and instance	`@staticmethod`

List Comprehension

List comprehension is a **concise** and **efficient** way to create lists in Python. It allows you to generate lists in a **single line of code**, making your code more readable and Pythonic.

---

1. Basic Syntax

```
[expression for item in iterable]
```

- `expression` → The operation to perform on each item
- `item` → The variable representing each element in the iterable
- `iterable` → The data structure being iterated over (list, range, etc.)

Example: Creating a list of squares

```python
squares = [x**2 for x in range(5)]
print(squares)  # Output: [0, 1, 4, 9, 16]
```

---

2. Using if Condition in List Comprehension

Example: Filtering even numbers

```python
evens = [x for x in range(10) if x % 2 == 0]
print(evens)  # Output: [0, 2, 4, 6, 8]
```

---

3. Using if-else Condition in List Comprehension

Example: Replacing even numbers with “Even” and odd numbers with “Odd”

```python
numbers = ["Even" if x % 2 == 0 else "Odd" for x in range(5)]
print(numbers)  # Output: ['Even', 'Odd', 'Even', 'Odd', 'Even']
```

---

4. Nested Loops in List Comprehension

Example: Creating pairs from two lists

```python
pairs = [(x, y) for x in range(2) for y in range(3)]
print(pairs)  # Output: [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1),
```

---

5. List Comprehension with Functions

Example: Converting a list of strings to uppercase

```
words = ["hello", "world", "python"]
upper_words = [word.upper() for word in words]
print(upper_words)  # Output: ['HELLO', 'WORLD', 'PYTHON']
```

6. List Comprehension with Nested List Comprehension

Example: Flattening a 2D list

```
matrix = [[1, 2], [3, 4], [5, 6]]
flattened = [num for row in matrix for num in row]
print(flattened)  # Output: [1, 2, 3, 4, 5, 6]
```

7. List Comprehension with Set and Dictionary Comprehensions

Set Comprehension

```
unique_numbers = {x for x in [1, 2, 2, 3, 4, 4]}
print(unique_numbers)  # Output: {1, 2, 3, 4}
```

Dictionary Comprehension

```
squared_dict = {x: x**2 for x in range(5)}
print(squared_dict)  # Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

8. When to Use List Comprehensions?

- You need to create a list in a **single line**
- The logic is **simple and readable**
- You want to improve **performance** (faster than loops)

Avoid when: - The logic is **too complex** (use a standard loop instead for clarity)

9. Performance Comparison: List Comprehension vs. Loop

```python
import time

# Using a for loop
start = time.time()
squares_loop = []
for x in range(10**6):
    squares_loop.append(x**2)
print("Loop time:", time.time() - start)

# Using list comprehension
start = time.time()
squares_comp = [x**2 for x in range(10**6)]
print("List Comprehension time:", time.time() - start)
```

List comprehensions are generally faster than loops because they are optimized internally by Python.

Summary

Concept	Example
Basic List Comprehension	`[x**2 for x in range(5)]`
With Condition (`if`)	`[x for x in range(10) if x % 2 == 0]`
With `if-else`	`["Even" if x % 2 == 0 else "Odd" for x in range(5)]`
Nested Loop	`[(x, y) for x in range(2) for y in range(3)]`
Flatten 2D List	`[num for row in matrix for num in row]`
Set Comprehension	`{x for x in [1, 2, 2, 3]}`
Dictionary Comprehension	`{x: x**2 for x in range(5)}`

Lambda Functions

A **lambda function** in Python is an **anonymous, single-expression function** defined using the `lambda` keyword. It is commonly used for **short, throwaway functions** where a full function definition is unnecessary.

---

1. Syntax of Lambda Functions

```
lambda arguments: expression
```

- `lambda` → Keyword to define a lambda function
- `arguments` → Input parameters (comma-separated)
- `expression` → The operation performed (must be a **single** expression, not multiple statements)

Example: Simple Lambda Function

```python
square = lambda x: x ** 2
print(square(5))  # Output: 25
```

---

2. Using Lambda Functions with `map()`, `filter()`, and `reduce()`

2.1 Using `map()` with Lambda

Applies a function to each element of an iterable.

```python
numbers = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, numbers))
print(squared)  # Output: [1, 4, 9, 16]
```

2.2 Using `filter()` with Lambda

Filters elements based on a condition.

```python
numbers = [1, 2, 3, 4, 5, 6]
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(evens)  # Output: [2, 4, 6]
```

2.3 Using `reduce()` with Lambda

Reduces an iterable to a single value (requires `functools.reduce`).

```python
from functools import reduce
```

```
numbers = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, numbers)
print(product)  # Output: 24
```

3. Lambda with Multiple Arguments

Example: Adding Two Numbers

```
add = lambda x, y: x + y
print(add(3, 7))  # Output: 10
```

Example: Finding the Maximum of Two Numbers

```
maximum = lambda x, y: x if x > y else y
print(maximum(10, 5))  # Output: 10
```

4. Lambda in Sorting Functions

Sorting a List of Tuples

```
students = [("Alice", 85), ("Bob", 78), ("Charlie", 92)]
students.sort(key=lambda student: student[1])  # Sort by score
print(students)  # Output: [('Bob', 78), ('Alice', 85), ('Charlie
```

5. When to Use Lambda Functions?

Use Lambda Functions When:

- The function is **short and simple**.

- Used **temporarily** inside another function (e.g., `map` , `filter`).
- Avoiding defining a full function with `def` .

Avoid Lambda Functions When:

- The function is **complex** (use `def` for readability).
- Multiple operations/statements are needed.

Summary

Feature	Example
Basic Lambda Function	`lambda x: x**2`
With `map()`	`map(lambda x: x**2, numbers)`
With `filter()`	`filter(lambda x: x % 2 == 0, numbers)`
With `reduce()`	`reduce(lambda x, y: x * y, numbers)`
Multiple Arguments	`lambda x, y: x + y`
Sorting with Lambda	`sort(key=lambda x: x[1])`