Python for Data Science

Course 04 || IBM DS PRO

About The Course

Course Introduction

Course Overview

- Python is a versatile and powerful language for data science and AI.
- The course is beginner-friendly, but experienced programmers can also benefit from it.

Python Use Cases

- Data analysis
- Web scraping
- Big data processing
- Finance
- Computer vision
- Natural language processing (NLP)
- Machine learning and deep learning

Course Overview

Course Content

This course is divided into five modules. You should set a goal to complete at least one module per week.

Module 1: Python Basics

- About the Course
- Types
- Expressions and Variables
- String Operations

Module 2: Python Data Structures

- Lists and Tuples
- Dictionaries
- Sets

Module 3: Python Programming Fundamentals

- Conditions and Branching
- Loops
- Functions
- Exception Handling
- Objects and Classes
- Practice with Python Programming Fundamentals

Module 4: Working with Data in Python

- Reading and Writing Files with Open
- Pandas
- Numpy in Python

Module 5: APIs and Data Collection

- Simple APIs
- REST APIs, Web Scraping, and Working with Files
- Final Exam

Python Basics

Module 01 || Course 04
IBM DS PRO

Lesson 01 || GS with Python and Jupyter

01-Intro to Python

Overview of Python

- Python is a widely used language, especially in data science.
- In 2019, 75% of data science job listings required Python.
- It is recommended as the first language to learn due to its simplicity.

Users of Python

- **Experienced Programmers**: Python's clear syntax allows for more efficient coding.
- **Beginners**: It's great for beginners due to its large community and abundant resources.
- **Data Professionals**: 80%+ of data professionals use Python, especially in data science, AI, and IoT.

Key Benefits of Python

- **Clear Syntax**: Easy to read and write.
- **Versatile**: Used in multiple fields like data science, machine learning, web development, and IoT.
- **Industry Adoption**: Used by major companies like IBM, Google, and NASA.
- **Powerful Libraries**:
    - **Data Science**: Pandas, NumPy, Matplotlib.
    - **Machine Learning**: TensorFlow, PyTorch, Scikit-learn.

Python in Data Science

- Libraries like Pandas and NumPy are perfect for data analysis and manipulation.
- Python also supports machine learning (Scikit-learn, TensorFlow) and NLP (NLTK).

Summary

- Python is simple, widely adopted, and backed by a diverse community.
- Its strong library support makes it ideal for data science, AI, and more.

02-Introduction to Jupyter

Introduction to Jupyter

Jupyter is a free, interactive web application that allows users to create and share documents with code, visualizations, and text. It supports multiple languages, with Python being the most popular.

Why Use Jupyter?

- **Intuitive and flexible** for both beginners and experienced coders.
- **Interactive environment** for data exploration, testing, and visualizing results.

Key Features

- **Interactive Computing**: Execute code in cells and see immediate results.
- **Multiple Languages**: Supports Python, R, Julia, and others.
- **Rich Output**: Create plots, charts, and visuals within notebooks.
- **Library Integration**: Works well with libraries like NumPy, Pandas, and TensorFlow.
- **Collaboration**: Share notebooks via email, GitHub, or Jupyter Viewer.

Jupyter in Data Science
Jupyter is essential for data analysis, machine learning, and research, offering a user-friendly, interactive platform.

03-Getting Started with Jupyter

In this video, you will learn how to work with Jupyter notebooks, including running, inserting, deleting cells, and managing multiple notebooks.

Key Steps

1. **Running a Notebook**
    - After opening a notebook, click the **Run** button or press **Shift + Enter** to execute code.
    - You can run selected cells or all cells using the dropdown menu or the **Run All Cells** option.
2. **Inserting and Deleting Cells**
    - To insert a new cell, click the **plus symbol** in the toolbar.
    - To delete a cell, highlight it, go to **Edit > Delete Cells**, or press **D** twice on the selected cell.
    - You can also move cells up or down.
3. **Working with Multiple Notebooks**
   Open a new notebook by clicking the **plus button** or using the **File** menu.
    - Arrange multiple notebooks side by side for easy comparison.
4. **Presenting Notebooks**
    - Use **Markdown** to add titles and text descriptions to your notebook.
    - Convert cells into slides for presentation, displaying code, results, and visualizations.
5. **Shutting Down a Notebook**
   To release memory, click the **stop icon** in the sidebar.
    - You can shut down all sessions or individual notebooks. After shutting down, the notebook will show "no kernel" at the top right.

Summary
You've learned how to run, insert, and delete cells, manage multiple notebooks, create presentations, and shut down notebooks once finished.

04-Hands on Lab || Jupyter

Click here to open the notebook and Practice

Lesson 02 || Data Types

01-Python Data Types

Python represents different types of data using various built-in types.

Common Data Types

- **Integers (`int`)**: Whole numbers (e.g., `11`, `-5`)
- **Floats (`float`)**: Real numbers, including decimals (e.g., `21.213`, `0.5`)
- **Strings (`str`)**: A sequence of characters (e.g., `"Hello"`)

Viewing Data Types

Use the `type()` function to check the data type of a value.

```
print(type(11))      # Output: <class 'int'>

print(type(21.213))  # Output: <class 'float'>

print(type("Hello")) # Output: <class 'str'>
```

Integer and Float Representation

- Integers can be positive or negative.
- Floats include all integers and numbers between them.
- The precision of floats is limited but allows zooming in on small differences.

Typecasting (Converting Data Types)

- Convert an `int` to a `float`:
  ```
  float(2)  # Output: 2.0
  ```

- Convert a `float` to an `int` (losing decimal information):
  ```
  int(1.9)  # Output: 1
  ```
- Convert a numeric string to an integer (only if it contains valid numbers):
  ```
  int("123")  # Output: 123
  ```
- Trying to convert `"123abc"` to an `int` will cause an error.

Boolean (`bool`) Type

- Boolean values are either `True` or `False` (must be capitalized).
- Boolean values convert to numbers
  ```
  int(True)   # Output: 1
  int(False)  # Output: 0
  ```
- Numbers convert to Booleans:
  ```
  bool(1)  # Output: True
  bool(0)  # Output: False
  ```

For more data types, refer to the Python documentation at [Python.org](https://www.python.org).

02-Hands on Lab || Data types

Click here to open and practice the notebook

Lesson 03 || Expression and Variables

01-Expression and Variables

Expressions

Expressions are operations that Python performs to compute values. They consist of:

- **Operands**: The values being operated on.
- **Operators**: The symbols representing mathematical operations.

Arithmetic Operations

- **Addition (`+`)**: `100 + 60 → 160`
- **Subtraction (`-`)**: `10 - 15 → -5`
- **Multiplication (`*`)**: `5 * 5 → 25`
- **Division (`/`)**: Always returns a float.
    - `25 / 5 → 5.0`
    - `25 / 6 → 4.1667`
- **Integer Division (`//`)**: Rounds down to the nearest integer.
    - `25 // 6 → 4`

Order of Operations: Python follows standard precedence rules (`*` and `/` before `+` and `-`).

5 + 2 * 10 # Output: 25 (Multiplication first)

Parentheses override precedence:

(5 + 2) * 10 # Output: 70

Variables

Variables store values and can be reassigned.

```
my_variable = 1
print(my_variable)  # Output: 1
```

Reassigning a variable:

```
my_variable = 10
print(my_variable)  # Output: 10
    # The old value is overwritten.
```

Storing Expression Results

```
x = 8 + 4 / 3
print(x)        # Output: 9.333
```

Variables can be used in further calculations:

```
y = x / 3
print(y)  # Output: 3.111
```

Meaningful Variable Names

- Use meaningful names for better readability.
  Use underscores (`_`) or capital letters for clarity.

  ```
  total_min = 142
  total_hour = total_min / 60
  print(total_hour)  # Output: 2.367
  ```

- If `total_min` changes, `total_hour` updates accordingly without modifying the rest of the code.

02-Hands on lab || Expression & Variable

Click here to open and practice the notebook

Lesson 04 || String Operations

01-String Operations

Strings in Python

- A string is a sequence of characters enclosed in either double or single quotes.
- It can contain spaces, digits, or special characters.
- Strings can be assigned to variables and are best thought of as ordered sequences, with each character having an index.

Indexing and Slicing

- Each character in a string can be accessed using an index (starting from 0).
- Negative indices count from the end (`-1` represents the last character).
- Strings can be sliced using a start and end index, optionally with a stride to skip characters.

String Operations

- **Concatenation**: Combine strings using the `+` operator.
- **Replication**: Repeat strings using the `*` operator.

Immutability

- Strings are **immutable**, meaning their values cannot be changed directly.
- New strings can be created by modifying the original one.

Escape Sequences

- `\n` – Creates a new line.
- `\t` – Inserts a tab space.
- `\\` – Used to include a backslash.
- `r"string"` – Raw string, where escape sequences are not processed.

String Methods

- `upper()` – Converts all characters to uppercase.
- `replace(old, new)` – Replaces a substring with another.
- `find(substring)` – Returns the index of the first occurrence of a substring or `-1` if not found.

For more details, refer to Python documentation or practice in the lab.

02-Format Strings

Click here to read the pdf. Contains,

- F strings
- R strings

03-Hands on Lab || String Operations

click here to open and practice the notebook. Important topics,

- Indexing
    - Negative indexing
    - slicing
    - stride
    - concatenation
- Escape Sequence
- String Manipulation Operation
    - upper/lower
    - find
    - replace
    - split
- Regex
    - search
    - findall
    - group
      `The match.group() method is used in Python's re module to retrieve the part of the string where the regular expression pattern matched`
    - sub
      `The sub function of a regular expression in Python is used to replace all occurrences of a pattern in a string with a specified replacement.`

Requires Clarification on REGEX (group, sub)

Module Summary

01-Summary

- **Python Data Types**

    - Python differentiates between integers, floats, strings, and Booleans.
    - **Integers**: Whole numbers, positive or negative.
    - **Floats**: Include integers and decimal numbers.
    - **Typecasting** allows conversion between integers, floats, and strings.
    - Boolean values (`True`, `False`) correspond to `1` and `0`.
- **Expressions and Mathematical Operations**

    - Expressions combine values and operators to produce a result.
    - Python supports addition, subtraction, multiplication, division, and more.
    - `//` performs **integer division**, discarding the fractional part.
    - Python follows **BODMAS** for order of operations.
- **Variables**

    - Variables store and manipulate data in Python.
    - The `=` assignment operator assigns values to variables.
    - Assigning a new value to a variable overrides its previous value.
    - Variables can store expressions and interact with other variables.
- **String Operations**

    - Strings are sequences of characters enclosed in quotes.
    - They can contain letters, spaces, digits, and special characters.
    - **Indexing**: Access individual characters using positive or negative indices.
    - **Slicing**: Extract portions of a string with start, stop, and step values.

    - **Concatenation**: Combine strings using `+`.
    - **Replication**: Repeat strings using `*`.
    - **Immutability**: Strings cannot be modified; operations create new strings.
- **Escape Sequences**

    - `\n` – New line
    - `\t` – Tab space
    - `\\` – Backslash
- **String Methods**

    - `upper()` – Converts text to uppercase.
    - `replace(old, new)` – Replaces a substring.
    - `find(substring)` – Returns the index of a substring or `-1` if not found.

By mastering these concepts, you can effectively manipulate data, perform calculations, and work with text in Python.

02-CheatSheet

06_cheatsheet.pdf click to read the pdf

Python Data Structures

Module 02 || Course 04

IBM DS PRO

Lesson 01 || Lists and Tuples

01-Lists and Tuples

Tuples

- **Definition**: Tuples are ordered sequences enclosed in parentheses `()`.
- **Types**: Can contain different data types (strings, integers, floats, etc.).
- **Indexing**: Elements can be accessed using an index (both positive and negative).
- **Concatenation**: Tuples can be combined using `+`.
- **Slicing**: `slice()` Extracts multiple elements, with the last index being one larger than desired.
- **Length**: `len()` function returns the number of elements in a tuple.
- **Immutability**:
    - Tuples **cannot be modified** after creation.
    - Variables referencing the same tuple share the same immutable object.
    - To modify, a new tuple must be created.
- **Sorting**: The `sort()` function returns a sorted list from a tuple.
- **Nesting**:
    - Tuples can contain other tuples and complex data types.
    - Nested tuples can be accessed using multiple indices.

Lists

- **Definition**: Lists are ordered sequences enclosed in square brackets `[]`.
- **Types**: Can contain multiple data types and even other lists/tuples.
- **Indexing & Slicing**: Follows the same rules as tuples.
- **Concatenation**: Lists can be combined using `+`.
- **Mutability**:
  - Lists **can be modified** after creation.
  - Elements can be changed, added, or removed.
- **List Methods**:
  - `extend()` – Adds elements from another list.
  - `append()` – Adds a single element at the end.
  - `del` – Deletes elements using their index.
- **String to List Conversion**:
  - `split()` splits a string into a list based on spaces or specific delimiters.
- **Aliasing & Cloning**:
  - Assigning a list to another variable creates an alias (both reference the same object).
  - Cloning (`list_name[:]`- slicing) creates a separate copy to avoid unwanted modifications.
- **Help Command**: `help(list)` or `help(tuple)` provides more information on operations.

Lists and tuples are essential data structures in Python, each suited for different use cases based on **mutability**.

02-Hands on Lab

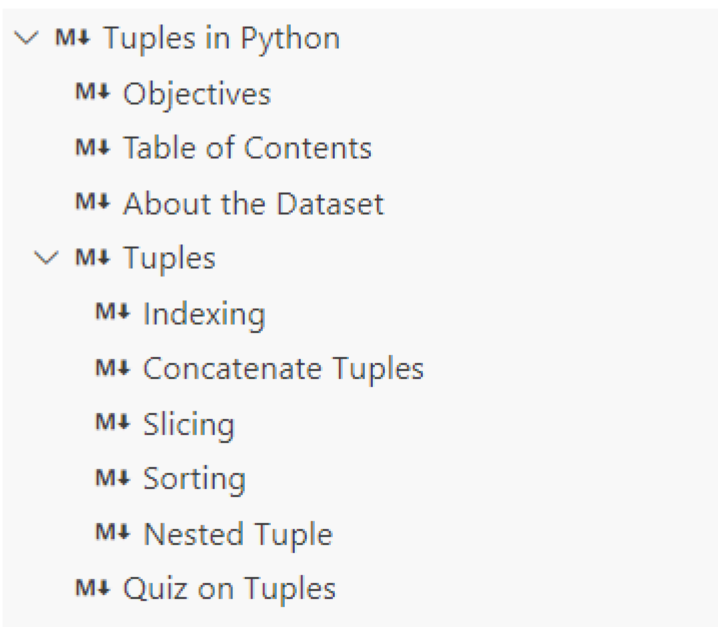
Lists

Click here to open the notebook.
new topics,

- del()
- append()
- extend()
- copy and clone

Tuple

Click here to open notebook
topics,



Find the first index of `"disco"`:

```python
# Write your code below and press Shift+Enter to execute
genres_tuple.index("disco")
```

28] ✓ 0.0s

```
7
```

03-Cheat Sheet Lists & Tuples

Click here to read pdf

Lesson 02 || Dictionary

01-Dictionary in Python

Overview

- A dictionary is a collection that stores **key-value** pairs.
- Unlike lists, dictionaries use **keys** instead of integer indexes.
- Keys must be **immutable** and **unique**, while values can be **mutable**, **immutable**, and **duplicate**.

Creating a Dictionary

- Use **curly brackets `{}`** to define a dictionary.
- Each key-value pair is separated by a **colon `:`**.
- Example:
  `album_dict = {"Back in Black": 1980, "The Dark Side Of The Moon": 1973}`

Accessing Values

- Use **square brackets `[]`** with the key to retrieve the value.
- Example: `album_dict["Back in Black"]` returns `1980`.

Adding and Deleting Entries

- **Adding**: Assign a new key-value pair → `album_dict["Graduation"] = 2007`.
- **Deleting**: Use `del` with the key → `del album_dict["Thriller"]`.

Checking for a Key

- Use the **`in`** keyword → `"Back in Black" in album_dict` returns `True`.

Getting Keys and Values

- **`keys()`** returns all dictionary keys → `album_dict.keys()`.
- **`values()`** returns all dictionary values → `album_dict.values()`.

Check out the labs for more examples! 🎵

02-Hands on Lab || Dictionaries

click here to open the jupyter

Lesson 03 || Sets

01-Sets in Python

Overview

- Sets are a type of **collection** like lists and tuples.
- **Unordered** – elements do not have a fixed position.
- **Unique elements** – duplicates are automatically removed.

Creating a Set

- Use **curly brackets `{}`** to define a set.
- Example: `my_set = {"AC/DC", "Back in Black", "Thriller"}`
- Duplicates are removed when the set is created.

Converting a List to a Set

- Use the `set()` function to remove duplicates.
- Example: `set(["AC/DC", "Back in Black", "AC/DC"])` → `{"AC/DC", "Back in Black"}`

Adding and Removing Elements

- **Add an element**: `my_set.add("NSYNC")`.
- **Remove an element**: `my_set.remove("NSYNC")`.

Checking Membership

- Use **`in`** to check if an element is in a set → `"AC/DC" in my_set` returns `True`.

Set Operations

Intersection (`&`)

- Returns a new set with elements **common** to both sets.
- Example: `set1 & set2` → `{common_elements}`.

Union (`|`)

- Returns a new set with **all** elements from both sets.
- Example: `set1 | set2` → `{all_unique_elements}`.

Subset (`issubset()`)

- Checks if a set is **fully contained** in another.
- Example: `set3.issubset(set1)` → `True`.

Check out the lab for more examples! 🎵

02-Hands on Lab || Sets

[click](#) here

Module Summary

Summary of Key Concepts

Tuples

- **Ordered & Immutable –** Once created, elements cannot be changed.
- **Defined using parentheses `()`** with comma-separated values.
- Can contain **strings, integers, and floats**.
- **Access elements** using both **positive and negative indexing**.
- **Operations**: Concatenation, slicing, and nesting (tuples inside tuples).
- **Modification** requires creating a new tuple.

Lists

- **Ordered & Mutable –** Elements can be modified.
- **Defined using square brackets `[]`** with comma-separated values.
- Can contain **strings, integers, floats, and nested lists**.
- **Access elements** using both **positive and negative indexing**.
- **Operations**: Adding, deleting, splitting, concatenation, and appending.
- **Aliasing**: Multiple names can refer to the same list.
- **Cloning** creates a separate copy.

Dictionaries

- **Key-Value pairs**, providing quick data retrieval.
- **Defined using curly brackets `{}`**.
- **Keys** must be **immutable and unique**.
- **Values** can be **mutable, immutable, and allow duplicates**.
- **Operations**: Adding, deleting, retrieving values, and checking key existence (`True`/`False`).
- **Methods**: Retrieve all keys (`keys()`) or values (`values()`).

Sets

- **Unordered collection** of **unique** elements.
- **Defined using curly brackets `{}`**.
- **No duplicates** – Automatically removes repeated values.
- **Operations**: Adding, removing, and checking membership (`in` keyword).
- **Set operations**:
  - **Intersection (`&`)** – Common elements in both sets.
  - **Union (`|`)** – Combines all unique elements.
  - **Subset (`issubset()`)** – Checks if one set is entirely within another.

Now you have a solid understanding of **tuples, lists, dictionaries, and sets** in Python! 🚀

Python Programming Fundamentals

Module 03 || Course 04

IBM DS PRO

Lesson 01 || Conditions & Branching

01-Conditions and Branching

Comparison Operations

- Compare values and return `True` or `False`.
- **Equality (`==`)** – Checks if two values are the same.
  - Example: `6 == 6 → True`, `7 == 6 → False`.
- **Greater than (`>`)** – Checks if the left value is larger.
  - Example: `6 > 5 → True`.
- **Greater than or equal (`>=`)** – Includes equality in the check.
  - Example: `5 >= 5 → True`.
- **Less than (`<`)** – Checks if the left value is smaller.
  - Example: `2 < 6 → True`.
- **Not equal (`!=`)** – Returns `True` if values are different.
  - Example: `2 != 6 → True`.
- Works for strings too, like `"ACDC" == "Michael Jackson" → False`.

Branching with If-Else

- **If Statement –** Runs a block of code if the condition is `True`.
  - Example

    ```
    if age >= 18:
         print("You will enter")
    ```

- **Else Statement** – Runs when `if` condition is `False`.
  - Example:

    ```
    if age < 18:
         print("Go to the Meatloaf concert")
    else:
         print("You will enter")
    ```

- **Elif (Else If) Statement –** Checks extra conditions if previous ones are `False`.
  - Example

    ```
    if age > 18:
         print("You will enter")
    elif age == 18:
         print("Go see Pink Floyd")
    else:
         print("Go to the Meatloaf concert").
    ```

Logical Operators

- **Not (`not`)** – Reverses `True` to `False` and vice versa.
  - Example:

    ```
    if not (age >= 18):
         print("You cannot enter the concert")
    ```

- **Or (`or`)** – Returns `True` if at least one condition is `True`.
  - Example:

    ```
    if album_year < 1980 or album_year > 1989:
         print("This album was made in the 70s or 90s")
    ```

- **And (`and`)** – Returns `True` only if both conditions are `True`.
  - Example:

    ```
    if album_year >= 1980 and album_year <= 1989:
         print("This album was made in the 80s").
    ```

Key Takeaways

- Use **comparison operators** (`==`, `>`, `<`, `>=`, `!=`) to evaluate conditions.
- **If-else statements** control the program flow based on conditions.
- **Logical operators** (`not`, `or`, `and`) help combine conditions.
- Example: Age-based concert entry decisions using `if`, `elif`, and `else`.

02-Hands on Lab || Conditionals

[click here]

contents,

- comparison operators

Similarly, from the table above we see that the value for **A** is 65, and the value for **B** is 66, therefore:

```python
# Compare characters

'B' > 'A'
```

0.0s Python

```
True
```

- Branching

Lesson 02 || Loops

01-Loops in Python

Range Function

- Generates an ordered sequence.
- If given a single positive integer `n`, it returns `[0, 1, ..., n-1]`.
- If given two numbers `a, b` (`a < b`), it returns `[a, a+1, ..., b-1]`.
- In Python 3, `range()` does not generate a list explicitly as in Python 2.

For Loops

- Used to perform repetitive tasks.

Using `enumerate()`

- Retrieves both index and element simultaneously.
- Example:
  - `enumerate(squares)` returns index-color pairs like `(0, "red")`, `(1, "yellow")`.

While Loops

- Executes repeatedly **while a condition is true**.

02-Loops Reading

should check the [pdf]

The Enumerated For Loop

Have you ever needed to keep track of both the item and its position in a list? An enumerated for loop comes to your rescue. It's like having a personal assistant who not only hands you the item but also tells you where to find it.

Consider this example:

```
fruits = ["apple", "banana", "orange"]
for index, fruit in enumerate(fruits):
    print(f"At position {index}, I found a {fruit}")
```

With this loop, you not only get the fruit but also its position in the list. It's as if you have a magical guide pointing out each fruit's location!

03-Hands on Lab || Loops

check the [notebook]

new content,

- Enumerated for loop

Lesson 03 || Functions

01-Functions

Introduction to Functions

- Functions take an input and produce an output or change.
- They help in reusing code efficiently.
- You can create your own functions or use built-in functions.
- Built-in functions do not require understanding of internal workings, just their usage.

Calling Functions vs Writing Repetitive Code

- Using functions reduces redundancy.
- Instead of writing repetitive lines of code, you can call a function multiple times.
- Example: Calling `f1` passes an input to a function and gets an output. The output can be used in another function `f2`.

Common Built-in Functions

1. **`len()`**
    - Takes a sequence (list, string, dictionary, etc.) and returns its length.
    - Example: `len([1, 2, 3, 4])` returns `4`.
2. **`sum()`**
    - Takes an iterable (list, tuple) and returns the total sum of its elements.
    - Example: `sum([10, 20, 30])` returns `60`.
3. **Sorting Methods**
    - `sorted(list)`: Returns a new sorted list without modifying the original list.
    - `list.sort()`: Sorts the list in place and modifies it.

Defining Custom Functions

Use `def` keyword followed by the function name and parameters.

```
def add_one(a):
     return a + 1
```

- Calling `add_one(5)` returns `6`.
- Calling `add_one(8)` returns `9`.

Function Documentation

- Use triple quotes (`'''`) for function documentation.
- The `help(function_name)` command displays the function's docstring.

Functions with Multiple Parameters

```
def multiply(a, b):
     return a * b
```

- `multiply(2, 3)` returns `6`.
- `multiply(10, 3.14)` returns `31.4`.
- Multiplying an integer and a string repeats the string.

Functions Without Return Statements

- If a function does not return anything, it returns `None`.

```
def print_name():
     print("Michael Jackson")
```

Calling `print_name()` prints "Michael Jackson" but returns `None`.

Functions with `pass`

- If a function has no implementation, use `pass` to avoid errors.

```
def no_work():
     pass
```

Functions with Loops

Example of using loops inside a function:

```
def print_elements(lst):
    for i, val in enumerate(lst):
        print(i, val)
```

Calls `print_elements([5, 10, 15])` prints:

```
0 5
1 10
2 15
```

Variadic Parameters (`*args`)

- Allows a function to accept a variable number of arguments.

```
def print_names(*names):
    for name in names:
        print(name)
```

Calling `print_names("Alice", "Bob")` prints:

```
Alice
Bob
```

Scope of Variables

Global vs Local Scope

- **Global variables**: Defined outside any function and accessible anywhere.
- **Local variables**: Defined inside a function and only accessible within it.

```
x = "Global"
def my_function():
     x = "Local"
     return x
```

- `my_function()` returns "Local".
- Printing `x` outside the function returns "Global".

Using Global Variables Inside Functions

- If a function accesses a global variable, it uses its global value.

```
rating = 9
def acdc():
     return rating + 1
```

- Calling `acdc()` returns `10`.

Modifying Global Variables in Functions

- Use the `global` keyword to modify a global variable inside a function.

```
def pink_floyd():
     global sales
     sales = "45 million"
```

- Calling `pink_floyd()` sets `sales` to "45 million" globally.

Conclusion

- Functions simplify and structure code.
- Built-in functions help perform common tasks efficiently.
- Custom functions allow flexibility and reusability.
- Understanding scope helps in managing variables efficiently.
- Check the lab for more hands-on examples!

02-Reading || Functions

should read the [pdf](). New topics,

- DocStrings (“””func documentation”””)
- Modify Data Structures using functions

Part 1: Initialize an empty list

```
# Define an empty list as the initial data structure
my_list = []
```

In this part, you start by creating an empty list named `my_list`. This empty list serves as the data structure that you will modify throughout the code.

Part 2: Define a function to add elements

```
# Function to add an element to the list
def add_element(data_structure, element):
    data_structure.append(element)
```

03-Hands on Lab || Functions

[click]() to get the jupyter notebook, things to check:

- docstrings
- help
- default func parameter

Lesson 04 || Exception Handling

01-Exception Handling

Introduction

- Exception handling prevents programs from crashing due to unexpected errors.
- Example: Entering a number instead of text in an input field triggers an error message instead of terminating the program.
- This happens because an event is triggered when the program tries to process incorrect input.
- Exception handling allows programs to catch errors and respond appropriately.

Try...Except Statement

- The `try` block attempts to execute the code.
- If an error occurs, execution moves to the appropriate `except` block.
- Example:
    - A program attempts to open and write a file.
    - If reading data fails, the program skips the `try` block and executes the `except` block.
    - If the error matches `IOError`, it prints: **"Unable to open or read the data in the file."**

Handling Multiple Exceptions

- If multiple types of errors can occur, multiple `except` blocks can be added.
- A generic `except` block (without specifying the error type) is not recommended because:
  - It catches all errors but provides no details, making debugging difficult in large programs.

Adding Else and Finally Statements

- **`else` statement**: Executes if no errors occur, providing confirmation.
  - Example: **"The file was written successfully."**
- **`finally` statement**: Executes regardless of errors, ensuring cleanup.
  - Example: **"File is now closed."**

Summary

- `try…except` helps handle errors and prevents program crashes.
- Defining specific errors improves debugging.
- `else` confirms successful execution.
- `finally` ensures essential cleanup, such as closing files.

This keeps the key concepts clear and structured. Let me know if you want any tweaks! 😊

02-Reading || Exception Handling

must check the pdf, contains

- types of exception
  - ZeroDivisionError
  - ValueError
  - FileNotFoundError
  - IndexError
  - KeyError
  - TypeError
  - AttributeError
  - ImportError
- try except

03-Hands on Lab || Exception Handling

must check the [notebook]() on exception handling

python documentation for [exception]()

try except else finally structure below

```python
# potential code before try catch

try:
    # code to try to execute
except ZeroDivisionError:
    # code to execute if there is a ZeroDivisionError
except NameError:
    # code to execute if there is a NameError
except:
    # code to execute if ther is any exception
else:
    # code to execute if there is no exception
finally:
    # code to execute at the end of the try except no matter what happens

# code that will execute if there is no exception or a one that we are handling
```

try except else finally example below

```python
a = 1

try:
    b = int(input("Please enter a number to divide a"))
    a = a/b
except ZeroDivisionError:
    print("The number you provided cant divide 1 because it is 0")
except ValueError:
    print("You did not provide a number")
except:
    print("Something went wrong")
else:
    print("success a=",a)
finally:
    print("Processing Complete")
```

Python

Lesson 05 || Object & Classes

01-Object & Classes

Objects in Python

- Python has different data types like integers, floats, strings, lists, and dictionaries.
- Each data type is an **object** with:
  - A **type**
  - An **internal representation**
  - A set of **methods** to interact with the data
- An **object** is an instance of a type.
- Example:
  - Creating an integer creates an **integer object**
  - Creating a list creates a **list object**
- The `type()` function helps identify an object's type.

Methods and Object Interaction

- Methods are functions specific to an object's type.
- Example:
  - The `sort()` method changes the data in a list.
  - The `reverse()` method reverses a list's order.
- Calling a method:
  - Use `object.method_name()` syntax.
- Methods **change** or **use** an object's data.

Creating Classes

- A **class** defines a new data type.
- Objects are created as **instances** of a class.
- Example:
    - A `Circle` class needs `radius` and `color` as data attributes.
    - A `Rectangle` class needs `height`, `width`, and `color`.
- Use the `class` keyword to define a class.
- Every class in this course has `object` as its parent.

Creating Objects

- To create an object, use the class constructor.
- Example:
    - A `Circle` object with `radius = 4`, `color = red`.
    - Another `Circle` object with `radius = 2`, `color = green`.
    - A `Rectangle` object with `height = 2`, `width = 3`, `color = blue`.
- Each object has **different attribute values** but belongs to the same class.

The Constructor (`__init__` Method)

- The `__init__` method initializes an object's attributes.
- `self` refers to the current instance of the class.
- Example:
    - `Circle` class sets `radius` and `color` when an object is created.
    - `Rectangle` class sets `height`, `width`, and `color`.

Accessing and Modifying Attributes

- Use `object.attribute_name` to access an attribute.
- Attributes can be modified directly.
    - Example:
      `circle1.radius = 10` changes the radius of `circle1`.
- Instead of modifying directly, **methods** are used for controlled changes.

Methods in Classes

- **Methods** define actions that an object can perform.
- Example:
    - `add_radius()` method increases the radius of a circle.
- Calling a method:
    - `circle1.add_radius(8)` increases `circle1`'s radius by 8.
- `self` ensures the method modifies the correct object.
- Some methods have **default values** for parameters.

Drawing Objects (Lab Example)

- `drawCircle()` method draws a circle object.
- `drawRectangle()` method draws a rectangle object.
- These methods help visualize objects.

Using `dir()` Function

- `dir(object)` lists all **methods and attributes** of an object.
- Attributes with underscores (`__name__`) are for internal use.
- Regular attributes are the **important ones** to focus on.

Summary

- A **class** defines an object's blueprint.
- An **object** is an instance of a class.
- Objects have **attributes (data)** and **methods (functions)**.
- Methods modify or interact with object data.
- The `dir()` function helps explore an object's capabilities.

02-Reading || Object & Classes

must read, click to open pdf, contains,

03-Hands on Lab || Objects & Classes

must click to open notebook, contains,

- class
- object
- method
- constructor
- attributes

and much more

Lesson 6-7 || Practice & Summary

01-Practice Lab || Text Analysis

[click](#)here to get the notebook

02-Summary || Module 03

Conditions and Branching

- `if` statements execute code based on **true/false conditions** from comparisons and Boolean expressions.
- Comparison operators: `=`, `>`, `<`
- `!` (exclamation mark) defines inequalities.
- Conditions can compare **integers, strings, and floats**.
- Branching directs program flow with **if, else, and elif** statements.
- `if` executes code when the condition is **true**.
- `else` executes when the condition is **false**.
- `elif` allows multiple conditions to be checked sequentially.

Loops

- **Loops automate repetition** and iterate over lists, dictionaries, etc.
- `range(start, stop, step)` generates sequences for loops.
- `for` loops iterate over sequences (lists, tuples, strings).
- `while` loops run as long as a condition is **true**.

Functions

- Functions are **reusable code blocks** that take inputs and return results.
- Python has built-in functions (`len()`, `sum()`, `sorted()`, `sort()`).
- You can **define custom functions**.
- Functions should be documented with a **docstring (`"""`...`"""`)**.
- `help(function_name)` retrieves function documentation.
- Functions can have **multiple parameters**.
- A function **without a return statement** returns `None`.
- `pass` keyword allows an empty function body.
- Functions typically perform **multiple tasks**.

Scope of Variables

- **Local scope**: Variable exists **only within a function**.
- **Global scope**: Variable can be accessed **anywhere** in the program.

Exception Handling

- Prevents errors from **crashing the program**.
- `try-except` handles errors safely.
- `try-except-else`: **Else block executes if no error occurs**.
- `try-except-else-finally`:
  - `try`: Attempt code execution.
  - `except`: Handle errors.
  - `else`: Runs if no error occurs.
  - `finally`: Always executes (cleanup actions).

Objects and Classes

- **Objects** are instances of classes with **data and behavior**.
- `type(object)` checks an object's type.
- Methods inside objects **can modify their attributes**.
- **Classes** are blueprints for creating objects with attributes and methods.
- `__init__` is a special method that **initializes attributes**.
- Objects can have **data attributes** (values that define them).
- **Methods** are functions inside a class that interact with data.
- Methods require `self` and may take additional parameters.

03-Cheatsheet || Python Fundamentals

click here to download pdf

Working with Data in Python

Module 04 || Course 04

IBM DS PRO

Lesson 01 || Reading & Writing Files

01-Reading Files with Open

Opening a File

- Use Python's `open()` function to create a **file object**.
- **Syntax:** `open("filename.txt", "mode")`
- **File path:** Includes **directory** and **file name**.
- **Modes:**
  - `'r'` → Read
  - `'w'` → Write
  - `'a'` → Append

File Object Attributes

- `name` → Returns file name as a string.
- `mode` → Returns file mode (`'r'`, `'w'`, etc.).

Closing a File

- Always close the file using `.close()`.
- **Better practice:** Use `with open() as file:` → **Automatically closes** after execution.

Reading File Content

- `.read()` → Reads **entire file** into a string.
- `.readlines()` → Returns **list of lines**, where each line is an element.
- `.readline()` → Reads **one line at a time**.

Reading Specific Characters

- `.readline(n)` → Reads **n** characters from a line.
- Calling `.readline(n)` multiple times continues reading from where it left off.

Looping Through File Content

- Use a `for` loop to read and print **each line individually**.

Raw Strings and New Lines

- `\n` represents a **new line** in Python strings.
- When reading a file, **`\n` is included** unless removed manually.

02-Reading || Reading Files with Open

01_Reading_Read_file_with_open.pdf

03-Hands on Lab || Reading Files

must practice this [notebook]

04-Writing Files with Python

Creating and Writing to a File

- Use `open("filename.txt", "w")` to **create a file** or overwrite an existing one.
- `.write("text")` writes data to the file.
- **Example:**
  - First `.write("This is line A\n")` → Writes "This is line A" and moves to a new line.
  - Second `.write("This is line B\n")` → Writes "This is line B".

Using `with` for Writing

- `with open()` **automatically closes** the file after execution.
- Writing multiple lines:
  - Store lines in a **list**.
  - Use a `for` loop to write each line to the file.

Appending to a File

- Use `open("filename.txt", "a")` to **append** without overwriting.
- `.write("This is line C\n")` → Adds new content at the end of the file.

Copying a File

1. Open the **source file** (`Example1.txt`) in `'r'` mode.
2. Open the **new file** (`Example3.txt`) in `'w'` mode.
3. Use a `for` loop to **copy line by line**.
4. Close both files.

05-Reading || Writing files

must click to read pdf

In Python, when opening a file using `open(filename, mode)`, the modes `"r+"`, `"w+"`, and `"a+"` define how the file is accessed. Here's the difference:

1. **`r+` (Read and Write)**
   - Opens the file for both reading and writing.
   - The file **must exist**; otherwise, it raises an error.
   - The file pointer is positioned at the beginning.
   - Does **not** truncate (delete contents).
2. **`w+` (Write and Read)**
   - Opens the file for both reading and writing.
   - If the file exists, it **truncates** (erases) its contents.
   - If the file does not exist, it creates a new one.
   - The file pointer is at the beginning.
3. **`a+` (Append and Read)**
   - Opens the file for both reading and appending.
   - If the file does not exist, it creates a new one.
   - The file pointer is at the **end**, so writing appends data instead of overwriting.
   - Reading starts from the beginning, but writing always happens at the end.

Summary Table:

Mode	Read	Write	Truncate	Pointer Start	Creates File if Missing
`r+`	✅	✅	❌	Beginning	❌
`w+`	✅	✅	✅	Beginning	✅
`a+`	✅	✅	❌	End (for writing)	✅

06-Hands on Lab || Writing Files

must practice this [jupyter notebook](), contains,

- `.tell()`
- `.seek(offset, from)`
- `.truncate()`

must do the exercise part in notebook

Lesson 02 || Pandas

01-Pandas | Loading Data

Dependencies and Libraries

Dependencies or libraries are pre-written code that helps solve problems. In this video, we introduce **pandas**, a popular library for data analysis.

Importing Pandas

- Use `import pandas` to import the library.
- This gives access to a large number of built-in classes and functions.
- If the library is not installed, you must install it first.

Reading CSV Files

- Pandas provides `read_csv` to load CSV files.
- Instead of typing `pandas` every time, use `import pandas as pd`.
- Now, use `pd.read_csv(file_path)` to load a CSV file into a **DataFrame**.

Reading Excel Files

- Use `pd.read_excel(file_path)` to load an Excel file.

Creating a DataFrame

- A **DataFrame** is a table with rows and columns.
- You can create one using a dictionary:
    - Keys correspond to column labels.
    - Values are lists corresponding to rows.
- Convert the dictionary into a DataFrame with `pd.DataFrame(dictionary)`.

Selecting Columns

- To select a single column: `df[['column_name']]` (creates a new DataFrame).
- To select multiple columns: `df[['column1', 'column2']]`.

Accessing Specific Elements

- **Using index positions (`iloc`):**
  - `df.iloc[row_index, column_index]`
  - Example: `df.iloc[0, 0]` (first row, first column).
- **Using labels (`loc`):**
  - `df.loc[row_label, column_label]`
  - Example: `df.loc['a', 'artist']` (first row, "artist" column).

Slicing DataFrames

- Select specific rows and columns: `df.iloc[:2, :3]` (first two rows, first three columns).
- Use `loc` for slicing based on labels: `df.loc[:, 'artist':'released']` (all rows, columns between "artist" and "released").

Check out the labs for more examples.

02-Pandas | Working & Saving Data

When working with a **DataFrame**, we can analyze data and save the results in different formats.

Finding Unique Elements

- Consider a stack of **13 blocks of different colors**, where there are only **3 unique colors**.
- In large datasets with millions of rows, finding unique values manually is difficult.
- Pandas provides the `unique` method to find unique elements in a column.
- Example: To find unique years in the "Released" column:
    - `df['Released'].unique()`

Filtering Data

- Suppose we want to create a new dataset of songs **from the 1980s and later**.
- We filter rows where the "Released" year is **after 1979**.
- Inequality operators can be used on entire columns in Pandas.
- Example: `df['Released'] > 1979` returns **Boolean values** (True/False).
- We use this Boolean series to **filter the DataFrame**:
    - `df1 = df[df['Released'] > 1979]`
- The new DataFrame **df1** contains only albums released after 1979.

Saving a DataFrame

- We can save the filtered DataFrame using `to_csv()`.
- Example: `df1.to_csv('filtered_albums.csv')`
- Other functions exist to save data in different formats.

03-Reading || Pandas

click to get pdf

off topic - pd series vs py list

in short, series is fast and efficient

Feature	Pandas Series	Python List
Data Type	Homogeneous (like NumPy arrays, but can store mixed types)	Heterogeneous (can store mixed data types)
Indexing	Supports labeled indexing	Only positional indexing
Operations	Vectorized operations (faster computations)	Requires loops for element-wise operations
Memory Efficiency	More memory efficient (uses NumPy under the hood)	Less efficient, as each element is a separate Python object
Functionality	Comes with built-in statistical and data-handling functions	No built-in operations for numerical/statistical computations
Integration	Works well with data analysis libraries like NumPy, Matplotlib	No direct integration with numerical computing tools

Accessing Elements in a Series

Accessing Elements in a Series

You can access elements in a Series using the index labels or integer positions. Here are a few common methods for accessing Series data:

Accessing by label

```
print(s[2])     # Access the element with label 2 (value 30)
```

Accessing by position

```
print(s.iloc[3]) # Access the element at position 3 (value 40)
```

Accessing multiple elements

```
print(s[1:4])   # Access a range of elements by label
```

DF & Series Attributes and Methods

DataFrame Attributes and Methods

DataFrames provide numerous attributes and methods for data manipulation and analysis, including:

- **shape**: Returns the dimensions (number of rows and columns) of the DataFrame.
- **info()**: Provides a summary of the DataFrame, including data types and non-null counts.
- **describe()**: Generates summary statistics for numerical columns.
- **head(), tail()**: Displays the first or last n rows of the DataFrame.
- **mean(), sum(), min(), max()**: Calculate summary statistics for columns.
- **sort_values()**: Sort the DataFrame by one or more columns.
- **groupby()**: Group data based on specific columns for aggregation.
- **fillna(), drop(), rename()**: Handle missing values, drop columns, or rename columns.
- **apply()**: Apply a function to each element, row, or column of the DataFrame.

Series Attributes and Methods

Pandas Series come with various attributes and methods to help you manipulate and analyze data

- **values**: Returns the Series data as a NumPy array.
- **index**: Returns the index (labels) of the Series.
- **shape**: Returns a tuple representing the dimensions of the Series.
- **size**: Returns the number of elements in the Series.
- **mean(), sum(), min(), max()**: Calculate summary statistics of the data.
- **unique(), nunique()**: Get unique values or the number of unique values.
- **sort_values(), sort_index()**: Sort the Series by values or index labels.
- **isnull(), notnull()**: Check for missing (NaN) or non-missing values.
- **apply()**: Apply a custom function to each element of the Series.

04-Practice Lab || Pandas

must practice the pandas lab notebook

05-Hands on Lab || Pandas

must practice the pandas lab notebook

Lesson 03 || Numpy

01-One Dimensional Numpy

Introduction to NumPy in 1D

NumPy is a **library for scientific computing** with many useful functions, offering advantages like **speed and memory efficiency**. It is also the foundation for **pandas**.

Topics Covered

- Basics and Array Creation
- Indexing and Slicing
- Basic Operations
- Universal Functions

Creating a NumPy Array

A **Python list** is a container that stores data, where elements are accessed using **indices**. A **NumPy array (ND array)** is similar but:

- **Fixed in size**
- **Contains elements of the same type**

To create a NumPy array:

1. Import NumPy.
2. Cast a list as a NumPy array.
3. Access elements using square brackets.

Example: `np.array([1, 2, 3])`

The array type is `numpy.ndarray`. We can check the **data type** using `.dtype`, which may return `int64` or `float64`, depending on the elements.

Basic Array Attributes

- **size** → Number of elements
- **ndim** → Number of dimensions (1D, 2D, etc.)
- **shape** → Tuple indicating size in each dimension

Indexing and Slicing

- **Modifying elements**:
  - Example: `a[0] = 100` (Changes the first element)
  - Example: `a[4] = 0` (Changes the fifth element)
- **Slicing arrays**:
  - Example: `d = a[1:4]` (Selects elements at indices 1 to 3)
  - Like lists, slicing **excludes the last index**.

Operations on 1D Arrays

NumPy allows **faster and more memory-efficient** operations compared to regular Python lists.

Vector Addition

- Consider vectors **u** and **v**:
  - First component of `z = u[0] + v[0]`
  - Second component of `z = u[1] + v[1]`
- Visualized as **arrows** using the **tip-to-tail method**.
- **One-line NumPy operation**: `z = u + v`
- Faster than multiple Python list operations.

Vector Subtraction

- Similar to addition but with a `-` sign.
- **One-line NumPy operation**: `z = u - v`

Scalar Multiplication

- Example: `y * 2` (Each element is multiplied by 2).
- Results in a **stretched vector**.

Hadamard Product (Element-wise Multiplication)

- Example: `z = u * v`
- Each component of `z` is the product of corresponding elements in `u` and `v`.

Dot Product

- **Measures similarity between two vectors.**
- Computed as: `(u[0] * v[0]) + (u[1] * v[1])`
- NumPy function: `np.dot(u, v)`

Broadcasting

NumPy allows operations to be applied **to all elements at once**:

- Example: `a + 5` (Adds 5 to each element).

Universal Functions

Universal functions (ufuncs) operate on entire arrays.

- Example: `a.mean()` (Computes the mean).
- Example: `b.max()` (Finds the max value).

Applying Functions

- Example: `np.sin(x)` applies `sin()` to each element.
- **Line space function** (`np.linspace(start, end, num)`) generates evenly spaced numbers.

Plotting with Matplotlib

- Generate `x` values: `np.linspace(0, 2*np.pi, 100)`
- Compute `y`: `np.sin(x)`
- Use `matplotlib.pyplot` to plot graphs.

Summary

NumPy **simplifies mathematical operations** in data science by making them **faster and more efficient** than Python lists. Explore more on **numpy.org**.

02-Hands on Lab || 1D Numpy Array

must click to download jupyter notebook

03-Reading || Matrix Mathematics

not necessary, matrix calculation basics

04-Two Dimensional Numpy

We can create **NumPy arrays with more than one dimension**. This section focuses on **2D arrays**, but NumPy supports **higher-dimensional arrays** as well.

Topics Covered

- **Basics and Array Creation in 2D**
- **Indexing and Slicing in 2D**
- **Basic Operations in 2D**

Creating a 2D NumPy Array

A **list of lists** can be converted into a **NumPy array** using

```
np.array([[1, 2], [4, 5], [7, 8]])
```

3 rows 2 columns, 3*2 matrix
Each **nested list** corresponds to a **row** in the array.

Understanding Dimensions

- The **ndim** attribute gives the number of dimensions (or axes).
- The **shape** attribute returns a **tuple** indicating:
    - **Rows** → Number of nested lists
    - **Columns** → Size of each nested list

For a **3×2** array:

- **First dimension** → Number of lists (Rows = 3)
- **Second dimension** → Number of elements per list (Columns = 2)

The **size** attribute gives the **total number of elements**, which is calculated as

```
3 × 2 = 6
```

Indexing in 2D Arrays

Access elements using **row and column indices**, for example, `a[1, 2]` retrieves the element at 2nd row and 3rd column

Alternatively, **single brackets** can be used to access entire rows, such as `a[1]`, which retrieves the second row `[4, 5]`.

Slicing in 2D Arrays

- Selecting specific **rows and columns**, such as
  `a[1:3, 0:2]`
  extracts
  `[[4, 5], [7, 8]]`.
- Selecting all **rows** but specific **columns**, such as
  `a[:, 1]`
  extracts
  `[2, 5, 8]`.

Basic Operations in 2D

Matrix Addition

Adding **two matrices** follows element-wise addition. Given

```
X = [[1, 2], [3, 4]]
```

and `Y = [[5, 6], [7, 8]]`

their sum,

```
X + Y
```

results in

```
[[6, 8], [10, 12]].
```

Scalar Multiplication

Multiplying a **matrix by a scalar** multiplies **each element**, such as

```
X * 2
```

which results in,

```
[[2, 4], [6, 8]].
```

Hadamard Product (Element-wise Multiplication)

Each element is multiplied by the corresponding element in the other matrix. Given

```
X = [[1, 2], [3, 4]]
Y = [[5, 6], [7, 8]]
```

their element-wise product

```
X * Y
```

results in

```
[[5, 12], [21, 32]].
```

Matrix Multiplication

For matrix multiplication, the **number of columns in the first matrix must match the number of rows in the second**. Given

```
A = [[0, 1], [2, 3]]
B = [[1, 2], [3, 4]]
```

their matrix multiplication

```
np.dot(A, B)
```

results in

```
[[3, 4], [11, 16]]
```

Each row of `A` is multiplied by each column of `B` using **dot product**.

Summary

NumPy makes working with **2D arrays efficient and intuitive**. You can perform:

- **Indexing and slicing** to extract data
- **Element-wise and matrix operations**
- **Matrix multiplication** for linear algebra applications

For more, visit **numpy.org**.

05-Hands on Lab || 2D Numpy Array

click to practice jupyter notebook

06-Reading || Beginner's Guide to Numpy

click to read the pdf

Operation with NumPy

Here's the list of operation which can be performed using Numpy

Operation	Description	Example
Array Creation	Creating a NumPy array.	arr = np.array([1, 2, 3, 4, 5])
Element-Wise Arithmetic	Element-wise addition, subtraction, and so on.	result = arr1 + arr2
Scalar Arithmetic	Scalar addition, subtraction, and so on.	result = arr * 2
Element-Wise Functions	Applying functions to each element.	result = np.sqrt(arr)
Sum and Mean	Calculating the sum and mean of an array.Calculating the sum and mean of an array.	total = np.sum(arr) average = np.mean(arr)
Maximum and Minimum Values	Finding the maximum and minimum values.	max_val = np.max(arr) min_val = np.min(arr)
Reshaping	Changing the shape of an array.	reshaped_arr = arr.reshape(2, 3)
Transposition	Transposing a multi-dimensional array.	transposed_arr = arr.T
Matrix Multiplication	Performing matrix multiplication.	result = np.dot(matrix1, matrix2)

07-Reading || Some Context on API

click to read the pdf

Module Summary

01-Summary

Here's a more concise version of your summary:

- **File Handling in Python**: Use `open()` to read (`r`), write (`w`), or append (`a`) files. Use `with open()` for safe file handling. `\n` starts a new line.

- **Pandas Basics**: Pandas is used for data manipulation with DataFrames (tables with rows and columns). Import with `import pandas as pd`. Use `df` to work with and modify data.

- **NumPy Basics**: NumPy provides efficient numerical operations. Arrays (`ndarray`) are like lists but optimized. Use `.dtype` for data type, `.size` for the total elements, and `.ndim` for dimensions.

- **NumPy Operations**: Supports indexing, slicing, vector addition/subtraction, scalar multiplication, Hadamard product (element-wise multiplication), and dot product (matrix multiplication).

- **2D NumPy Arrays**: Represent data in a grid (rows and columns). `.shape` gives dimensions, and `.size` gives total elements. Use brackets for indexing.

- **Visualization**: NumPy works with Matplotlib for plotting data.

This keeps the key ideas while making it even more compact. Let me know if you need further refinements!

02-Cheatsheet || Working with data in Python

click to read

APIs and Data Collection

Module 05 || Course 04
IBM DS PRO

Lesson 01 || Simple APIs

01-API

APIs and REST APIs Overview

What is an API?

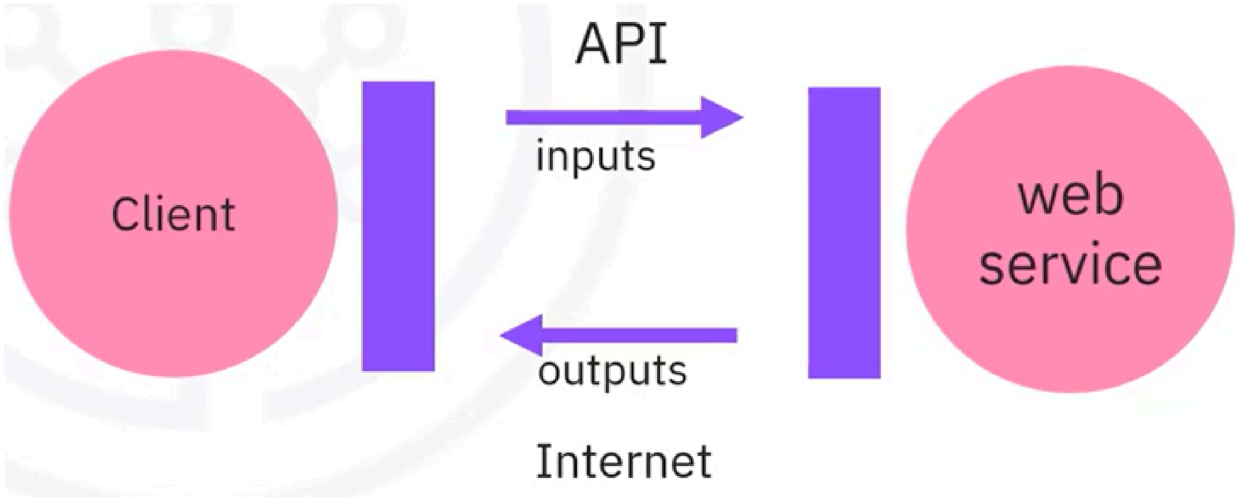
- An **API (Application Programming Interface)** allows two software components to communicate.
- Like a function, you only need to know the **inputs and outputs**, not how it works internally.
- Example: Pandas itself is an API that interacts with various software components.

Using APIs in Pandas

- When you create a **DataFrame**, you're creating an **instance** of the Pandas API.
- Methods like `head()` and `mean()` communicate with the API to process data.

REST APIs

- **REST (Representational State Transfer) APIs** allow communication over the internet.
- Your program is the **client**, and the API connects to a **web service (resource)**.
- Communication follows a set of rules using **requests and responses**.



Key Concepts

- **Client**: Your program
- **Resource**: The web service providing data
- **Endpoint**: The URL where the service is accessed
- **HTTP Methods**: Specify the type of operation (e.g., GET, POST)
- **JSON Format**: Used for requests and responses

PyCoinGecko for Cryptocurrency Data

- **Why use APIs for crypto data?**
  - Crypto data is constantly updated, making APIs useful for real-time trading.

Steps to Collect Bitcoin Data

1. **Install & import PyCoinGecko**
2. **Create a client object**
3. **Request Bitcoin data (past 30 days in USD)**
4. **Extract prices from JSON response**
5. **Convert data to a Pandas DataFrame**
6. **Format timestamps using `to_datetime()`**

```
!pip install pycoingecko
from pycoingecko import CoinGeckoAPI
cg = CoinGeckoAPI()
bitcoin_data = cg.get_coin_market_chart_by_id(id = 'bitcoin',
vs_currency = 'usd', days=30)
```

```
import pandas as pd
data = pd.DataFrame(bitcoin_data['prices'], columns=['TimeStamp','Price’])
```

```
data['Date'] = pd.to_datetime(data['TimeStamp'], unit = 'ms’)
```

Creating a Candlestick Chart

- **Group data by date** to find daily min, max, open, and close prices.
- **Use Plotly** to generate the candlestick chart.
- **Save as an HTML file**, then open it and click "Trust HTML" to view.

This gives a structured, simplified summary of the video transcript. Let me know if you need adjustments!

```
candlestick_data = data.groupby(data.Date.dt.date).agg({'Price': ['min',
'max', 'first', 'last']})
```

```
import plotly.graph_objects as go
import plotlyfig = go.Figure(data=[go.Candlestick(x = candlestick_data.index,
                                     open = candlestick_data['Price']['first'],
                                     high = candlestick_data['Price']['max'],
                                     low = candlestick_data['Price']['min'],
                                     close = candlestick_data['Price']['last']
                                    )
                     ])
fig.update_layout(xaxis_rangeslider_visible = False, xaxis_title = 'Date',
yaxis_title = 'Price (USD $)', title = 'Bitcoin Candlestick Chart Over Past 30
Days')

plotly.offline.plot(fig, filename = './bitcoin_candlestick_graph.html',
auto_open=False)
```

02-Hands on Lab || Intro to API

03-Practice Project || GDP Data

Lesson 02 || Rest API & Web Scraping

01-Rest API p1

HTTP Protocol Overview

What is the HTTP Protocol?

- **HTTP (HyperText Transfer Protocol)** is a general protocol for transferring data on the web.
- REST APIs use **HTTP requests and responses** to communicate.
- The **client (browser)** sends an HTTP request, and the **server** responds with the requested resource.

Uniform Resource Locator (URL)

A URL helps locate web resources and consists of three parts:

1. **Scheme** – The protocol (e.g., `http://`).
2. **Base URL** – The website address (e.g., `www.ibm.com`).
3. **Route** – The specific resource location (e.g., `/images/logo.png`).

HTTP Request and Response

Request Message Structure

- **Start Line** – Includes the HTTP method (e.g., `GET index.html`).

- **Headers** – Additional information (empty in simple `GET` requests).
- **Body** – Contains data (used in `POST` requests).

Response Message Structure

- **Start Line** – Includes HTTP version, status code (`200 OK`).
- **Headers** – Metadata about the response.
- **Body** – Contains the requested content (e.g., an HTML page).

Common HTTP Status Codes

- **1xx** – Informational (e.g., `100 Continue`).
- **2xx** – Success (e.g., `200 OK`).
- **4xx** – Client errors (e.g., `401 Unauthorized`).
- **5xx** – Server errors (e.g., `501 Not Implemented`).

HTTP Methods

- **GET** – Retrieve data from the server.
- **POST** – Send data to the server.

In the next video, Python will be used to apply **GET** (retrieving data) and **POST** (sending data).

02-Rest API p2

HTTP Requests in Python using the Requests Library

What is the Requests Library?

- A Python library for sending **HTTP/1.1 requests** easily.
- Alternative to `httplib` and `urllib`.

GET Requests

- Used to **retrieve** data from a web server.
- Example: `requests.get("https://www.ibm.com")`
- The response object (`r`) contains:
  - **Status code**: `r.status_code` (200 means OK)
  - **Headers**: `r.headers` (contains metadata like `Date` and `Content-Type`)
  - **Body**: `r.text` (contains HTML if applicable)

Query Strings in GET Requests

- Allows sending parameters via the **URL**.
- Format: `?key1=value1&key2=value2`

Example: Sending `name=Joseph` and `ID=123` using `httpbin.org`:
payload = {"name": "Joseph", "ID": "123"}
r = requests.get("https://httpbin.org/get", params=payload)

-

The **query string** appears in the URL:
https://httpbin.org/get?name=Joseph&ID=123

-
- The response is usually in **JSON** format and can be converted to a Python dictionary using `r.json()`.

POST Requests

- Used to **send** data to a server.
- Unlike GET, **data is sent in the request body**, not the URL.

Example: Sending data using `httpbin.org/post`:

payload = {"name": "Joseph", "ID": "123"}

r = requests.post("https://httpbin.org/post", data=payload)

-
- The URL **does not** contain query parameters.
- The response body contains the sent data under the key `"form"`.

Key Differences Between GET and POST

Feature	GET Request	POST Request
Purpose	Retrieve data	Send data
Data Location	URL (query string)	Request body
Visibility	Visible in URL	Hidden in body
Usage	Fetch web pages, APIs	Submit forms, upload files

This covers the basics of HTTP requests in Python. Let me know if you need further simplifications! 🚀

03-Reading || Web Scraping & HTML basics

04-Hands on Lab || Access REST API & Request HTTP

more on [requests]() (may need in future)

must practice [jupyter notebook]()

05-Hands on Lab || API Example

more on [randomuser]() module (may need in future)

06-HTML for Web Scrapping

Introduction to HTML for Web Scraping

Web pages contain valuable data, such as real estate prices, coding solutions, and sports statistics. Understanding **HTML** allows us to extract this data using **Python**.

---

1. Basic Structure of an HTML Page

- **HTML tags** define how content is displayed.
- **DOCTYPE html**: Declares the document as an HTML file.
- **`<html>`**: Root element of the page.
- **`<head>`**: Contains metadata.
- **`<body>`**: Contains visible content.

Example:

```
<!DOCTYPE html>
<html>
<head>
    <title>NBA Salaries</title>
</head>
<body>
    <h3>Player Name: John Doe</h3>
    <p>Salary: $10,000,000</p>
</body>
</html>
```

- **`<h3>`**: Displays bold headings (e.g., player names).
- **`<p>`**: Represents a paragraph (e.g., player salaries).

---

2. HTML Tag Composition

Every **HTML element** consists of:

- **Start tag** (`<a>` for hyperlinks).
- **Content** (text or other elements).
- **End tag** (`</a>` to close the hyperlink).
- **Attributes** (provides additional info).

Example of a **hyperlink**:

```html
<a href="https://www.ibm.com">IBM</a>
```

- **Tag Name**: `a` (defines a link).
- **Attribute Name**: `href` (specifies the link destination).
- **Attribute Value**: `"https://www.ibm.com"` (the URL).

---

3. HTML Document as a Tree

HTML documents follow a **tree structure**:

- **Parent**: An element containing other elements (e.g., `<html>` is the parent of `<head>` and `<body>`).
- **Children**: Elements inside a parent (e.g., `<h3>` and `<p>` inside `<body>`).
- **Siblings**: Elements at the same level (e.g., `<h3>` and `<p>` are siblings).

Example:

```
<html>
  ├── <head>
  │     └── <title>
  └── <body>
        ├── <h3>
        ├── <p>
        └── <b>
```

---

4. HTML Tables

Tables structure data using:

- **<table>**: Defines the table.
- **<tr>**: Defines a row.
- **<th>**: Defines a table header (bold).
- **<td>**: Defines a table cell.

Example:

```
<table>
    <tr>
        <th>Player</th>
        <th>Salary</th>
    </tr>
    <tr>
        <td>John Doe</td>
        <td>$10,000,000</td>
    </tr>
</table>
```

Next Steps: Extracting Data

With this knowledge, we can now use Python to scrape and extract data from web pages. 🚀

07-Web Scraping

Introduction to Web Scraping

Manually copying data from websites is time-consuming and inefficient. **Web scraping** automates this process, allowing us to extract and analyze data quickly using Python.

---

1. What is Web Scraping?

- **Definition**: Web scraping is the process of extracting data from websites automatically.
- **Why use it?**: It saves time and effort when collecting large amounts of data (e.g., analyzing sports players' performance).

---

2. Tools for Web Scraping

To scrape a webpage, we use **two Python libraries**:

- **Requests**: Downloads the webpage content.
- **BeautifulSoup**: Parses and extracts information from HTML.

Example:

```
from bs4 import BeautifulSoup
import requests

# Download the webpage
url = "https://example.com"
page = requests.get(url).text

# Parse HTML
soup = BeautifulSoup(page, "html.parser")
```

---

3. Understanding BeautifulSoup Objects

BeautifulSoup represents **HTML as a tree structure**, allowing easy navigation and data extraction.

Navigating the HTML Tree

Access a tag (e.g., first <h3> element):

```
tag = soup.h3
print(tag.text)  # Extracts text content
```

Finding a tag's parent (moves up the tree):

```
print(tag.parent)
```

Finding the next sibling (moves to the next element at the same level):

```
print(tag.next_sibling)
```

---

4. Using `find_all()` for Filtering

The `find_all()` method retrieves all matching elements based on:

Tag name:

```
soup.find_all("h3")  # Finds all <h3> elements
```

Attributes:

```
soup.find_all("a", href=True)  # Finds all links
```

Text content:

```
soup.find_all(string="Lebron James")
    # Finds elements containing this text
```

5. Extracting Data from HTML Tables

Web pages often store data in **tables**, which can be extracted using BeautifulSoup.

Example: Extracting table rows and cells

```
table = soup.find("table")
rows = table.find_all("tr")

for row in rows:
    cells = row.find_all("td")
    for cell in cells:
        print(cell.text)
            # Extracts text from each cell
```

---

6. Steps to Web Scraping

1. **Import required libraries** (`requests`, `BeautifulSoup`).
2. **Download webpage** using `requests.get()`.
3. **Parse HTML** with BeautifulSoup.
4. **Find and extract elements** (`find()`, `find_all()`).
5. **Process extracted data** (store in lists, DataFrames, etc.).

Next Steps

Try scraping a real website using BeautifulSoup! 🚀

08-Reading || Web Scraping

key libraries

- beautifulsoup4
- scrapy
- selenium

must read the pdf

09-Working with Different File Formats

Working with Different File Formats

When working with data, you'll encounter various file formats. Python simplifies reading and extracting data using predefined libraries.

---

1. Common File Formats

File extensions help identify file types and determine how to read them. Some common formats:

- **CSV (`.csv`)** – Comma-separated values
- **JSON (`.json`)** – JavaScript Object Notation
- **XML (`.xml`)** – Extensible Markup Language

2. Reading CSV Files

Python’s **Pandas** library makes it easy to read CSV files.

Example:

```
import pandas as pd

df = pd.read_csv("FileExample.csv")
print(df)
```

If the file has no headers, Pandas assumes the first row is the header. To specify column names:

df.columns = ["Column1", "Column2", "Column3"]

---

3. Reading JSON Files

JSON stores data in key-value pairs, similar to a Python dictionary.

Example:

```
import json

with open("FileExample.json") as file:
   data = json.load(file)

print(data)
```

4. Reading XML Files

XML organizes data using nested tags, similar to HTML. Since Pandas doesn't natively support XML, we use `xml.etree.ElementTree`.

Steps to Read an XML File:

1. Import `xml.etree.ElementTree`
2. Parse the file
3. Extract data using loops

Example:

```
import xml.etree.ElementTree as ET

tree = ET.parse("FileExample.xml")
root = tree.getroot()

for element in root:
   print(element.tag, element.text)
```

5. Summary

✅ Recognized different file types
✅ Used Python libraries (`pandas`, `json`, `xml.etree.ElementTree`) to read data
✅ Organized data using DataFrames

Now, try opening and processing different file formats on your own! 🚀

10-Hands on Lab || Working with Different file Format

must practice jupyter notebook

in pandas

Read/Save Other Data Formats

Data Formate	Read	Save
csv	`pd.read_csv()`	`df.to_csv()`
json	`pd.read_json()`	`df.to_json()`
excel	`pd.read_excel()`	`df.to_excel()`
hdf	`pd.read_hdf()`	`df.to_hdf()`
sql	`pd.read_sql()`	`df.to_sql()`
...	...	...

Module Summary

Module Summary: Working with APIs, Web Scraping, and File Formats

1. APIs in Python

- APIs (Application Programming Interfaces) allow two pieces of software to communicate.
- Simple APIs provide easy-to-use methods for interacting with services or data.
- Using an API in Python involves:
  - Importing the required library
  - Making HTTP requests
  - Parsing responses

2. Pandas API and Data Processing

- Pandas API interacts with other software components for data processing.
- Creating a Pandas object involves:
  - Defining a dictionary
  - Using the DataFrame constructor
- Common Pandas methods:
  - `head(n)`: Displays the first `n` rows (default is 5).
  - `mean()`: Computes the mean of numerical columns.

3. REST APIs & HTTP Communication

- REST APIs enable internet-based communication for accessing resources and AI models.
- HTTP (HyperText Transfer Protocol) transfers data over the web.
- HTTP messages often contain JSON files.
- A **client** sends requests, and a **server** responds.
- Common HTTP request methods:
  - **GET**: Retrieves information
  - **POST**: Submits data
  - **PUT**: Updates data
  - **DELETE**: Removes data

4. URLs and Query Strings

- A **URL (Uniform Resource Locator)** consists of:
  - **Scheme** (e.g., `https://`)
  - **Base URL** (e.g., `api.example.com`)
  - **Route** (e.g., `/data`)
- Query strings modify request results (e.g., filtering by ID).

5. Web Scraping with Python

- Web scraping extracts data from websites using:
  - **Requests**: Fetches web pages
  - **Beautiful Soup**: Parses HTML/XML
- HTML structure:
  - Tags (`<p>`, `<table>`, etc.) form a tree-like structure.
  - HTML tables contain headers, rows, and data.
- Methods for extracting data:
  - `find_all()`: Retrieves all matching elements.
  - `read_html()`: Extracts tabular data with Pandas.

6. File Formats in Python

- File formats define how data is stored (e.g., `.csv`, `.json`, `.xml`).
- **CSV (Comma-Separated Values):**
  - Read with `pandas.read_csv()`.
- **JSON (JavaScript Object Notation):**
  - Read using the `json` library.
- **XML (Extensible Markup Language):**
  - Parsed using `xml.etree.ElementTree`.

Key Takeaways

✅ APIs enable software interaction
✅ Pandas helps process and analyze data
✅ HTTP methods facilitate data exchange
✅ Web scraping extracts structured data from websites
✅ Python can read multiple file formats like CSV, JSON, and XML

You've now completed this module! 🚀