Table of Contents

[1 Course Overview 12](#_Toc210755756)

[1.1 Course Content 12](#_Toc210755757)

[1.1.1 Module 1: Python Basics 12](#_Toc210755758)

[1.1.2 Module 2: Python Data Structures 12](#_Toc210755759)

[1.1.3 Module 3: Python Programming Fundamentals 12](#_Toc210755760)

[1.1.4 Module 4: Working with Data in Python 13](#_Toc210755761)

[1.1.5 Module 5: APIs and Data Collection 13](#_Toc210755762)

[1.2 Learning Assets 13](#_Toc210755763)

[2 Introduction to Python 13](#_Toc210755764)

[2.1 Users of Python 13](#_Toc210755765)

[2.1.1 Experienced Programmers 13](#_Toc210755766)

[2.1.2 Beginners 13](#_Toc210755767)

[2.2 Benefits of Using Python 13](#_Toc210755768)

[2.3 Popularity of Python 14](#_Toc210755769)

[2.3.1 Organizations Using Python 14](#_Toc210755770)

[2.4 Python in Data Science and AI 14](#_Toc210755771)

[2.4.1 Scientific Computing Libraries 14](#_Toc210755772)

[2.4.2 Artificial Intelligence and Machine Learning 15](#_Toc210755773)

[2.4.3 Natural Language Processing 15](#_Toc210755774)

[2.5 Diversity and Inclusion in the Python Community 15](#_Toc210755775)

[2.5.1 Python Software Foundation 15](#_Toc210755776)

[2.5.2 PyLadies 15](#_Toc210755777)

[2.6 Summary 15](#_Toc210755778)

[3 Getting Started with Jupyter 15](#_Toc210755779)

[3.1 Introduction 15](#_Toc210755780)

[3.2 Launching a Notebook 15](#_Toc210755781)

[3.2.1 Running Your First Cell 16](#_Toc210755782)

[3.3 Working with Cells 16](#_Toc210755783)

[3.3.1 Inserting a Cell 16](#_Toc210755784)

[3.3.2 Deleting a Cell 16](#_Toc210755785)

[3.3.3 Moving Cells 16](#_Toc210755786)

[3.4 Working with Multiple Notebooks 16](#_Toc210755787)

[3.4.1 Example 16](#_Toc210755788)

[3.5 Presenting with Jupyter 16](#_Toc210755789)

[3.6 Shutting Down Notebooks 17](#_Toc210755790)

[3.7 Summary 17](#_Toc210755791)

[4 Module 1 Summary: Python Basics 17](#_Toc210755792)

[4.1 Data Types in Python 17](#_Toc210755793)

[4.1.1 Typecasting 17](#_Toc210755794)

[4.2 Expressions and Operations 17](#_Toc210755795)

[4.3 Variables 18](#_Toc210755796)

[4.4 String Operations 18](#_Toc210755797)

[4.5 Key Takeaways 18](#_Toc210755798)

[5 Lists and Tuples 19](#_Toc210755799)

[5.1 Compound Data Types 19](#_Toc210755800)

[5.2 Tuples 19](#_Toc210755801)

[5.2.1 Definition 19](#_Toc210755802)

[5.2.2 Accessing Elements 19](#_Toc210755803)

[5.2.3 Operations on Tuples 19](#_Toc210755804)

[5.2.4 Immutability 19](#_Toc210755805)

[5.2.5 Functions with Tuples 19](#_Toc210755806)

[5.2.6 Nesting and Indexing 19](#_Toc210755807)

[5.3 Lists 20](#_Toc210755808)

[5.3.1 Definition 20](#_Toc210755809)

[5.3.2 Accessing Elements 20](#_Toc210755810)

[5.3.3 Operations on Lists 20](#_Toc210755811)

[5.3.4 Mutability 20](#_Toc210755812)

[5.3.5 Conversion 21](#_Toc210755813)

[5.4 Summary 21](#_Toc210755814)

[6 Dictionaries 21](#_Toc210755815)

[6.1 Definition 21](#_Toc210755816)

[6.2 Structure 21](#_Toc210755817)

[6.3 Example 21](#_Toc210755818)

[6.4 Accessing Values 21](#_Toc210755819)

[6.5 Modifying Dictionaries 21](#_Toc210755820)

[6.6 Checking Membership 22](#_Toc210755821)

[6.7 Dictionary Methods 22](#_Toc210755822)

[6.8 Visualization 22](#_Toc210755823)

[6.9 Summary 22](#_Toc210755824)

[7 Sets 22](#_Toc210755825)

[7.1 Definition 22](#_Toc210755826)

[7.2 Creating Sets 22](#_Toc210755827)

[7.3 Basic Set Operations 23](#_Toc210755828)

[7.3.1 Adding Elements 23](#_Toc210755829)

[7.3.2 Removing Elements 23](#_Toc210755830)

[7.3.3 Membership Test 23](#_Toc210755831)

[7.4 Mathematical Set Operations 23](#_Toc210755832)

[7.4.1 Intersection 23](#_Toc210755833)

[7.4.2 Union 23](#_Toc210755834)

[7.4.3 Subset 23](#_Toc210755835)

[7.5 Visualization with Venn Diagrams 23](#_Toc210755836)

[7.6 Summary 24](#_Toc210755837)

[8 Module 2 Summary: Python Data Structures 24](#_Toc210755838)

[8.1 Tuples 24](#_Toc210755839)

[8.2 Lists 24](#_Toc210755840)

[8.3 Dictionaries 24](#_Toc210755841)

[8.4 Sets 24](#_Toc210755842)

[8.5 Summary 25](#_Toc210755843)

[9 Exploring Python Functions 25](#_Toc210755844)

[9.1 Objectives 25](#_Toc210755845)

[9.2 Introduction to Functions 25](#_Toc210755846)

[9.3 Purpose of Functions 25](#_Toc210755847)

[9.4 Benefits of Using Functions 25](#_Toc210755848)

[9.5 How Functions Work 26](#_Toc210755849)

[9.5.1 Inputs (Parameters) 26](#_Toc210755850)

[9.5.2 Performing Tasks 26](#_Toc210755851)

[9.5.3 Producing Outputs 26](#_Toc210755852)

[9.6 Python’s Built-in Functions 26](#_Toc210755853)

[9.6.1 Common Examples 26](#_Toc210755854)

[9.7 Defining Your Own Functions 26](#_Toc210755855)

[9.7.1 Example with Parameters 26](#_Toc210755856)

[9.7.2 Docstrings 27](#_Toc210755857)

[9.7.3 Return Statement 27](#_Toc210755858)

[9.8 Understanding Scopes and Variables 27](#_Toc210755859)

[9.9 Using Functions with Loops 27](#_Toc210755860)

[9.9.1 Example with Loop and Function 27](#_Toc210755861)

[9.10 Modifying Data Structures with Functions 27](#_Toc210755862)

[9.10.1 Adding and Removing Elements in a List 27](#_Toc210755863)

[10 Exception Handling in Python 28](#_Toc210755864)

[10.1 Objectives 28](#_Toc210755865)

[10.2 What Are Exceptions? 28](#_Toc210755866)

[10.3 Errors vs. Exceptions 28](#_Toc210755867)

[10.4 Common Exceptions in Python 29](#_Toc210755868)

[10.4.1 ZeroDivisionError 29](#_Toc210755869)

[10.4.2 ValueError 29](#_Toc210755870)

[10.4.3 FileNotFoundError 29](#_Toc210755871)

[10.4.4 IndexError 29](#_Toc210755872)

[10.4.5 KeyError 29](#_Toc210755873)

[10.4.6 TypeError 29](#_Toc210755874)

[10.4.7 AttributeError 29](#_Toc210755875)

[10.4.8 ImportError 30](#_Toc210755876)

[10.5 Handling Exceptions 30](#_Toc210755877)

[10.5.1 How It Works 30](#_Toc210755878)

[10.5.2 Example: Division by Zero 30](#_Toc210755879)

[11 Objects and Classes in Python 30](#_Toc210755880)

[11.1 Objectives 30](#_Toc210755881)

[11.2 Introduction to Objects 30](#_Toc210755882)

[11.2.1 Characteristics of an Object 30](#_Toc210755883)

[11.2.2 Examples of Objects 31](#_Toc210755884)

[11.3 Classes and Methods 31](#_Toc210755885)

[11.4 Creating Your Own Classes 31](#_Toc210755886)

[11.4.1 Circle Class Example 31](#_Toc210755887)

[11.4.2 Rectangle Class Example 31](#_Toc210755888)

[11.4.3 Class Definition 32](#_Toc210755889)

[11.4.4 Creating Objects from a Class 32](#_Toc210755890)

[11.5 Constructors and the \_\_init\_\_ Method 32](#_Toc210755891)

[11.6 Accessing and Modifying Attributes 32](#_Toc210755892)

[11.7 Methods in Classes 32](#_Toc210755893)

[11.7.1 Example: Adding to Radius 32](#_Toc210755894)

[11.7.2 Using the Method 32](#_Toc210755895)

[11.8 Drawing Methods 33](#_Toc210755896)

[11.9 Summary 33](#_Toc210755897)

[12 Module 3 Summary: Python Programming Fundamentals 33](#_Toc210755898)

[12.1 Conditional Statements 33](#_Toc210755899)

[12.2 Loops 33](#_Toc210755900)

[12.3 Functions 34](#_Toc210755901)

[12.4 Variable Scope 34](#_Toc210755902)

[12.5 Exception Handling 34](#_Toc210755903)

[12.6 Objects and Classes 34](#_Toc210755904)

[13 Reading a File with Open() 35](#_Toc210755905)

[13.1 Introduction 35](#_Toc210755906)

[13.2 Objectives 35](#_Toc210755907)

[13.3 Plain Text Files 35](#_Toc210755908)

[13.4 Opening the File 35](#_Toc210755909)

[13.4.1 1. Using Python’s open() Function 35](#_Toc210755910)

[13.4.2 2. Using the with Statement 35](#_Toc210755911)

[13.5 Reading Operations 36](#_Toc210755912)

[13.5.1 1. Reading the Entire Content 36](#_Toc210755913)

[13.5.2 2. Reading the Content Line by Line 36](#_Toc210755914)

[13.5.3 3. Reading Specific Characters 37](#_Toc210755915)

[13.6 Conclusion 37](#_Toc210755916)

[14 Writing on a File with Open() 37](#_Toc210755917)

[14.1 Objective 37](#_Toc210755918)

[14.2 Writing to a File 38](#_Toc210755919)

[14.2.1 Code Explanation 38](#_Toc210755920)

[14.3 Writing Multiple Lines to a File Using a List and Loop 38](#_Toc210755921)

[14.3.1 Code Explanation 38](#_Toc210755922)

[14.4 Appending Data to an Existing File 39](#_Toc210755923)

[14.4.1 Code Explanation 39](#_Toc210755924)

[14.5 Copying Contents from One File to Another 39](#_Toc210755925)

[14.5.1 Code Explanation 39](#_Toc210755926)

[14.6 File Modes in Python 39](#_Toc210755927)

[14.7 Conclusion 40](#_Toc210755928)

[15 Introduction to Pandas for Data Analysis 41](#_Toc210755929)

[15.1 Objectives 41](#_Toc210755930)

[15.2 What is Pandas? 41](#_Toc210755931)

[15.2.1 Key Features of Pandas 41](#_Toc210755932)

[15.3 Importing Pandas 41](#_Toc210755933)

[15.4 Data Loading 41](#_Toc210755934)

[15.5 What is a Series? 42](#_Toc210755935)

[15.5.1 Accessing Elements in a Series 42](#_Toc210755936)

[15.5.2 Series Attributes and Methods 42](#_Toc210755937)

[15.6 What is a DataFrame? 42](#_Toc210755938)

[15.6.1 Creating DataFrames from Dictionaries 43](#_Toc210755939)

[15.6.2 Column Selection 43](#_Toc210755940)

[15.6.3 Accessing Rows 43](#_Toc210755941)

[15.6.4 Slicing 43](#_Toc210755942)

[15.6.5 Finding Unique Elements 43](#_Toc210755943)

[15.6.6 Conditional Filtering 43](#_Toc210755944)

[15.6.7 Saving DataFrames 43](#_Toc210755945)

[15.7 DataFrame Attributes and Methods 43](#_Toc210755946)

[15.8 Conclusion 44](#_Toc210755947)

[16 Beginner's Guide to NumPy 44](#_Toc210755948)

[16.1 Objectives 44](#_Toc210755949)

[16.2 What is NumPy? 44](#_Toc210755950)

[16.2.1 Key Aspects of NumPy in Python 44](#_Toc210755951)

[16.3 Installation 45](#_Toc210755952)

[16.4 Creating NumPy Arrays 45](#_Toc210755953)

[16.4.1 Creating a 1D Array 45](#_Toc210755954)

[16.4.2 Creating a 2D Array 45](#_Toc210755955)

[16.5 Array Attributes 45](#_Toc210755956)

[16.6 Indexing and Slicing 45](#_Toc210755957)

[16.7 Basic Operations 46](#_Toc210755958)

[16.7.1 Array Addition 46](#_Toc210755959)

[16.7.2 Scalar Multiplication 46](#_Toc210755960)

[16.7.3 Element-wise Multiplication (Hadamard Product) 46](#_Toc210755961)

[16.7.4 Matrix Multiplication 46](#_Toc210755962)

[16.8 Operations with NumPy 46](#_Toc210755963)

[16.9 Conclusion 47](#_Toc210755964)

[17 Some Context on APIs 47](#_Toc210755965)

[17.1 What are APIs? 47](#_Toc210755966)

[17.2 Importance of APIs 47](#_Toc210755967)

[17.2.1 Benefits of APIs 47](#_Toc210755968)

[17.3 Applications of APIs 47](#_Toc210755969)

[17.3.1 Social Media Platforms 48](#_Toc210755970)

[17.3.2 E-Commerce Websites 48](#_Toc210755971)

[17.3.3 Weather Applications 48](#_Toc210755972)

[17.3.4 Maps and Navigation Applications 48](#_Toc210755973)

[17.3.5 Payment Gateways 48](#_Toc210755974)

[17.3.6 Messaging Applications 48](#_Toc210755975)

[17.4 Conclusion 48](#_Toc210755976)

[18 Module 4 Summary: Working with Data in Python 49](#_Toc210755977)

[18.1 File Handling in Python 49](#_Toc210755978)

[18.1.1 File Modes 49](#_Toc210755979)

[18.1.2 Key Functions and Concepts 49](#_Toc210755980)

[18.2 Pandas: Data Manipulation and Analysis 49](#_Toc210755981)

[18.2.1 Importing and Aliasing 49](#_Toc210755982)

[18.2.2 Working with DataFrames 49](#_Toc210755983)

[18.3 NumPy: Numerical and Matrix Operations 49](#_Toc210755984)

[18.3.1 Key Concepts 49](#_Toc210755985)

[18.3.2 Array Attributes and Methods 50](#_Toc210755986)

[18.4 Vector and Matrix Operations in NumPy 50](#_Toc210755987)

[18.4.1 Vector Operations 50](#_Toc210755988)

[18.4.2 Visualization with Matplotlib 50](#_Toc210755989)

[18.5 Two-Dimensional Arrays in NumPy 50](#_Toc210755990)

[18.6 Summary 50](#_Toc210755991)

[19 Application Program Interfaces (APIs) 51](#_Toc210755992)

[19.1 Introduction 51](#_Toc210755993)

[19.2 What is an API? 51](#_Toc210755994)

[19.3 APIs in Python 51](#_Toc210755995)

[19.3.1 Pandas API 51](#_Toc210755996)

[19.4 REST APIs 51](#_Toc210755997)

[19.4.1 REST API Terminology 51](#_Toc210755998)

[19.4.2 HTTP and JSON Communication 52](#_Toc210755999)

[19.5 Example: PyCoinGecko API 52](#_Toc210756000)

[19.5.1 Steps to Use PyCoinGecko 52](#_Toc210756001)

[19.5.2 Data Conversion and Processing 52](#_Toc210756002)

[19.5.3 Candlestick Chart Creation 52](#_Toc210756003)

[19.6 Summary 52](#_Toc210756004)

[20 HTTP Protocol 53](#_Toc210756005)

[20.1 Introduction 53](#_Toc210756006)

[20.2 Overview of HTTP 53](#_Toc210756007)

[20.2.1 Example: Web Server Resources 53](#_Toc210756008)

[20.3 Uniform Resource Locator (URL) 53](#_Toc210756009)

[20.4 HTTP Request and Response 54](#_Toc210756010)

[20.4.1 Request Message 54](#_Toc210756011)

[20.4.2 Response Message 54](#_Toc210756012)

[20.5 HTTP Status Codes 54](#_Toc210756013)

[20.6 HTTP Methods 54](#_Toc210756014)

[20.7 Summary 55](#_Toc210756015)

[21 Working with the HTTP Protocol using the Requests Library 55](#_Toc210756016)

[21.1 Introduction 55](#_Toc210756017)

[21.2 Overview of the Requests Library 55](#_Toc210756018)

[21.3 Exploring the Response Object 56](#_Toc210756019)

[21.3.1 Checking Status Code 56](#_Toc210756020)

[21.3.2 Viewing Request Headers 56](#_Toc210756021)

[21.3.3 Viewing Request Body 56](#_Toc210756022)

[21.3.4 Viewing Response Headers 56](#_Toc210756023)

[21.3.5 Checking Encoding and Response Text 56](#_Toc210756024)

[21.4 GET Requests with Parameters 56](#_Toc210756025)

[21.4.1 Structure of a Query String 57](#_Toc210756026)

[21.4.2 Sending GET Request in Python 57](#_Toc210756027)

[21.5 POST Requests 57](#_Toc210756028)

[21.5.1 Differences Between GET and POST 57](#_Toc210756029)

[21.5.2 Comparing URLs and Bodies 58](#_Toc210756030)

[21.6 Summary 58](#_Toc210756031)

[22 Web Scraping and HTML Basics 58](#_Toc210756032)

[22.1 Objectives 58](#_Toc210756033)

[22.2 Introduction to Web Scraping 58](#_Toc210756034)

[22.3 How Web Scraping Works 58](#_Toc210756035)

[22.3.1 HTTP Request 58](#_Toc210756036)

[22.3.2 Web Page Retrieval 58](#_Toc210756037)

[22.3.3 HTML Parsing 59](#_Toc210756038)

[22.3.4 Data Extraction 59](#_Toc210756039)

[22.3.5 Data Transformation 59](#_Toc210756040)

[22.3.6 Storage 59](#_Toc210756041)

[22.3.7 Automation 59](#_Toc210756042)

[22.4 Document Tree 60](#_Toc210756043)

[22.4.1 HTML Structure 60](#_Toc210756044)

[22.4.2 Composition of an HTML Tag 60](#_Toc210756045)

[22.4.3 HTML Document Tree 60](#_Toc210756046)

[22.5 HTML Tables 60](#_Toc210756047)

[22.6 Web Scraping with Python 61](#_Toc210756048)

[22.7 Navigating the HTML Structure 62](#_Toc210756049)

[22.8 Custom Data Extraction 62](#_Toc210756050)

[22.9 Using BeautifulSoup for HTML Parsing 62](#_Toc210756051)

[22.10 Using Pandas read\_html for Table Extraction 62](#_Toc210756052)

[22.11 Conclusion 62](#_Toc210756053)

[23 Introduction to HTML for Web Scraping 62](#_Toc210756054)

[23.1 Overview 62](#_Toc210756055)

[23.2 Objectives 62](#_Toc210756056)

[23.3 Introduction to HTML for Web Scraping 63](#_Toc210756057)

[23.4 Basic HTML Structure 63](#_Toc210756058)

[23.4.1 Common HTML Elements 63](#_Toc210756059)

[23.4.2 Example 63](#_Toc210756060)

[23.5 Composition of an HTML Tag 63](#_Toc210756061)

[23.5.1 Tag Structure 63](#_Toc210756062)

[23.5.2 Example: Anchor Tag 64](#_Toc210756063)

[23.6 HTML Document Tree 64](#_Toc210756064)

[23.6.1 Parent and Child Relationships 64](#_Toc210756065)

[23.7 HTML Tables 64](#_Toc210756066)

[23.7.1 Table Elements 64](#_Toc210756067)

[23.7.2 Example Structure 64](#_Toc210756068)

[23.8 Inspecting and Extracting HTML 65](#_Toc210756069)

[23.9 Summary 65](#_Toc210756070)

[24 Web Scraping 65](#_Toc210756071)

[24.1 Overview 65](#_Toc210756072)

[24.2 Objectives 65](#_Toc210756073)

[24.3 Introduction to Web Scraping 65](#_Toc210756074)

[24.4 BeautifulSoup Object 66](#_Toc210756075)

[24.4.1 Creating a BeautifulSoup Object 66](#_Toc210756076)

[24.4.2 Tag Objects 66](#_Toc210756077)

[24.4.3 Accessing Attributes and Text 66](#_Toc210756078)

[24.5 The find\_all() Method 66](#_Toc210756079)

[24.5.1 Example: Extracting Table Data 66](#_Toc210756080)

[24.6 Scraping a Webpage with Requests and BeautifulSoup 67](#_Toc210756081)

[24.6.1 Steps: 67](#_Toc210756082)

[24.7 Summary 67](#_Toc210756083)

[25 Web Scraping: A Key Tool in Data Science 68](#_Toc210756084)

[25.1 Introduction 68](#_Toc210756085)

[25.2 Importance of Web Scraping in Data Science 68](#_Toc210756086)

[25.2.1 Data Collection 68](#_Toc210756087)

[25.2.2 Real-time Applications 68](#_Toc210756088)

[25.2.3 Machine Learning 68](#_Toc210756089)

[25.3 Web Scraping with Python 68](#_Toc210756090)

[25.3.1 BeautifulSoup 68](#_Toc210756091)

[25.3.2 Scrapy 68](#_Toc210756092)

[25.3.3 Selenium 69](#_Toc210756093)

[25.4 Applications of Web Scraping 69](#_Toc210756094)

[25.4.1 Price Comparison 69](#_Toc210756095)

[25.4.2 Email Address Gathering 69](#_Toc210756096)

[25.4.3 Social Media Scraping 69](#_Toc210756097)

[25.5 Conclusion 69](#_Toc210756098)

[26 Web Scraping Tables using Pandas 69](#_Toc210756099)

[26.1 Introduction 69](#_Toc210756100)

[26.2 Example: Extracting Largest Banks by Market Capitalization 70](#_Toc210756101)

[26.3 Limitations of Using read\_html() 71](#_Toc210756102)

[26.3.1 1. Non-Visible Tables 71](#_Toc210756103)

[26.3.2 2. Extra HTML Elements 72](#_Toc210756104)

[26.4 Example: Extracting a Table with Hyperlinks 73](#_Toc210756105)

[26.5 Comparison with BeautifulSoup 73](#_Toc210756106)

[26.6 Summary 74](#_Toc210756107)

[27 Working With Different File Formats 74](#_Toc210756108)

[27.1 Learning Objectives 74](#_Toc210756109)

[27.2 Introduction 74](#_Toc210756110)

[27.3 Understanding File Formats 74](#_Toc210756111)

[27.4 Using Pandas to Read CSV Files 74](#_Toc210756112)

[27.4.1 Reading a CSV File 74](#_Toc210756113)

[27.4.2 Adding Headers to a CSV File 75](#_Toc210756114)

[27.5 Working with JSON Files 75](#_Toc210756115)

[27.5.1 Reading a JSON File 75](#_Toc210756116)

[27.6 Working with XML Files 75](#_Toc210756117)

[27.6.1 Reading an XML File 75](#_Toc210756118)

[27.6.2 Creating a DataFrame from XML Data 75](#_Toc210756119)

[27.7 Summary 76](#_Toc210756120)

[28 Summary 76](#_Toc210756121)

[28.1 Introduction 76](#_Toc210756122)

[28.2 Using APIs in Python 76](#_Toc210756123)

[28.2.1 Pandas API 76](#_Toc210756124)

[28.3 REST APIs and HTTP Communication 76](#_Toc210756125)

[28.3.1 HTTP Protocol 77](#_Toc210756126)

[28.3.2 HTTP Methods 77](#_Toc210756127)

[28.3.3 HTTP Messages and Responses 77](#_Toc210756128)

[28.4 URL Structure 77](#_Toc210756129)

[28.5 Working with the Requests Library 77](#_Toc210756130)

[28.6 Time Series Data and Plotting 77](#_Toc210756131)

[28.7 Web Scraping in Python 77](#_Toc210756132)

[28.7.1 Understanding HTML 78](#_Toc210756133)

[28.7.2 Extracting Tabular Data 78](#_Toc210756134)

[28.7.3 Beautiful Soup Library 78](#_Toc210756135)

[28.8 Working with File Formats 78](#_Toc210756136)

[28.8.1 Accessing Different File Types 78](#_Toc210756137)

[28.9 Summary 79](#_Toc210756138)

[29 Next Steps 79](#_Toc210756139)

[29.1 Next Steps 79](#_Toc210756140)

[29.1.1 Python Project Courses 79](#_Toc210756141)

[29.2 Additional Learning Opportunities 79](#_Toc210756142)

[29.3 Professional Certificates 79](#_Toc210756143)

[29.3.1 IBM Professional Certificates 79](#_Toc210756144)

[29.3.2 IBM Specializations 80](#_Toc210756145)

Module 1

# Course Overview

Welcome to the **Python for Data Science, AI, and Development** course. After completing this course, you will:

* Possess basic knowledge of Python.
* Understand different data types.
* Use lists, tuples, dictionaries, and sets.
* Apply conditions and branching.
* Implement loops and create functions.
* Perform exception handling.
* Create and use objects.
* Read and write files.
* Collect data using APIs and web scraping.

In addition to the module labs, you will demonstrate your skills in a peer-graded project and validate your knowledge with the final quiz.

## Course Content

This course is divided into five modules. Aim 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

## Learning Assets

This course contains a variety of learning resources:

* **Videos & Readings**: Present core instruction.
* **Labs & Activities**: Provide hands-on learning experiences.
* **Discussions**: Enable peer interaction and knowledge sharing.
* **Peer-Reviewed Project**: Mimics real-world scenarios to showcase your skills.
* **Practice Quizzes**: Allow you to test your understanding.
* **Graded Quizzes**: Assess how well you have learned the concepts.

# Introduction to Python

## Users of Python

Python is widely used by professionals across different fields because of its clear and readable syntax.

### Experienced Programmers

* Can develop the same programs from other languages with less code.

### Beginners

* Python is an excellent starting language due to its large global community and wealth of documentation.

## Benefits of Using Python

Python is a powerful, high-level, general-purpose programming language. It offers:

* A large standard library with tools for:
  + Databases
  + Automation
  + Web scraping
  + Text processing
  + Image processing
  + Machine learning
  + Data analytics
* Applications across:
  + **Data Science**
  + **Artificial Intelligence and Machine Learning**
  + **Web Development**
  + **Internet of Things (IoT)**
* A strong global community and support from the Python Software Foundation.

## Popularity of Python

Python has become the most widely used and popular programming language in the data science industry:

* **2019 Kaggle Survey**: 75% of over 10,000 respondents reported using Python regularly.
* **Glassdoor (2019)**: More than 75% of data science job listings included Python.
* Over 80% of data professionals worldwide reported using Python in 2019 surveys.

### Organizations Using Python

Major organizations that rely on Python include:

* IBM
* Wikipedia
* Google
* Yahoo!
* CERN
* NASA
* Facebook
* Amazon
* Instagram
* Spotify
* Reddit

## Python in Data Science and AI

Python’s rich ecosystem of libraries supports advanced applications:

### Scientific Computing Libraries

* Pandas
* NumPy
* SciPy
* Matplotlib

### Artificial Intelligence and Machine Learning

* TensorFlow
* PyTorch
* Keras
* Scikit-learn

### Natural Language Processing

* Natural Language Toolkit (NLTK)

## Diversity and Inclusion in the Python Community

Python’s community is known for its commitment to diversity and inclusion:

### Python Software Foundation

* Implements a code of conduct to ensure safe and inclusive participation both online and offline.

### PyLadies

* An international mentorship group dedicated to supporting women in becoming active contributors and leaders in the Python open-source community.

## Summary

* Python has clear, readable syntax for both beginners and experienced developers.
* It is widely used in Data Science, AI, Web Development, and IoT.
* Python offers powerful libraries for scientific computing, machine learning, and NLP.
* The community is globally active and emphasizes diversity and inclusion.

# Getting Started with Jupyter

## Introduction

Welcome to **Getting Started with Jupyter**. After completing this module, you will be able to:

* Describe how to run, insert, and delete a cell in a notebook.
* Work with multiple notebooks.
* Present a notebook.
* Shut down a notebook session.

## Launching a Notebook

In the lab session of this module, you can launch a notebook using the Skills Network virtual environment.

* Select the check box, click **Open Tool**, and the environment will launch Jupyter Lab.
* Once the notebook opens, you can rename it by clicking **File > Rename Notebook** and entering a new name.

### Running Your First Cell

* Create a new notebook and type print("Hello World").
* Click the **Run** button to execute the cell.
* Alternatively:
  + From the main menu bar, click **Run > Run Selected Cells**.
  + Or press **Shift + Enter** as a shortcut.
* To run all cells in the notebook, select **Run All Cells**.

## Working with Cells

### Inserting a Cell

* Click the **plus (+)** symbol in the toolbar to insert a new cell.

### Deleting a Cell

* Highlight the cell, then click **Edit > Delete Cells**.
* Shortcut: Press **D** twice (**DD**) on the highlighted cell.

### Moving Cells

* You can move cells up or down as needed.

## Working with Multiple Notebooks

* Click the **plus (+)** button in the toolbar and select the file you want to open.
* Alternatively: **File > Open New Launcher** or **File > Open New Notebook**.
* Opened notebooks can be arranged side by side.

### Example

In one notebook, assign:

one = 1

two = 2

print(one + two)

This demonstrates working with multiple notebooks simultaneously.

## Presenting with Jupyter

Jupyter supports creating presentations directly from notebooks:

* Use **Markdown cells** to add titles, text, and descriptions.
* Create plots and combine them with code outputs.
* Convert cells into slides or sub-slides for presentations.

This feature allows seamless delivery of code, visualizations, and narrative as part of a project.

## Shutting Down Notebooks

When you finish working:

* Click the **stop icon** on the sidebar (second icon from the top).
* You can terminate all sessions at once or shut them down individually.
* Once shut down, you will see **“No Kernel”** in the top-right corner, confirming the notebook is inactive.
* Close the tabs after shutting down.

## Summary

In this module, you learned how to:

* Run, insert, and delete code cells.
* Work with multiple notebooks at the same time.
* Present results using Markdown and code cells.
* Shut down notebook sessions when finished.

# Module 1 Summary: Python Basics

## Data Types in Python

Python can distinguish among various data types:

* **Integers**: Whole numbers that can be positive or negative.
* **Floats**: Numbers with decimal points that represent whole or fractional values.
* **Strings**: Text data enclosed in single or double quotes, consisting of letters, digits, whitespace, or special characters.
* **Booleans**: Represent logical values, True or False.

### Typecasting

* Integers can be converted to floats and vice versa.
* Integers and floats can be converted to strings.
* Integers or floats can be converted to Booleans:
  + 0 → False
  + Non-zero values → True

## Expressions and Operations

Expressions in Python combine values and operations to produce results.

* Supports mathematical operations such as addition, subtraction, multiplication, and division.
* // performs integer division, discarding the fractional part.
* Python follows the **BODMAS** order of operations when evaluating expressions.

## Variables

Variables store and manipulate data in Python:

* The assignment operator (=) assigns values to variables.
* Reassigning a variable overrides its previous value.
* Mathematical operations can be performed on variables.
* Changing one variable affects others only if they reference the same **mutable object**.

## String Operations

Python provides powerful operations for manipulating strings:

* **String Basics**:
  + Strings are ordered sequences of characters.
  + Characters are indexed using positive and negative indices.
  + Strings support indexing, slicing, concatenation, and replication.
  + Strings are **immutable** and cannot be changed once created.
* **Escape Sequences**:
  + \n → New line
  + \t → Tab
  + \\ → Backslash
* **String Methods**:
  + Search, modify, and format strings.
  + Change case, replace characters, and find items.
  + Applying a method to a string creates and returns a new string.

## Key Takeaways

* Python supports multiple data types: integers, floats, strings, and Booleans.
* Typecasting allows conversion between data types.
* Expressions and operators enable mathematical calculations.
* Variables help store and manipulate data.
* Strings offer rich operations like slicing, formatting, and built-in methods, but remain immutable.

Module 2

# Lists and Tuples

## Compound Data Types

Lists and tuples are examples of compound data types in Python. They are key data structures used to organize and manage collections of data.

## Tuples

### Definition

* Tuples are **ordered sequences**.
* Represented as comma-separated elements enclosed in parentheses ( ).
* They can contain multiple data types (strings, integers, floats, etc.), but the variable type remains a **tuple**.

### Accessing Elements

* Elements are accessed using **indexing** with square brackets [].
* Indexing starts from 0.
* Negative indexes can be used to access elements from the end of the tuple.

### Operations on Tuples

* **Concatenation**: Tuples can be combined using the + operator.
* **Slicing**: A range of elements can be accessed using [start:end]. The end index is exclusive.
* **Length**: The len() function returns the number of elements in a tuple.

### Immutability

* Tuples are **immutable** (cannot be modified after creation).
* Assigning one tuple variable to another references the same object.
* To modify a tuple, a **new tuple** must be created.

### Functions with Tuples

* The sorted() function creates a **new sorted list** from a tuple.
* Tuples can also contain other tuples or complex objects, known as **nesting**.

### Nesting and Indexing

* Tuples can be nested (tuples within tuples).
* Standard indexing rules apply to access nested elements.
* This nesting can be visualized like a **tree structure**.
* Example: tup[2][3] -> means the fourth element of the sub-tuple at index 2 inside a tuple.

## Lists

### Definition

* Lists are **ordered sequences** represented with square brackets [ ].
* Lists are **mutable**, meaning their contents can be changed.

### Accessing Elements

* Elements are accessed using indexing, similar to tuples.
* Negative indexes are supported.
* Slicing works the same as in tuples.

### Operations on Lists

* **Concatenation**: Lists can be combined using the + operator.
* **Extend**: The .extend() method adds multiple elements to the end of a list.
* **Append**: The .append() method adds a single element.
* **Deletion**: Elements can be removed using the del command.

### Mutability

* Lists can be modified directly:
  + Changing an element by index.
  + Adding new elements using methods.
  + Deleting elements.
* **Aliasing**: Assigning one list variable to another references the same object. Changes in one affect the other. e.g. A = [1,2,3] and B = A.
* **Cloning**: To avoid aliasing, lists can be cloned so each variable references a separate object.



### Conversion

* Strings can be converted into lists using the .split() method.
* A delimiter can be passed to .split() to control how the string is divided.

## Summary

* **Tuples**: Ordered, immutable, represented by parentheses ( ).
* **Lists**: Ordered, mutable, represented by square brackets [ ].
* Both support indexing, slicing, concatenation, and nesting.
* Tuples are often used when data should remain constant, while lists are preferred when modifications are needed.

# Dictionaries

## Definition

* Dictionaries are **collections of key-value pairs** in Python.
* Represented using curly brackets { }.
* Keys act like indexes but do not have to be integers. They are usually strings and must be **immutable** and **unique**.
* Values can be immutable, mutable, and can include duplicates.

## Structure

* Each **key** is followed by a **value**, separated by a colon :.
* Each key-value pair is separated by a comma.

## Example

albums = {

"Back in Black": 1980,

"The Dark Side of the Moon": 1973,

"The Bodyguard": 1992,

"Thriller": 1982

}

* Keys: Album titles (e.g., "Back in Black").
* Values: Release years (e.g., 1980).

## Accessing Values

* Values are accessed using **square brackets** with the key.

albums["Back in Black"] # Output: 1980

albums["The Dark Side of the Moon"] # Output: 1973

## Modifying Dictionaries

* **Add Entry**: Assign a new key with a value.

albums["Graduation"] = 2007

* **Delete Entry**: Use the del command.

del albums["Thriller"]

## Checking Membership

* Use the in command to check if a key exists.

"Back in Black" in albums # Output: True

"Random Album" in albums # Output: False

## Dictionary Methods

* **Keys**: albums.keys() returns all keys.
* **Values**: albums.values() returns all values.

## Visualization

* Think of a dictionary like a **table**:
  + The first column represents **keys** (e.g., album titles).
  + The second column represents **values** (e.g., release years).

## Summary

* Dictionaries store data as key-value pairs.
* Keys must be unique and immutable.
* Values can be duplicates and may be mutable or immutable.
* They are powerful for lookups, modifications, and managing structured data.

# Sets

## Definition

* Sets are a **collection type** in Python.
* Like lists and tuples, they can contain different data types.
* **Key Characteristics**:
  + **Unordered**: They do not record element positions.
  + **Unique Elements**: Duplicate values are not allowed.

## Creating Sets

* Defined using **curly brackets { }**.
* Duplicate items are automatically removed.
* You can convert a list to a set using the set() function (**typecasting**).

my\_list = ["ACDC", "BackInBlack", "ACDC"]

my\_set = set(my\_list)

print(my\_set) # Output: {"ACDC", "BackInBlack"}

## Basic Set Operations

### Adding Elements

* Use the add() method.

A = {"Thriller", "ACDC"}

A.add("inSync")

* Adding the same item again has no effect (no duplicates).

### Removing Elements

* Use the remove() method.

A.remove("inSync")

### Membership Test

* Use the in keyword.

"ACDC" in A # Output: True

"Who" in A # Output: False

## Mathematical Set Operations

### Intersection

* The intersection of two sets contains only the elements present in **both sets**.

AlbumSet1 = {"ACDC", "BackInBlack", "Thriller"}

AlbumSet2 = {"ACDC", "BackInBlack", "The Bodyguard"}

AlbumSet3 = AlbumSet1 & AlbumSet2

print(AlbumSet3) # Output: {"ACDC", "BackInBlack"}

### Union

* The union of two sets contains **all unique elements** from both sets.

AlbumSet1 | AlbumSet2

### Subset

* Check if a set is a subset of another using issubset().

AlbumSet3.issubset(AlbumSet1) # Output: True

## Visualization with Venn Diagrams

* **Sets** can be represented as circles.
* **Intersection**: Overlapping area.
* **Union**: A Combination of both circles.
* **Subset**: One circle fully inside another.

## Summary

* Sets are unordered collections of unique elements.
* They support operations like **add**, **remove**, and **membership checks**.
* Mathematical operations include **intersection**, **union**, and **subset checking**.
* Useful for managing collections with no duplicates and for mathematical computations.

# Module 2 Summary: Python Data Structures

## Tuples

* Ordered and immutable collections of elements.
* Defined using **parentheses ()** with comma-separated values.
* Can include strings, integers, floats, and even nested tuples.
* Access elements using positive and negative indexing.
* Operations include combining, concatenating, and slicing.
* Since tuples are immutable, new tuples must be created for modifications.

## Lists

* Ordered and mutable collections of items.
* Defined using **square brackets []**.
* Can contain mixed data types such as strings, integers, floats, and nested lists.
* Elements are accessed using positive and negative indexing.
* Support operations like adding, deleting, splitting, concatenating, and appending.
* Aliasing occurs when multiple names refer to the same list object.
* Lists can be cloned to create independent copies.

## Dictionaries

* Store **key-value pairs** for flexible data retrieval.
* Defined using **curly brackets {}**.
* Keys must be immutable and unique, while values can be mutable and allow duplicates.
* Key-value pairs are separated by commas.
* Access values using keys.
* Support operations like adding, deleting, and checking for the existence of keys (returns True/False).
* Provide methods to retrieve lists of keys and values.

## Sets

* Unordered collections of **unique elements**.
* Defined using **curly brackets {}**.
* Automatically remove duplicate items.
* A list passed through the set() function generates a set of unique elements.
* Support set operations such as **adding, removing, and membership checks**.
* Mathematical operations include:
  + **Intersection (&)** → Common elements between sets.
  + **Union (|)** → All unique elements from both sets.
  + **Subset check (issubset())** → Determine if one set is contained within another.

## Summary

* **Tuples**: Ordered, immutable, suitable for fixed collections.
* **Lists**: Ordered, mutable, versatile for dynamic collections.
* **Dictionaries**: Key-value storage with unique keys and flexible values.
* **Sets**: Unordered, unique elements, ideal for duplicate removal and set operations.

Module 3

# Exploring Python Functions

## Objectives

After reading this, you should be able to:

* Describe the function concept and the importance of functions in programming
* Write a function that takes inputs and performs tasks
* Use built-in functions like len(), sum(), and others effectively
* Define and use your own functions in Python
* Differentiate between global and local variable scopes
* Use loops within a function
* Modify data structures using functions

## Introduction to Functions

A function is a fundamental building block that encapsulates specific actions or computations. Similar to mathematics, functions in programming take inputs, perform operations, and return outputs.

## Purpose of Functions

Functions promote **modularity** and **reusability**. Instead of duplicating code, you can define a function once and call it wherever needed, making code cleaner and easier to maintain.

## Benefits of Using Functions

* **Modularity**: Break complex tasks into manageable parts
* **Reusability**: Use functions multiple times without rewriting code
* **Readability**: Enhance understanding with meaningful function names
* **Debugging**: Isolated functions simplify troubleshooting
* **Abstraction**: Simplify complex processes with easy-to-use interfaces
* **Collaboration**: Teams can work on separate functions concurrently
* **Maintenance**: Changes in a function reflect everywhere it is used

## How Functions Work

### Inputs (Parameters)

Functions receive inputs (parameters) that allow them to work with specific data.

### Performing Tasks

Functions execute predefined tasks such as calculations, operations on data, formatting, or fetching information.

### Producing Outputs

Functions return results (outputs) that can be reused, stored in variables, or passed to other functions.

**Example:**

def calculate\_total(a, b): # Parameters: a and b

total = a + b # Task: Addition

return total # Output: Sum

result = calculate\_total(5, 7)

print(result) # Output: 12

## Python’s Built-in Functions

Python offers many built-in functions that simplify coding tasks.

### Common Examples

# len()

string\_length = len("Hello, World!") # Output: 13

list\_length = len([1, 2, 3, 4, 5]) # Output: 5

# sum()

total = sum([10, 20, 30, 40, 50]) # Output: 150

# max()

highest = max([5, 12, 8, 23, 16]) # Output: 23

# min()

lowest = min([5, 12, 8, 23, 16]) # Output: 5

## Defining Your Own Functions

def function\_name():

pass

* pass is a placeholder statement to maintain correct syntax until code is added.

### Example with Parameters

def greet(name):

return "Hello, " + name

print(greet("Alice")) # Output: Hello, Alice

### Docstrings

def multiply(a, b):

"""

This function multiplies two numbers.

Input: a (number), b (number)

Output: Product of a and b

"""

print(a \* b)

multiply(2, 6) # Output: 12

### Return Statement

def add(a, b):

return a + b

sum\_result = add(3, 5)

print(sum\_result) # Output: 8

## Understanding Scopes and Variables

* **Global Scope**: Variables defined outside functions (accessible everywhere)
* **Local Scope**: Variables defined inside a function (accessible only inside it)

global\_variable = "I'm global"

def example\_function():

local\_variable = "I'm local"

print(global\_variable) # Access global

print(local\_variable) # Access local

example\_function()

print(global\_variable) # Works

# print(local\_variable) # Error

## Using Functions with Loops

Functions can contain loops, making tasks more organized and reusable.

def print\_numbers(limit):

for i in range(1, limit + 1):

print(i)

print\_numbers(5) # Output: 1 2 3 4 5

### Example with Loop and Function

def greet(name):

return "Hello, " + name

for \_ in range(3):

print(greet("Alice"))

## Modifying Data Structures with Functions

### Adding and Removing Elements in a List

# Initial list

my\_list = []

# Function to add elements

def add\_element(data\_structure, element):

data\_structure.append(element)

# Function to remove elements

def remove\_element(data\_structure, element):

if element in data\_structure:

data\_structure.remove(element)

else:

print(f"{element} not found in the list.")

# Add elements

add\_element(my\_list, 42)

add\_element(my\_list, 17)

add\_element(my\_list, 99)

print("Current list:", my\_list)

# Remove elements

remove\_element(my\_list, 17)

remove\_element(my\_list, 55) # Not found

print("Updated list:", my\_list)

# Exception Handling in Python

## Objectives

By the end of this reading, you should be able to:

* Understand exceptions
* Distinguish errors from exceptions
* Recognize common Python exceptions
* Manage exceptions effectively

## What Are Exceptions?

Exceptions are alerts that occur when something unexpected happens while running a program. They could result from coding mistakes or unplanned situations. Python can raise exceptions automatically, but we can also trigger them manually using the raise statement. By handling exceptions, we can prevent programs from crashing.

## Errors vs. Exceptions

Errors and exceptions differ in their severity and handling:

|  |  |  |
| --- | --- | --- |
| Aspect | Errors | Exceptions |
| Origin | Caused by environment, hardware, or operating system | Caused by problematic code execution within the program |
| Nature | Severe, often causing program crashes | Less severe, can usually be fixed so the program continues |
| Handling | Not usually caught or handled by the program | Can be caught using try-except blocks |
| Examples | SyntaxError (incorrect syntax), NameError (undefined variable) | ZeroDivisionError, FileNotFoundError |
| Categorization | Not classified | Categorized into classes (ArithmeticError, IOError, ValueError, etc.) |

## Common Exceptions in Python

Here are some commonly encountered exceptions:

### ZeroDivisionError

Occurs when dividing a number by zero.

result = 10 / 0 # Raises ZeroDivisionError

### ValueError

Occurs when inappropriate values are used, e.g., converting a non-numeric string to an integer.

num = int("abc") # Raises ValueError

### FileNotFoundError

Occurs when trying to access a non-existent file.

with open("nonexistent\_file.txt", "r") as file:

content = file.read() # Raises FileNotFoundError

### IndexError

Occurs when accessing a list index out of range.

my\_list = [1, 2, 3]

missing = my\_list[5] # Raises IndexError

### KeyError

Occurs when accessing a non-existent dictionary key.

my\_dict = {"name": "Alice", "age": 30}

missing = my\_dict["city"] # Raises KeyError

### TypeError

Occurs when using an object in an incompatible manner.

result = "hello" + 5 # Raises TypeError

### AttributeError

Occurs when accessing an attribute or method that does not exist.

text = "example"

missing = text.some\_method() # Raises AttributeError

### ImportError

Occurs when attempting to import a non-existent module.

import non\_existent\_module # Raises ImportError

**Note**: These are just a few examples; Python has many exceptions. With proper handling, you can manage them effectively.

## Handling Exceptions

Python uses **try-except** blocks to handle exceptions and prevent crashes.

### How It Works

* Code that might raise an exception is placed in the try block.
* If an exception occurs, control jumps to the except block.
* The except block defines how to handle the exception gracefully.
* The program continues execution after handling the exception.

### Example: Division by Zero

try:

result = 10 / 0

except ZeroDivisionError:

print("Error: Cannot divide by zero")

print("Outside of try and except block")

# Objects and Classes in Python

**Estimated time needed: 20 minutes**

## Objectives

By the end of this reading, you should be able to:

* Understand objects and classes in Python
* Identify data attributes and methods
* Create your own classes and objects
* Work with constructors and methods
* Recognize how to modify and interact with objects

## Introduction to Objects

Python has many different data types: integers, floats, strings, lists, dictionaries, and Booleans. In Python, each is an **object**.

### Characteristics of an Object

Every object has:

* A **type**
* An **internal representation**
* A set of **functions (methods)** to interact with the data

An **object** is an instance of a particular type. For example:

* Each integer you create is an integer object.
* Each list you create is a list object.

We can find out the type of an object by using the type() command.

### Examples of Objects

* Integer object
* List object
* String object
* Dictionary object

## Classes and Methods

A **class** is a blueprint for creating objects. It defines:

* **Data attributes/State**: characteristics of the object
* **Methods/Behavior**: functions that interact with the object

We have already used methods, such as list.sort() or list.reverse(), which change the state of the list object.

## Creating Your Own Classes

You can create your own classes in Python. A class defines the structure of objects:

### Circle Class Example

Data attributes:

* radius
* color

### Rectangle Class Example

Data attributes:

* color
* height
* width

### Class Definition

To create a class in Python:

class Circle(object):

def \_\_init\_\_(self, radius, color):

self.radius = radius

self.color = color

### Creating Objects from a Class

circle1 = Circle(4, "red")

circle2 = Circle(2, "green")

Similarly, you can create rectangle objects with height, width, and color.

## Constructors and the \_\_init\_\_ Method

The \_\_init\_\_ method is a **constructor**. It initializes the object when it is created.

* self refers to the instance of the class.
* Parameters (e.g., radius, color) initialize the object’s data attributes.

class Rectangle(object):

def \_\_init\_\_(self, height, width, color):

self.height = height

self.width = width

self.color = color

## Accessing and Modifying Attributes

You can access attributes with the dot operator:

print(circle1.radius)

print(circle1.color)

You can also modify them directly:

circle1.color = "blue"

## Methods in Classes

Methods are functions inside a class that interact with its data attributes.

### Example: Adding to Radius

class Circle(object):

def \_\_init\_\_(self, radius, color):

self.radius = radius

self.color = color

def add\_radius(self, r):

self.radius = self.radius + r

### Using the Method

circle = Circle(2, "red")

circle.add\_radius(8)

print(circle.radius) # Output: 10

## Drawing Methods

You can also define methods like drawCircle or drawRectangle to visualize objects. (See lab exercises for implementation.)

## Summary

* Objects are **instances of classes**.
* Classes define **data attributes** and **methods**.
* The \_\_init\_\_ method is the constructor used to initialize object attributes.
* Methods allow interaction with and modification of object data.
* You can use the dir() function to inspect available attributes and methods of an object.

For more advanced details, visit [python.org](https://www.python.org/).

# Module 3 Summary: Python Programming Fundamentals

## Conditional Statements

* Python conditions use **if statements** to execute code based on true/false conditions created by comparisons and Boolean expressions.
* **Comparison operators** include:
  + == (equal to)
  + > (greater than)
  + < (less than)
  + != (not equal to)
* You can compare **integers, strings, and floats**.
* Python branching uses if, else, and elif to direct program flow and execute different code blocks.
* if defines actions when the condition is true.
* else defines actions when all previous conditions are false.
* elif provides additional checks only if the initial condition is false.
* Boolean logic operators are used to perform operations on Boolean values.

## Loops

* Loops are control structures that automate repetitive tasks and iterate over data structures.
* The range() function generates a sequence of numbers with a start, stop, and step value.
* **For loop** iterates over sequences like lists, tuples, or strings, executing code for each item.
* **While loop** executes a block of code as long as the condition remains true.

## Functions

* Functions are reusable code blocks that perform tasks, accept inputs, and often return results.
* Python has many **built-in functions**, such as:
  + len() to find the length of a sequence
  + sum() to calculate the sum of a sequence
  + sorted() to return a sorted list
  + sort() to sort items in the original list
* You can also **create your own functions**.
* Functions should include **documentation strings** (docstrings) to ensure clarity and maintainability.
* The help() command displays documentation for a function.
* Functions can have multiple parameters.
* If no return statement is used, the function returns **None** by default.
* The pass keyword can be used as a placeholder.
* Functions usually perform multiple tasks.

## Variable Scope

* The scope of a variable defines where it can be accessed or modified.
* **Local scope**: variables defined inside a block or function, accessible only within it.
* **Global scope**: variables defined at the top level, accessible throughout the program.

## Exception Handling

* Exception handling prevents errors from crashing a program.
* try-except is used to handle errors.
* try-except-else adds an else block that runs when no exceptions occur.
* try-except-else-finally ensures the finally block always runs, regardless of exceptions.

## Objects and Classes

* Objects are **instances of classes** that encapsulate data and behavior.
* The type() function determines the type of an object.
* **Classes** are blueprints for creating objects, defining attributes and methods.
* **Methods** can modify an object’s state while maintaining its type.
* The \_\_init\_\_ method initializes object attributes.
* Instances of a class can be created with specific attributes.
* **Data attributes** define the data of an object.
* **Methods** are functions that interact with and modify attributes.
* Methods require self as the first parameter, along with other parameters.

Module 4

# Reading a File with Open()

**Estimated Time Needed:** 10 minutes

## Introduction

File handling is an essential aspect of programming, and Python provides built-in functions to interact with files. This guide explores how to use Python's open() function to read text files (.txt files).

## Objectives

* Describe how to use the open() and read() Python functions to open and read the contents of a text file.
* Explain how to use the with statement in Python.
* Describe how to use the readline() function in Python.
* Explain how to use the seek() function to read specific character(s) in a text file.

## Plain Text Files

Plain text files contain unformatted text without any specific structure. You can read these files line by line or load all the content into memory.

## Opening the File

There are two primary methods to open a file in Python.

### 1. Using Python’s open() Function

Suppose we have a file named file.txt. The open() function creates a file object and allows access to the file’s contents. It takes two key parameters:

* **File Path:** The file name and its directory.
* **Mode:** Specifies the purpose of opening the file (e.g., 'r' for reading, 'w' for writing, 'a' for appending).

# Open the file in read ('r') mode

file = open('file.txt', 'r')

This line opens file.txt in read mode and returns a file object stored in the variable file.

### 2. Using the with Statement

The with statement simplifies file handling by automatically closing the file when operations within its block are completed.

# Open the file using 'with' in read ('r') mode

with open('file.txt', 'r') as file:

# further code

The with statement ensures the file is properly closed after operations, even if an exception occurs.

#### Advantages of Using the with Statement

* **Automatic Resource Management:** The file closes automatically when exiting the with block.
* **Cleaner Code:** No need to explicitly call close(), making code concise and less error-prone.

**Note:** For most file reading and writing operations, using the with statement is best practice.

## Reading Operations

### 1. Reading the Entire Content

You can read the entire content of a file using the read() method. The data is stored as a string in a variable.

# Reading and Storing the Entire Content of a File

with open('file.txt', 'r') as file:

file\_stuff = file.read()

print(file\_stuff)

**Step-by-Step Explanation:**

1. **Open the File:** Open file.txt in read mode using with.
2. **Read Content:** Use read() to get the entire file content.
3. **Process Content:** The file’s data is now in file\_stuff; you can display or manipulate it.
4. **Automatic Closure:** The file automatically closes after the block ends.

### 2. Reading the Content Line by Line

Python provides multiple methods for reading files line by line.

#### Using readlines()

Reads the entire file and stores each line as an element in a list.

#### Using readline()

Reads one line at a time and can be called repeatedly.

file = open('file.txt', 'r')

line1 = file.readline()

line2 = file.readline()

print(line1)

if 'important' in line2:

print('This line is important!')

while True:

line = file.readline()

if not line:

break

print(line)

file.close()

**Explanation:**

* Each call to readline() reads the next line.
* The loop continues until there are no more lines.
* Always close the file using close() when finished.

### 3. Reading Specific Characters

You can specify how many characters to read using the read() method.

#### Steps:

1. **Open the File:**

file = open('file.txt', 'r')

1. **Navigate to a Position (Optional):**  
   Use seek() to move the file pointer to a specific position.

file.seek(10) # Moves to the 11th byte

1. **Read Characters:**

characters = file.read(5) # Reads 5 characters

print(characters)

1. **Close the File:**

file.close()

## Conclusion

In conclusion, file handling is a fundamental aspect of programming. Python’s built-in functions like open(), read(), readline(), and seek() make file operations simple and efficient. The use of the with statement ensures proper resource management, making it the recommended approach for file operations.

# Writing on a File with Open()

## Objective

* Create and write data to a file in Python.
* Write multiple lines of text to a file using lists and loops.
* Add new information to an existing file without erasing its content.
* Compare and contrast different file modes in Python, understanding their meanings and use cases.

## Writing to a File

You can create a new text file and write data to it using Python’s open() function. The open() function takes two primary arguments:

* **File path:** The name and directory of the file.
* **Mode:** Specifies the operation you want to perform on the file.

For writing, use the mode 'w'. Here's an example:

# Create a new file Example2.txt for writing

with open('Example2.txt', 'w') as file1:

file1.write("This is line A\n")

file1.write("This is line B\n")

# file1 is automatically closed when the 'with' block exits

### Code Explanation

* **Line 2:** The open() function opens or creates a file named Example2.txt for writing ('w' mode). The with statement ensures the file is automatically closed when the block exits.
* **Line 3:** The write() method writes the text This is line A followed by a newline (\n).
* **Line 4:** Writes This is line B to a new line in the same file.

## Writing Multiple Lines to a File Using a List and Loop

You can store multiple lines of text in a list and write them to a file using a loop.

# List of lines to write to the file

Lines = ["This is line 1", "This is line 2", "This is line 3"]

# Create a new file Example3.txt for writing

with open('Example3.txt', 'w') as file2:

for line in Lines:

file2.write(line + "\n")

# file2 is automatically closed when the 'with' block exits

### Code Explanation

* **Line 2:** A list named Lines stores multiple strings to be written to the file.
* **Line 5:** Opens Example3.txt for writing using 'w' mode.
* **Line 6–7:** Iterates over each line in the list and writes it to the file, appending a newline character.
* **Line 8:** The file closes automatically at the end of the with block.

## Appending Data to an Existing File

You can use the 'a' mode to append new data to an existing file without overwriting its contents.

# Data to append to the existing file

new\_data = "This is line C"

# Open an existing file Example2.txt for appending

with open('Example2.txt', 'a') as file1:

file1.write(new\_data + "\n")

# file1 is automatically closed when the 'with' block exits

### Code Explanation

* **Line 2:** Defines the variable new\_data containing the text to append.
* **Line 5:** Opens Example2.txt in 'a' mode. If the file doesn’t exist, it is created.
* **Line 6:** Appends the new data to the file, adding a newline at the end.
* **Line 7:** The file automatically closes after exiting the with block.

## Copying Contents from One File to Another

You can copy the contents of one file to another by reading from the source file and writing to the destination file.

# Open the source file for reading

with open('source.txt', 'r') as source\_file:

# Open the destination file for writing

with open('destination.txt', 'w') as destination\_file:

# Read lines from the source file and copy them to the destination file

for line in source\_file:

destination\_file.write(line)

# Destination file is automatically closed when the 'with' block exits

# Source file is automatically closed when the 'with' block exits

### Code Explanation

* **Line 2:** Opens source.txt in read ('r') mode.
* **Line 4:** Opens destination.txt in write ('w') mode.
* **Line 6–7:** Loops through each line of the source file and writes it to the destination file.
* **Line 8–9:** Both files close automatically when their respective with blocks exit.

## File Modes in Python

The following table lists different file modes, their syntax, and common use cases:

| **Mode** | **Syntax** | **Description** |
| --- | --- | --- |
| Read | 'r' | Opens an existing file for reading. Raises an error if the file doesn’t exist. |
| Write | 'w' | Creates a new file for writing. Overwrites the file if it already exists. |
| Append | 'a' | Opens a file for appending data. Creates the file if it doesn’t exist. |
| Exclusive | 'x' | Creates a new file for writing. Raises an error if the file already exists. |
| Read Binary | 'rb' | Opens a binary file for reading. |
| Write Binary | 'wb' | Creates a binary file for writing. |
| Append Binary | 'ab' | Opens a binary file for appending data. |
| Exclusive Binary | 'xb' | Creates a binary file for writing. Raises an error if it already exists. |
| Read Text | 'rt' | Opens a text file for reading (default). |
| Write Text | 'wt' | Creates a new text file for writing. |
| Append Text | 'at' | Opens a text file for appending data. |
| Exclusive Text | 'xt' | Creates a text file for writing but raises an error if it already exists. |
| Read & Write | 'r+' | Opens an existing file for both reading and writing. |
| Write & Read | 'w+' | Creates a new file for reading and writing. Overwrites if it already exists. |
| Append & Read | 'a+' | Opens a file for both appending and reading. Creates the file if it doesn’t exist. |
| Exclusive Read & Write | 'x+' | Creates a new file for reading and writing. Raises an error if it already exists. |

## Conclusion

Working with files is a fundamental part of programming. Python provides flexible tools like open(), write(), and with statements to create, write, append, and copy files efficiently. Understanding file modes ensures proper use of resources and data management in any Python project.

# Introduction to Pandas for Data Analysis

## Objectives

* Learn what Pandas Series are and how to create them.
* Understand how to access and manipulate data within a Series.
* Discover the basics of creating and working with Pandas DataFrames.
* Learn how to access, modify, and analyze data in DataFrames.
* Gain insights into common DataFrame attributes and methods.

## What is Pandas?

Pandas is a popular open-source data manipulation and analysis library for Python. It provides a powerful and flexible set of tools for working with structured data, making it a fundamental tool for data scientists, analysts, and engineers.

Pandas can handle data in various formats, such as tabular and time-series data, making it an essential part of the data processing workflow in many industries.

### Key Features of Pandas

* **Data Structures:** Pandas offers two main data structures:
  + **DataFrame:** A two-dimensional, size-mutable table with labeled axes (rows and columns).
  + **Series:** A one-dimensional labeled array, representing a single column or row of data.
* **Data Import and Export:** Easily read and write data from sources like CSV, Excel, SQL, and more.
* **Data Merging and Joining:** Combine multiple DataFrames using methods like merge and join, similar to SQL operations.
* **Efficient Indexing:** Quickly access specific rows and columns using indexing and selection methods.
* **Custom Data Structures:** Create and manipulate data in customized ways to fit specific needs.

## Importing Pandas

To use Pandas, you must first import it into your Python environment. It is commonly imported using the alias pd:

import pandas as pd

## Data Loading

Pandas allows you to load data from various sources, such as CSV and Excel files. Use the read\_csv() function to load CSV data into a DataFrame:

import pandas as pd

# Read the CSV file into a DataFrame

df = pd.read\_csv('your\_file.csv')

Replace 'your\_file.csv' with the actual file path. Ensure the file is located in your working directory or provide the correct path.

## What is a Series?

A **Series** is a one-dimensional labeled array in Pandas. It can be thought of as a single column of data with labels (indices) for each element. A Series can be created from lists, NumPy arrays, or dictionaries.

import pandas as pd

# Create a Series from a list

data = [10, 20, 30, 40, 50]

s = pd.Series(data)

print(s)

Pandas automatically assigns numerical indices (0, 1, 2, 3, 4), but you can specify custom labels if needed.

### Accessing Elements in a Series

You can access Series elements using index labels or integer positions.

print(s[2]) # Access by label (value 30)

print(s.iloc[3]) # Access by position (value 40)

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

### Series Attributes and Methods

Some useful attributes and methods of a Series include:

* values – Returns data as a NumPy array.
* index – Returns index labels.
* shape, size – Returns dimensions and size.
* mean(), sum(), min(), max() – Summary statistics.
* unique(), nunique() – Unique values and their count.
* sort\_values(), sort\_index() – Sort by values or index.
* isnull(), notnull() – Detect missing values.
* apply() – Apply custom functions to elements.

## What is a DataFrame?

A **DataFrame** is a two-dimensional labeled data structure with columns of potentially different data types. Think of it as a table where each column represents a variable and each row represents an observation.

### Creating DataFrames from Dictionaries

You can create a DataFrame from a dictionary, where keys are column labels and values are lists representing data.

import pandas as pd

# Create a DataFrame from a dictionary

data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],

'Age': [25, 30, 35, 28],

'City': ['New York', 'San Francisco', 'Los Angeles', 'Chicago']}

df = pd.DataFrame(data)

print(df)

### Column Selection

Select a single column using its name or multiple columns by passing a list:

print(df['Name'])

print(df[['Name', 'Age']])

### Accessing Rows

Access rows by position or label using .iloc[] and .loc[]:

print(df.iloc[2]) # Third row by position

print(df.loc[1]) # Second row by label

### Slicing

Slice DataFrames to select specific rows or columns:

print(df[['Name', 'Age']]) # Select specific columns

print(df[1:3]) # Select specific rows

### Finding Unique Elements

Find unique elements in a column:

unique\_ages = df['Age'].unique()

### Conditional Filtering

Filter data using conditional expressions:

high\_above\_25 = df[df['Age'] > 25]

### Saving DataFrames

Save a DataFrame to a CSV file using to\_csv():

df.to\_csv('people\_data.csv', index=False)

## DataFrame Attributes and Methods

Key attributes and methods for DataFrames include:

* shape – Dimensions of the DataFrame.
* info() – Summary including data types and null counts.
* describe() – Statistical summary of numerical columns.
* head(), tail() – View first or last n rows.
* mean(), sum(), min(), max() – Column-wise summary statistics.
* sort\_values() – Sort DataFrame by columns.
* groupby() – Group data for aggregation.
* fillna(), drop(), rename() – Handle missing data, drop, or rename columns.
* apply() – Apply custom functions.

For more details, refer to the official Pandas documentation.

## Conclusion

Mastering Pandas Series and DataFrames is essential for effective data analysis in Python. Series handle one-dimensional labeled data, while DataFrames provide a flexible, table-like structure for two-dimensional data.

By using Pandas, you can efficiently clean, explore, and analyze datasets. Practice with real data and consult Pandas documentation to deepen your understanding and enhance your data analysis skills.

# Beginner's Guide to NumPy

## Objectives

In this reading, we will learn:

* Basics of NumPy
* How to create NumPy arrays
* Array attributes and indexing
* Basic operations like addition and multiplication

## What is NumPy?

NumPy, short for **Numerical Python**, is a fundamental library for numerical and scientific computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a wide range of mathematical functions to operate on these arrays.

NumPy serves as the foundation for many data science and machine learning libraries, making it an essential tool for data analysis and scientific research.

### Key Aspects of NumPy in Python

* **Efficient Data Structures:** NumPy arrays are faster and more memory-efficient than Python lists.
* **Multi-Dimensional Arrays:** Enables representation of matrices and tensors, useful in scientific computing.
* **Element-wise Operations:** Simplifies mathematical operations on entire datasets.
* **Random Number Generation:** Offers tools for simulations and statistical analysis.
* **Integration with Other Libraries:** Works seamlessly with libraries like SciPy, Pandas, and Matplotlib.
* **Performance Optimization:** Built using C and Fortran for high-speed computation.

## Installation

To install NumPy, use the following command:

pip install numpy

## Creating NumPy Arrays

You can create NumPy arrays from Python lists. Arrays can be one-dimensional or multi-dimensional.

### Creating a 1D Array

import numpy as np

# Creating a 1D array

arr\_1d = np.array([1, 2, 3, 4, 5])

Here, a one-dimensional array arr\_1d is created from a Python list. It contains five elements: 1, 2, 3, 4, and 5.

### Creating a 2D Array

import numpy as np

# Creating a 2D array

arr\_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

This creates a two-dimensional array arr\_2d representing a 3x3 matrix with elements from 1 to 9.

## Array Attributes

NumPy arrays provide several useful attributes:

print(arr\_2d.ndim) # Number of dimensions

print(arr\_2d.shape) # Shape (rows, columns)

print(arr\_2d.size) # Total number of elements

**Output:**

2

(3, 3)

9

## Indexing and Slicing

Access array elements using indexing and slicing:

# Indexing and slicing

print(arr\_1d[2]) # Access the 3rd element

print(arr\_2d[1, 2]) # Access element at 2nd row, 3rd column

print(arr\_2d[1]) # Access the 2nd row

print(arr\_2d[:, 1]) # Access the 2nd column

## Basic Operations

NumPy allows element-wise operations such as addition, subtraction, multiplication, and division.

### Array Addition

array1 = np.array([1, 2, 3])

array2 = np.array([4, 5, 6])

result = array1 + array2

print(result) # [5 7 9]

### Scalar Multiplication

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

result = array \* 2

print(result) # [2 4 6]

### Element-wise Multiplication (Hadamard Product)

array1 = np.array([1, 2, 3])

array2 = np.array([4, 5, 6])

result = array1 \* array2

print(result) # [4 10 18]

### Matrix Multiplication

matrix1 = np.array([[1, 2], [3, 4]])

matrix2 = np.array([[5, 6], [7, 8]])

result = np.dot(matrix1, matrix2)

print(result)

# [[19 22]

# [43 50]]

## Operations with 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, etc. | result = arr1 + arr2 |
| Scalar Arithmetic | Addition, subtraction with scalars | result = arr \* 2 |
| Element-Wise Functions | Applying functions to each element | result = np.sqrt(arr) |
| Sum and Mean | Calculating sum and mean | total = np.sum(arr) average = np.mean(arr) |
| Maximum and Minimum Values | Finding max and min | max\_val = np.max(arr) min\_val = np.min(arr) |
| Reshaping | Changing array shape | reshaped\_arr = arr.reshape(2, 3) |
| Transposition | Transposing arrays | transposed\_arr = arr.T |
| Matrix Multiplication | Performing matrix multiplication | result = np.dot(matrix1, matrix2) |

## Conclusion

NumPy is a powerful and essential library for numerical and scientific computing in Python. It simplifies data handling with efficient arrays and mathematical operations. This guide introduced NumPy basics, array creation, indexing, and core operations. To explore further, visit [numpy.org](https://numpy.org/) for more tutorials and examples.

# Some Context on APIs

## What are APIs?

APIs, or **Application Programming Interfaces**, are a crucial part of software development. They enable developers to build new applications by utilizing existing functionalities from other systems. APIs define how software components interact, facilitating communication between different products and services without requiring direct implementation.

## Importance of APIs

APIs are essential for developers because they allow access to data and functionality from other systems, saving both time and resources.

### Benefits of APIs

* Enable integration of applications into existing architectures.
* Facilitate communication between products and services without direct implementation.
* Allow developers to create new applications leveraging existing systems.
* Streamline the engineering and development process.

APIs are widely used across various domains, including social media, e-commerce, web, mobile, and desktop applications.

## Applications of APIs

APIs serve multiple purposes across different industries and platforms.

### Social Media Platforms

Platforms like **Facebook**, **Twitter**, and **Instagram** use APIs to let developers access their data and functionality. This allows the creation of apps that enhance user experience and interaction with these platforms.

### E-Commerce Websites

E-commerce giants such as **Amazon** and **eBay** use APIs to provide access to product catalogs and transactional data. Developers can build applications that interact with these platforms for enhanced shopping features.

### Weather Applications

Weather services like **AccuWeather** and **The Weather Channel** use APIs to distribute real-time weather data. Developers can integrate this data to provide users with up-to-date weather information.

### Maps and Navigation Applications

Applications like **Google Maps** and **Waze** use APIs to access geolocation data and navigation details. Developers can integrate these APIs to offer directions, traffic updates, and location-based services.

### Payment Gateways

Payment services such as **PayPal** and **Stripe** provide APIs that allow secure payment processing. Developers can integrate these APIs into their apps to handle transactions efficiently and safely.

### Messaging Applications

Messaging services like **WhatsApp** and **Facebook Messenger** use APIs to provide access to their messaging capabilities. Developers can use these APIs to build apps that extend messaging functionalities.

## Conclusion

APIs are a fundamental part of modern software development. They enable access to external data and services, enhance functionality, and promote efficiency in development. By leveraging APIs, developers can create powerful, interconnected applications across diverse domains.

# Module 4 Summary: Working with Data in Python

## File Handling in Python

Python provides the open() function to read and write files, allowing developers to access and manipulate file contents effectively.

### File Modes

* **r (read):** Opens the file for reading.
* **w (write):** Opens the file for writing, overwriting existing content.
* **a (append):** Opens the file for appending data without deleting existing content.

### Key Functions and Concepts

* open() function: Used to open files for reading or writing.
* with open() statement: Ensures files are properly opened and closed.
* "\n" character: Starts a new line within a file.
* Various methods exist to print and process lines from file attributes.

## Pandas: Data Manipulation and Analysis

Pandas is a powerful Python library for data manipulation and analysis, offering data structures and tools for working with structured data such as DataFrames and Series.

### Importing and Aliasing

* Import Pandas using the command: import pandas as pd.
* The as keyword is used to provide a shorter alias for easier access.

### Working with DataFrames

* **DataFrame:** A two-dimensional structure consisting of rows and columns.
* You can create new DataFrames using columns from existing ones.
* DataFrames allow reading, manipulating, and saving data in multiple formats.
* The unique() method identifies unique elements in a DataFrame column.
* Inequality operators with df assign Boolean values for conditional filtering.
* You can create new DataFrames derived from existing ones containing filtered values.

## NumPy: Numerical and Matrix Operations

NumPy is a foundational Python library for numerical computations and matrix operations. It provides multidimensional array objects and mathematical functions for efficient data handling.

### Key Concepts

* **NumPy as a Foundation:** Pandas is built on top of NumPy.
* **Array (ndarray):** A fixed-size collection of elements of the same data type.
* **1D Array:** A linear sequence of elements optimized for numerical operations.
* Elements are accessed via indexing.

### Array Attributes and Methods

* dtype: Returns the data type of array elements.
* size: Returns the total number of elements in the array.
* ndim: Returns the number of array dimensions.
* Indexing and slicing are used to access specific elements or ranges.

## Vector and Matrix Operations in NumPy

NumPy supports fast mathematical operations on vectors and matrices.

### Vector Operations

* **Vector Addition:** Adds corresponding elements of arrays.
* **Vector Subtraction:** Replaces the addition sign with a minus sign.
* **Scalar Multiplication:** Multiplies each element of an array by a scalar value.
* **Hadamard Product:** Element-wise multiplication of two arrays with the same shape.
* **Dot Product:** Computes the sum of element-wise products of two arrays, often used in vector and matrix operations.

### Visualization with Matplotlib

NumPy often works alongside **Matplotlib** to visualize numerical data from arrays through graphs and charts.

## Two-Dimensional Arrays in NumPy

* A **2D NumPy array** represents data in rows and columns, similar to a matrix or table.
* The shape attribute reveals the number of rows and columns in the array.
* The size attribute gives the total number of elements.
* Elements can be accessed using rectangular indexing.
* Scalars can be used to multiply all elements within an array.

## Summary

* Python enables efficient file handling through the use of open() and related methods.
* Pandas enables powerful data manipulation through DataFrames.
* NumPy provides efficient numerical computation with arrays and matrix operations.
* Combined with Matplotlib, these tools allow comprehensive data processing, analysis, and visualization in Python.

Module 5

# Application Program Interfaces (APIs)

## Introduction

In this section, we will explore **Application Program Interfaces (APIs)**. Specifically, we will discuss what an API is, API libraries, and REST APIs — including requests, responses, and an example using **PyCoinGecko**.

## What is an API?

An **API** allows two pieces of software to communicate with each other through inputs and outputs.

For example:

* You have a program and some data.
* APIs enable communication between your program and other software components.
* Like a function, you don’t need to understand how the API works internally — you only need to know its inputs and outputs.

## APIs in Python

### Pandas API

**Pandas** is an example of a software package that provides an API. Although much of Pandas isn’t written in Python, it allows Python code to communicate with underlying components.

* When you create a **dictionary** and pass it to the **DataFrame constructor**, this is considered creating an **instance** in API terminology.
* The data is passed to the Pandas API, and you interact with it through the **DataFrame** object.
* For example:
  + head() → Displays the first few rows of data.
  + mean() → Calculates and returns mean values.

## REST APIs

**REST APIs** (Representational State Transfer) are another type of API that allow communication over the **internet**, providing access to services like storage, data, and AI algorithms.

### REST API Terminology

* **Client:** The program or code making the request.
* **Resource:** The web service being accessed.
* **Endpoint:** The URL through which the client accesses the resource.
* **Request:** The message sent by the client to the server.
* **Response:** The message sent back from the server to the client.

### HTTP and JSON Communication

REST APIs typically use **HTTP** for communication:

* Requests and responses are sent as **HTTP messages**.
* These messages often contain **JSON files** with instructions or data.
* The web service performs the requested operation and sends back a **JSON response**.

## Example: PyCoinGecko API

Cryptocurrency data is ideal for APIs since it updates frequently. We will use the **PyCoinGecko** Python client/wrapper for the **CoinGecko API**, which updates every minute.

### Steps to Use PyCoinGecko

1. **Install and import** the library.
2. **Create a client object** to connect to the API.
3. **Request data** using a function call.

Example: Retrieving Bitcoin data in USD for the past 30 days.

* The API returns a **JSON** response as a Python dictionary containing:
  + price
  + market\_cap
  + total\_volumes
* Each includes a UNIX timestamp and a value.

### Data Conversion and Processing

* Select only the **price** data using the key price.
* Convert the nested list to a **DataFrame** with columns 'timestamp' and 'price'.
* Use the **pandas function to\_datetime()** to make timestamps readable:
  + Convert the timestamp column (unit: milliseconds) into human-readable **date** values.

### Candlestick Chart Creation

To visualize daily price data:

1. **Group by date** to calculate the minimum, maximum, first, and last prices of each day.
2. Use **Plotly** to create a **candlestick chart**.
3. Open the generated HTML file and click **“Trust HTML”** to view the chart.

## Summary

* APIs allow communication between different software components.
* **REST APIs** enable communication over the web using HTTP and JSON.
* **PyCoinGecko** simplifies cryptocurrency data retrieval and analysis.
* Using Pandas and Plotly, we can convert and visualize API data effectively.

# HTTP Protocol

## Introduction

In this section, we will discuss the **HTTP protocol** — the foundation of communication on the web. Specifically, we will cover the following topics:

* Uniform Resource Locator (URL)
* HTTP Request and Response

## Overview of HTTP

The **HTTP protocol (Hypertext Transfer Protocol)** is a general protocol used for transferring information through the web. It forms the basis of communication for **REST APIs**, which function by sending **requests** and receiving **responses** via HTTP messages.

When you, the **client**, use a web page, your browser sends an **HTTP request** to the **server** where the page is hosted. The server then searches for the requested resource, typically index.html by default.

If the request is successful, the server responds with an **HTTP response** that contains the requested object along with metadata such as:

* The type of resource
* The length of the resource
* Other relevant information

### Example: Web Server Resources

A typical web server contains various resources such as:

* HTML files
* PNG images
* Text files

When a request is made, the server sends the requested resource (e.g., one of the files) back to the client.

## Uniform Resource Locator (URL)

A **Uniform Resource Locator (URL)** is the standard way to locate resources on the web. A URL can be divided into three main components:

1. **Scheme** – Defines the protocol (e.g., http://).
2. **Internet Address (Base URL)** – Indicates the domain or location, such as www.ibm.com or www.gitlab.com.
3. **Route** – Specifies the path to the resource on the server, for example: /images/IDSNlogo.png.

## HTTP Request and Response

HTTP communication involves **requests** from the client and **responses** from the server.

### Request Message

An HTTP **request message** includes the following components:

* **Start Line:** Specifies the HTTP method (e.g., GET) and the resource requested (e.g., index.html).
* **Request Header:** Contains additional information about the request. In a GET request, this header may often be empty.
* **Request Body:** Contains data sent to the server (used in methods like POST).

### Response Message

An HTTP **response message** includes:

* **Start Line:** Contains the HTTP version, status code, and a descriptive phrase.
  + Example: HTTP/1.0 200 OK indicates success.
* **Response Header:** Provides additional metadata about the response.
* **Response Body:** Contains the requested data (e.g., an HTML document).

## HTTP Status Codes

HTTP status codes indicate the result of the request. They are grouped by their prefix:

| **Status Code Range** | **Type of Response** | **Example Code** | **Description** |
| --- | --- | --- | --- |
| 100s | Informational | 100 | Request received, continuing process |
| 200s | Successful | 200 | The request has succeeded |
| 400s | Client Error | 401 | Unauthorized request |
| 500s | Server Error | 501 | Not implemented |

## HTTP Methods

An **HTTP method** defines the action the client wants the server to perform. Common HTTP methods include:

* **GET** – Retrieve data from the server.
* **POST** – Send data to the server.
* **PUT** – Update an existing resource.
* **DELETE** – Remove a resource from the server.

## Summary

* The **HTTP protocol** enables data exchange between clients and servers across the web.
* A **URL** identifies the resource being accessed.
* **Requests** and **responses** are the core of HTTP communication.
* **Status codes** help identify the result of the request.
* HTTP **methods** define the type of action performed during communication.

In the next section, we will use **Python** to apply the GET method for retrieving data from a server and the POST method for sending data to a server.

# Working with the HTTP Protocol using the Requests Library

## Introduction

In this section, we will explore the **HTTP protocol** in Python using the requests library — a popular and easy-to-use library for handling HTTP requests. We will cover the following topics:

* Overview of the Requests library
* GET Requests
* POST Requests
* Comparison between GET and POST

## Overview of the Requests Library

The **requests** module is one of several Python libraries (along with httplib and urllib) that work with the HTTP protocol. It allows developers to send **HTTP/1.1 requests** effortlessly.

To start using it, import the library as follows:

import requests

You can make a **GET request** to a website such as www.ibm.com using:

r = requests.get('https://www.ibm.com')

The response is stored in an object named r, which contains details about the request and the response.

## Exploring the Response Object

The response object includes various attributes and methods for retrieving information:

### Checking Status Code

r.status\_code

* Returns the HTTP status code (e.g., 200 for success).

### Viewing Request Headers

You can access the headers of the request using:

r.request.headers

### Viewing Request Body

r.request.body

* For a **GET** request, the body will be None since no data is sent in the body.

### Viewing Response Headers

r.headers

* Returns a Python dictionary containing HTTP response headers.

Examples of commonly used keys:

* **Date** – The date and time when the response was sent.
* **Content-Type** – The type of data (e.g., text/html, application/json).

### Checking Encoding and Response Text

* To check encoding:

r.encoding

* To display the HTML content:

r.text[:100]

This displays the first 100 characters of the response body.

## GET Requests with Parameters

The **GET** method can also retrieve data from APIs using **query parameters**.

For this example, we will use the test website httpbin.org, a simple HTTP request and response service.

A GET request URL may look like this:

https://httpbin.org/get?name=Joseph&id=123

### Structure of a Query String

* Begins with a **?** followed by key-value pairs.
* Each pair is separated by an **=**.
* Multiple pairs are separated by an **&**.

Example:

| **Parameter** | **Value** |
| --- | --- |
| name | Joseph |
| id | 123 |

### Sending GET Request in Python

import requests

payload = {'name': 'Joseph', 'id': '123'}

r = requests.get('https://httpbin.org/get', params=payload)

You can print out the full URL and view details:

print(r.url) # Shows the URL with parameters

print(r.request.body) # None, since data is in the URL

print(r.status\_code) # 200 (OK)

To view the response as JSON:

r.json()

This converts the JSON response into a Python dictionary. The key args contains the parameters and values sent in the query string.

## POST Requests

A **POST** request sends data to a server in the **request body**, not in the URL.

To send a POST request, modify the route to /post:

payload = {'name': 'Joseph', 'id': '123'}

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

### Differences Between GET and POST

| **Feature** | **GET** | **POST** |
| --- | --- | --- |
| Data Sent | In URL as query string | In request body |
| Visibility | Visible in URL | Hidden from URL |
| Common Use | Retrieve data | Send or submit data |

### Comparing URLs and Bodies

* The **POST** request URL does **not** contain name-value pairs.
* Only the POST request contains a body.
* You can access the data using the key form in the response JSON.

## Summary

* The **requests** library simplifies working with HTTP in Python.
* **GET requests** retrieve data and send parameters via the URL.
* **POST requests** send data through the request body.
* You can access headers, status codes, and content using the response object.
* httpbin.org is a helpful site for testing and learning about HTTP requests.

# Web Scraping and HTML Basics

## Objectives

After completing this reading, you will be able to:

* Explain key concepts related to HTML structure and tag composition.
* Explore the concept of HTML document trees.
* Familiarize yourself with HTML tables.
* Gain insight into web scraping using Python and BeautifulSoup.

## Introduction to Web Scraping

Web scraping, also known as web harvesting or web data extraction, is the process of extracting information from websites or web pages using automation. It is widely used for:

* Data analysis and mining
* Price comparison
* Content aggregation
* Market research

## How Web Scraping Works

### HTTP Request

The process starts with an **HTTP request**, where a web scraper sends a request to a URL—similar to how a browser retrieves a webpage. Most scrapers use the **HTTP GET** method to access web content.

### Web Page Retrieval

The server responds with the **HTML content** of the requested webpage, including visible text, images, and the underlying structure that defines its layout.

### HTML Parsing

Once received, the HTML content is **parsed** into components (tags, attributes, and text). In Python, the **BeautifulSoup** library is used to parse and navigate this structure.

### Data Extraction

After parsing, scrapers identify and extract the required data—such as text, links, images, or tables—by searching for relevant HTML tags and attributes.

### Data Transformation

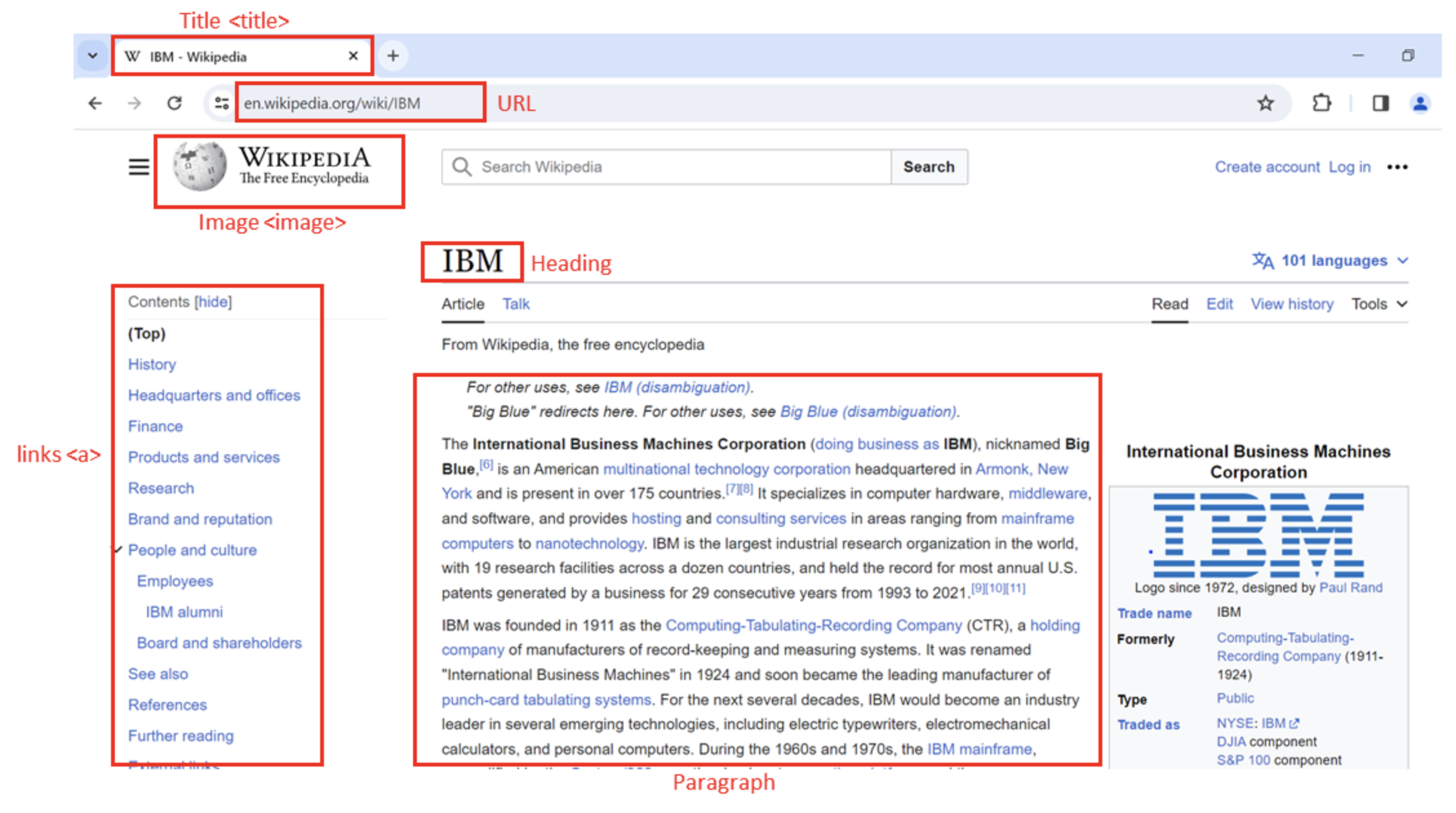
Extracted data is often cleaned or transformed by removing unnecessary HTML tags or formatting inconsistencies before use.

### Storage

The cleaned data is then stored in formats like **CSV**, **JSON**, or databases depending on the project requirements.

### Automation

Automation tools and scripts can perform repeated scraping across multiple pages or dynamic websites. This is especially helpful when sites frequently update content.



## Document Tree

### HTML Structure

HTML (Hypertext Markup Language) forms the backbone of web pages. Its structure consists of nested tags:

* <html>: Root element of the HTML page.
* <head>: Contains metadata about the page.
* <body>: Displays the main content.
* <h3>: Heading level 3 tag, used for titles or subheadings.
* <p>: Paragraph tag, used to hold text content.

### Composition of an HTML Tag

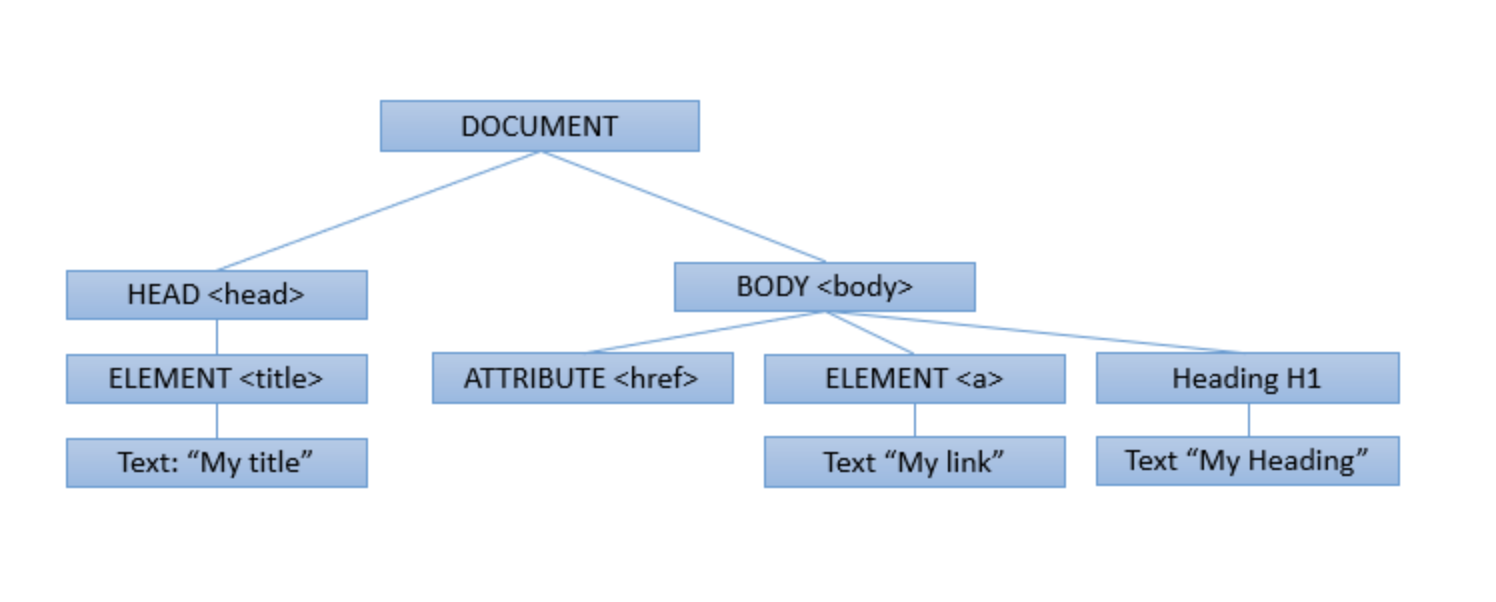
HTML tags define how content is structured and can contain attributes for additional information:

* Tags include an **opening** and **closing** tag (e.g., <p> and </p>).
* Tags may have **attributes** like href or src that provide extra details.

### HTML Document Tree

An HTML document can be visualized as a **tree structure** where each tag is a node:

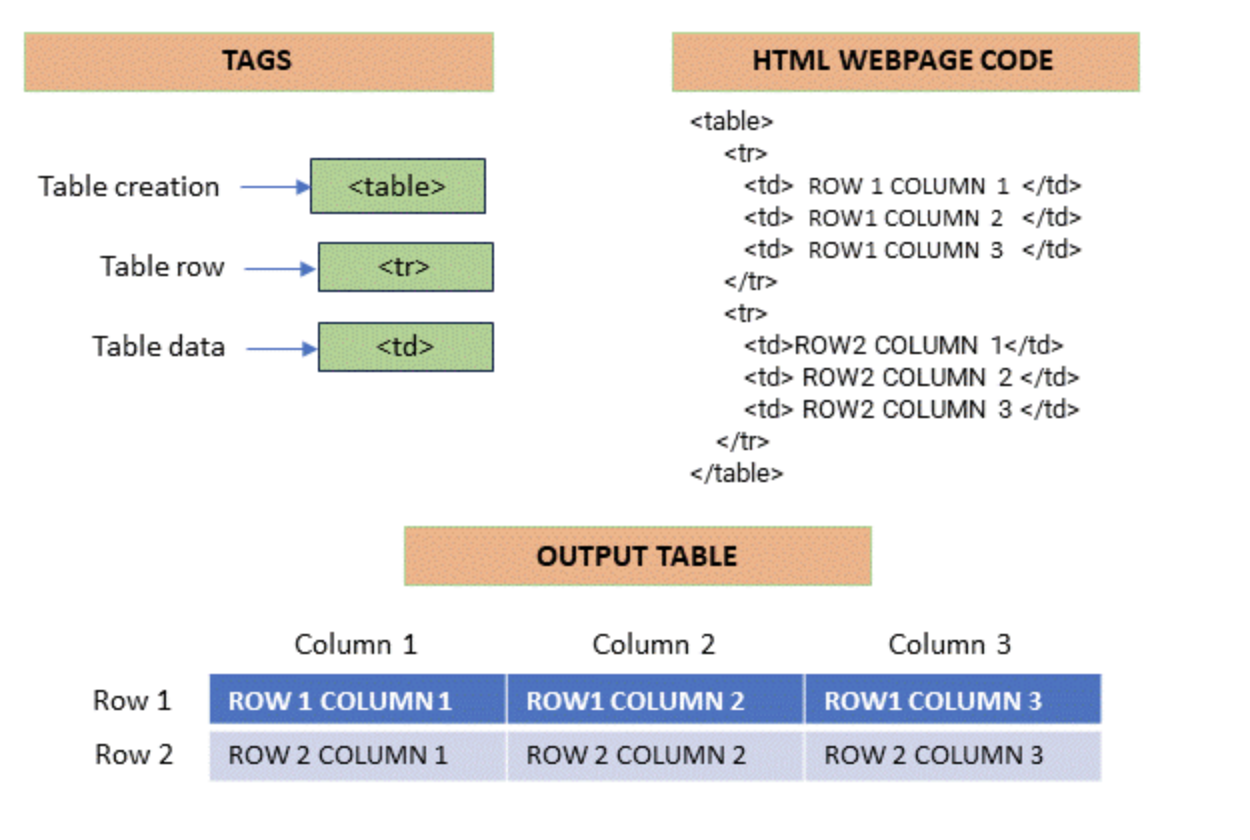
* Tags may contain other tags (children) or text.
* Tags within the same parent are siblings.
* Example: <html> contains <head> and <body> as children, and both are siblings.



## HTML Tables

HTML tables are used for displaying structured data:

* <table>: Defines the table.
* <tr>: Defines a table row.
* <th>: Table header cell (used for column headings).
* <td>: Table data cell (holds data entries).



## Web Scraping with Python

Web scraping in Python automates data extraction from web pages. It requires two key libraries:

* **Requests** – for sending HTTP requests.
* **BeautifulSoup** – for parsing HTML.

Example:

from bs4 import BeautifulSoup

import requests

# Specify the URL

url = 'https://en.wikipedia.org/wiki/IBM'

# Send a GET request

response = requests.get(url)

# Parse the HTML content

soup = BeautifulSoup(response.text, 'html.parser')

# Display the first 500 characters

print(response.text[:500])

## Navigating the HTML Structure

BeautifulSoup treats HTML as a **tree-like structure**, making navigation easy. For example:

# Find all anchor (<a>) tags

links = soup.find\_all('a')

# Print the text of each link

for link in links:

print(link.text)

## Custom Data Extraction

Custom extraction involves locating elements based on tags, attributes, or text content using BeautifulSoup’s methods.

## Using BeautifulSoup for HTML Parsing

BeautifulSoup helps you navigate, search, and extract information from HTML efficiently, providing flexibility to target specific sections of a web page.

## Using Pandas read\_html for Table Extraction

The **pandas** library provides read\_html() to automatically extract HTML tables into DataFrames for analysis:

import pandas as pd

url = 'https://en.wikipedia.org/wiki/IBM'

tables = pd.read\_html(url)

This method is ideal for importing webpage tables into a format suitable for data analysis.

## Conclusion

In this reading, you learned about the basics of **web scraping**, **HTML structure**, and the use of **BeautifulSoup** and **pandas** for data extraction. You now understand how to parse, navigate, and extract data responsibly from web pages.

# Introduction to HTML for Web Scraping

## Overview

This section reviews **Hypertext Markup Language (HTML)** and its importance in **web scraping**. Understanding HTML allows developers to extract useful data such as real estate listings, player statistics, and much more using Python.

## Objectives

* Review the structure and composition of an HTML document.
* Understand the composition of an HTML tag.
* Explain the concept of HTML document trees.
* Describe HTML tables and their elements.

## Introduction to HTML for Web Scraping

HTML is the language used to structure and display web pages. When performing web scraping, HTML provides the framework from which data is extracted. Python libraries like **BeautifulSoup** allow users to parse and retrieve specific data from HTML code.

Web pages, including those on sites like **Wikipedia**, contain large amounts of valuable information. If you understand HTML, you can easily access and extract this data programmatically.

## Basic HTML Structure

An HTML document is composed of nested elements called **tags**, enclosed in angle brackets < >. Tags define how the content should appear in a browser.

### Common HTML Elements

* <!DOCTYPE html> — Declares that the document is written in HTML.
* <html> — The root element that contains all other elements.
* <head> — Contains metadata about the page, such as the title.
* <body> — Contains the visible content displayed in the browser.
* <h3> — Represents a level-3 heading, making text larger and bold.
* <p> — Defines a paragraph of text.

### Example

In an example web page showing **National Basketball League** player data:

* The <h3> tags contain player names.
* The <p> tags contain player salaries.

Each piece of information is enclosed within its respective tags, defining its purpose and display.

## Composition of an HTML Tag

HTML tags consist of several components:

### Tag Structure

* **Opening Tag:** Defines where an element begins (e.g., <a>)
* **Closing Tag:** Defines where an element ends (e.g., </a>)
* **Content:** The data or text displayed within the element.
* **Attributes:** Additional information about an element, written as key-value pairs.

### Example: Anchor Tag

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

* **Tag Name:** a (defines a hyperlink)
* **Attribute:** href (specifies the URL destination)
* **Displayed Content:** IBM

You can think of each tag name as a **class** in Python and each tag instance as an **object**. Real-world web pages often include additional technologies like **CSS** and **JavaScript** for styling and functionality.

## HTML Document Tree

An HTML document can be represented as a **tree structure**, known as the **Document Object Model (DOM)**.

### Parent and Child Relationships

* The <html> tag is the **root** element.
* The <head> and <body> tags are **children** of <html>.
* The <title> tag is a **child** of <head>.
* The <h3> and <p> tags are **children** of <body>.
* Tags that share the same parent are called **siblings**.

This hierarchical representation makes it easy to navigate and extract specific parts of a web page using tools like **BeautifulSoup**.

## HTML Tables

HTML tables display data in a structured, grid-like format. They are composed of several key elements:

### Table Elements

* <table> — Defines the table.
* <tr> — Represents a table row.
* <th> — Defines a header cell (used for column headings).
* <td> — Defines a data cell.

### Example Structure

<table>

<tr>

<th>Player</th>

<th>Salary</th>

</tr>

<tr>

<td>John Doe</td>

<td>$1,000,000</td>

</tr>

</table>

Each <tr> contains multiple <td> elements, representing the individual cells of that row.

## Inspecting and Extracting HTML

Modern browsers allow users to **inspect** web pages to view the underlying HTML structure. This is crucial for identifying which tags contain the data you want to extract.

To inspect a web page:

1. Right-click on a web page.
2. Select **Inspect** or **Inspect Element**.
3. Review the HTML elements in the developer console.

## Summary

* HTML forms the backbone of all web pages.
* Tags define structure, while attributes provide additional information.
* HTML documents are organized as trees, allowing structured navigation.
* Tables store structured data that can be easily extracted.
* Understanding HTML is the foundation for effective web scraping using Python.

# Web Scraping

## Overview

Now, we’ll explore **Web Scraping** — a powerful technique for automatically extracting data from websites using Python. Instead of manually copying information, web scraping allows you to gather large amounts of data in just minutes.

## Objectives

* Define **Web Scraping**.
* Understand the role of **BeautifulSoup Objects**.
* Apply the **find\_all()** method.
* Perform basic **web scraping** operations on a website.

## Introduction to Web Scraping

Imagine you want to analyze hundreds of data points to find the best players on a sports team. Manually copying and pasting information into spreadsheets would take hours — or even days. **Web scraping** automates this process, extracting information from websites within minutes using a few lines of Python code.

To begin, we need two essential Python modules:

* **Requests** — to download web pages.
* **BeautifulSoup** — to parse and extract data from HTML.

## BeautifulSoup Object

Let’s say you want to find player names and salaries from a National Basketball League webpage.

### Creating a BeautifulSoup Object

1. Import the BeautifulSoup library.
2. Store the webpage HTML in a variable, e.g., HTML.
3. Pass it to the BeautifulSoup() constructor to create a soup object.

from bs4 import BeautifulSoup

soup = BeautifulSoup(HTML, 'html.parser')

The **BeautifulSoup object** represents the HTML document as a **nested tree structure**, making it easy to navigate and search.

### Tag Objects

Each **tag object** corresponds to an HTML tag. For example, <title> or <h3>. If multiple tags have the same name, BeautifulSoup selects the first occurrence by default.

In an example with **Lebron James**, the player’s name is enclosed within a bold (<b>) tag. Using the **tree representation**, you can navigate to child elements or move between **parent** and **sibling** elements.

#### Navigating the Tree

* **Child Elements:** Access using .contents or .children.
* **Parent Element:** Access using .parent.
* **Siblings:** Access using .next\_sibling or .previous\_sibling.

### Accessing Attributes and Text

Each tag may contain attributes, which can be accessed like dictionary key-value pairs.

tag['href'] # Access the href attribute

The text within a tag can be retrieved using .string or .text, returning a **NavigableString** — similar to a normal Python string but with BeautifulSoup functionality.

## The find\_all() Method

The **find\_all()** method filters HTML elements based on tag names, attributes, or text.

### Example: Extracting Table Data

Suppose we have a list of pizza places in an HTML table.

1. Create a BeautifulSoup object named table.
2. Use find\_all('tr') to find all table rows.

table\_rows = table.find\_all('tr')

Each element in table\_rows represents a row in the table — including the header row.

1. Iterate through the rows to extract each table cell.

for row in table\_rows:

cells = row.find\_all('td')

for cell in cells:

print(cell.text)

Each iteration retrieves cell data from the table, enabling structured extraction.

## Scraping a Webpage with Requests and BeautifulSoup

To apply web scraping on a real webpage, we also use the **Requests** library.

### Steps:

1. **Import the Modules:**

import requests

from bs4 import BeautifulSoup

1. **Download the Webpage:**

page = requests.get('https://example.com').text

1. **Create a BeautifulSoup Object:**

soup = BeautifulSoup(page, 'html.parser')

1. **Scrape the Desired Content:**  
   Navigate and extract the desired information from the soup using BeautifulSoup methods.

## Summary

* **Web Scraping** automates data extraction from websites using Python.
* **Requests** retrieves the webpage’s HTML, while **BeautifulSoup** parses and organizes it.
* **BeautifulSoup Objects** represent HTML as a tree structure for easy navigation.
* The **find\_all()** method is used to filter and extract specific elements.
* Together, these tools allow you to collect and analyze data efficiently from online sources.

# Web Scraping: A Key Tool in Data Science

## Introduction

Web scraping, also known as **web harvesting** or **web data extraction**, is a technique used to extract large amounts of data from websites. The data on websites is often **unstructured**, and web scraping enables us to convert it into a **structured** and analyzable form.

## Importance of Web Scraping in Data Science

In data science, **web scraping** plays a vital role in data collection and analysis. It is used for multiple purposes:

### Data Collection

Web scraping is a primary method for gathering data from the internet. This collected data can then be used for **analysis, research, and insights**.

### Real-time Applications

It is widely used for **real-time** use cases, such as **weather updates**, **price comparison**, and **news aggregation**.

### Machine Learning

Machine learning models require large datasets for training. Web scraping provides the raw data used to build these models.

## Web Scraping with Python

Python offers several libraries to perform efficient and scalable web scraping.

### BeautifulSoup

**BeautifulSoup** is a Python library used to extract data from **HTML** and **XML** documents. It parses page source code and organizes it in a tree structure, making data extraction simple and readable.

from bs4 import BeautifulSoup

import requests

URL = "http://www.example.com"

page = requests.get(URL)

soup = BeautifulSoup(page.content, "html.parser")

### Scrapy

**Scrapy** is an open-source web crawling and scraping framework for Python. It is designed for large-scale data extraction.

import scrapy

class QuotesSpider(scrapy.Spider):

name = "quotes"

start\_urls = ['http://quotes.toscrape.com/tag/humor/']

def parse(self, response):

for quote in response.css('div.quote'):

yield {'quote': quote.css('span.text::text').get()}

### Selenium

**Selenium** is a powerful tool for **automating web browsers**. It is often used for scraping dynamic content that requires JavaScript execution.

from selenium import webdriver

driver = webdriver.Firefox()

driver.get("http://www.example.com")

## Applications of Web Scraping

Web scraping is used across various industries and has many real-world applications:

### Price Comparison

Web scraping tools like **ParseHub** gather product data from e-commerce sites and compare prices across platforms.

### Email Address Gathering

Companies that use **email marketing** employ scraping techniques to collect email addresses for targeted campaigns.

### Social Media Scraping

It is used to extract data from social media platforms like **Twitter** or **Instagram** to analyze trends, user sentiment, and engagement.

## Conclusion

**Web scraping** is a critical skill in the data science toolkit. It transforms the web into a rich source of data for analysis and innovation. However, it must be practiced **ethically** and **responsibly**, respecting each website’s **terms of use** and **robots.txt** policies.

# Web Scraping Tables using Pandas

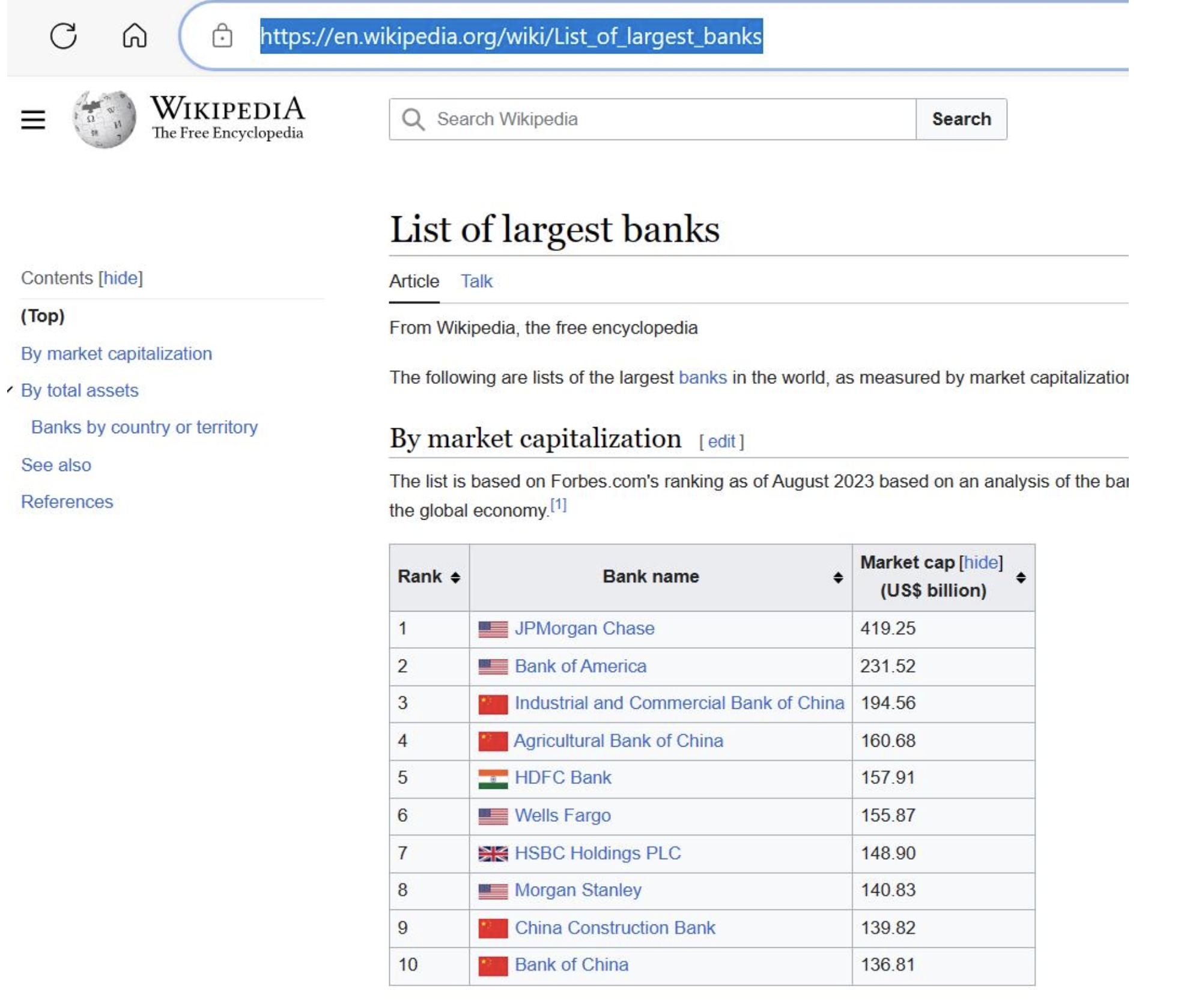
## Introduction

The **Pandas** library in Python contains a function called read\_html() that can be used to **extract tabular information** directly from web pages. This function allows users to quickly collect and analyze data presented in HTML tables.

## Example: Extracting Largest Banks by Market Capitalization

To understand how read\_html() works, let’s extract the **list of the largest banks in the world by market capitalization** from the following URL:

URL = 'https://en.wikipedia.org/wiki/List\_of\_largest\_banks'



We can use the pandas.read\_html() function to extract all the tables from this web page:

import pandas as pd

URL = 'https://en.wikipedia.org/wiki/List\_of\_largest\_banks'

tables = pd.read\_html(URL)

df = tables[0]

print(df)

This code extracts all tables from the web page and stores the **first table** (index 0) in a DataFrame called df. The extracted table displays the list of the largest banks by market capitalization.



**Note:** The web page is live and may be updated over time. The process of extraction remains the same regardless of future updates.

## Limitations of Using read\_html()

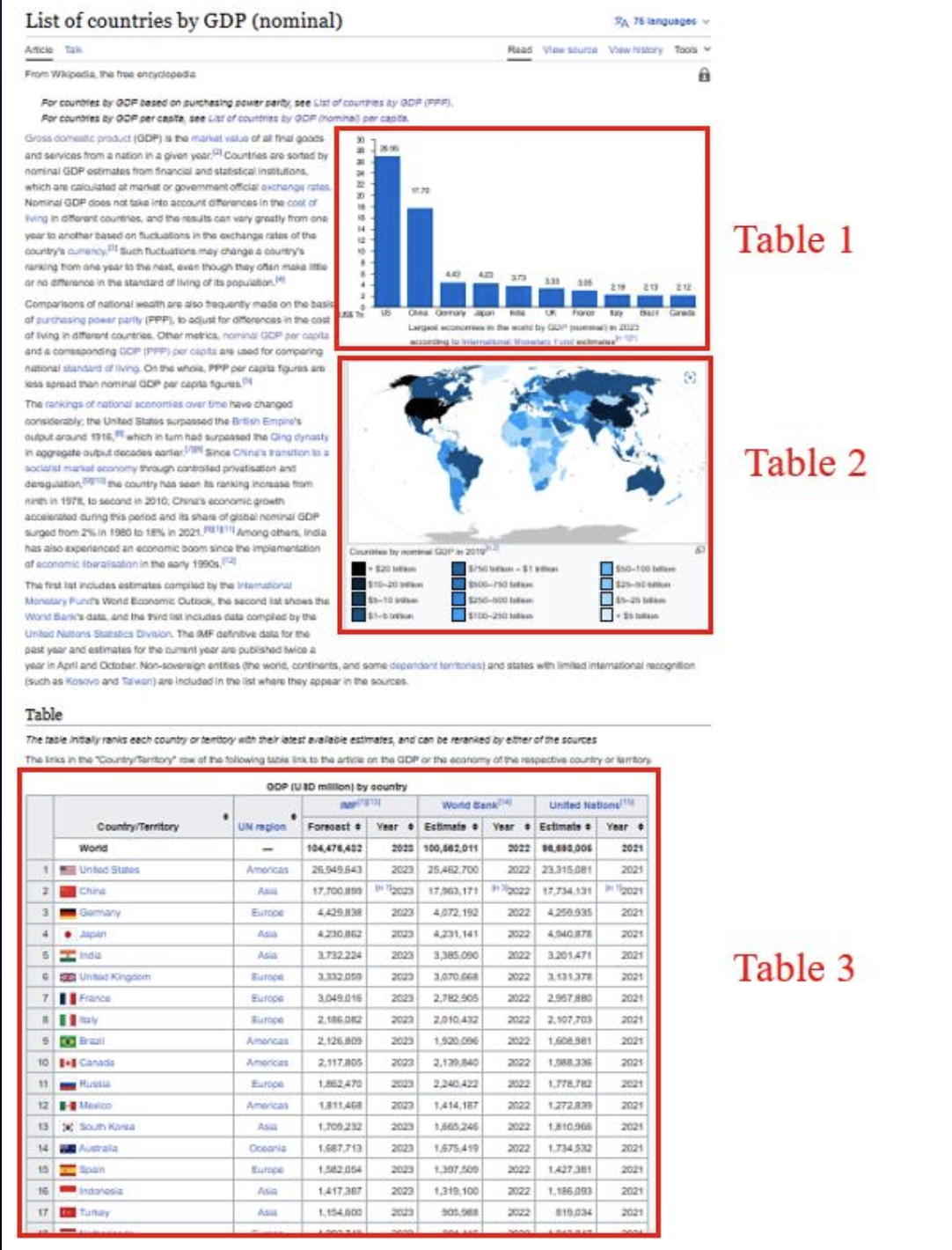
Although convenient, this method has certain **limitations**:

### 1. Non-Visible Tables

Some web pages store content in **table tags** that do not visually appear as tables. For example:

URL = 'https://en.wikipedia.org/wiki/List\_of\_countries\_by\_GDP\_(nominal)'

The page contains several tables, including ones that represent **images** stored in tabular form.



### 2. Extra HTML Elements

The extracted data may include unwanted elements such as **hyperlinks**, **footnotes**, or **symbols**. These are scraped directly by Pandas, which may require **additional data cleaning**.



## Example: Extracting a Table with Hyperlinks

Let’s extract a table from the page listing countries by GDP:

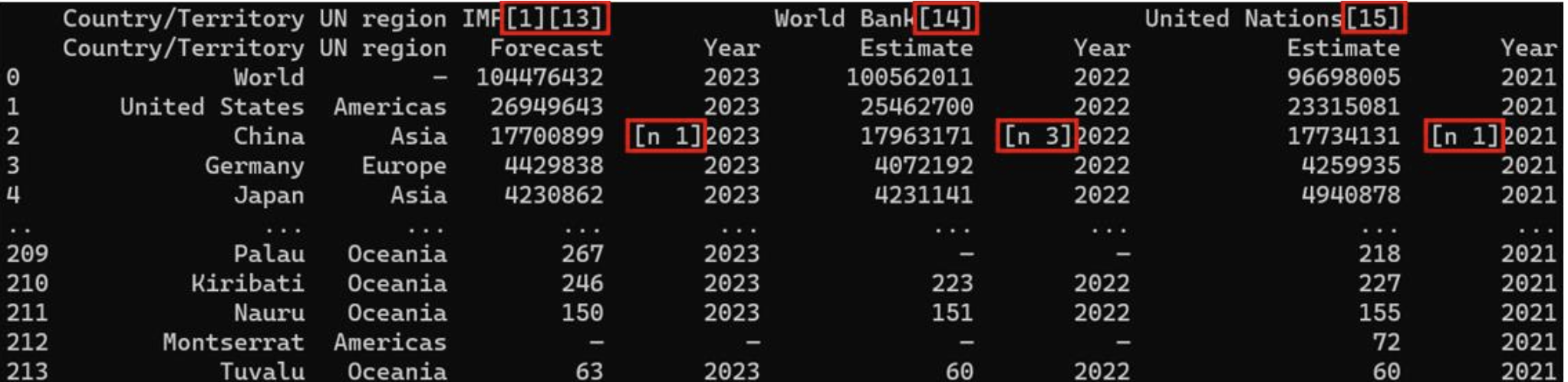
import pandas as pd

URL = 'https://en.wikipedia.org/wiki/List\_of\_countries\_by\_GDP\_(nominal)'

tables = pd.read\_html(URL)

df = tables[2] # The required table has index 2

print(df)



The extracted DataFrame contains hyperlink texts and other embedded elements, which need to be cleaned if only pure textual data is desired.

## Comparison with BeautifulSoup

While pandas.read\_html() is ideal for **tabular data extraction**, it only works with structured table formats. For **non-tabular or complex HTML data extraction**, the **BeautifulSoup** library remains the default and more flexible option.

## Summary

* pandas.read\_html() extracts tabular data directly from web pages into DataFrames.
* It is simple and effective, but may include extra HTML elements like hyperlinks.
* This method works best when the data is well-structured in HTML tables.
* For complex or unstructured data, use **BeautifulSoup** for customized extraction.

# Working With Different File Formats

## Learning Objectives

* Define different file formats such as **CSV**, **XML**, and **JSON**.
* Write simple programs to read and output data.
* List the Python libraries needed to extract data.

## Introduction

When collecting data, you will encounter many different file formats that need to be read or processed for analysis. Python simplifies this process with its predefined libraries. Before diving into Python, let’s review some common file formats.

## Understanding File Formats

Each file has an **extension** at the end of its name, which identifies its format and the type of software required to open it. For example:

* A file named FileExample.csv indicates it is a **CSV** (Comma-Separated Values) file.
* Other common file formats include **JSON** and **XML**.

When working with these file formats in Python, specific libraries help you read and extract data efficiently.

## Using Pandas to Read CSV Files

The first Python library to learn for handling files is **Pandas**. Once imported, Pandas allows you to easily read various file types.

### Reading a CSV File

Example:

import pandas as pd

file = 'FileExample.csv'

df = pd.read\_csv(file)

print(df)

In this example, the CSV file had no headers, so the first line of data was automatically set as the header.

### Adding Headers to a CSV File

To make the output more organized, you can manually define the headers:

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

print(df)

This ensures that the data is neatly organized with proper column names.

## Working with JSON Files

The next file format is **JSON** (JavaScript Object Notation). It stores data in a structured, language-independent format similar to Python dictionaries.

### Reading a JSON File

Example:

import json

with open('FileExample.json') as file:

data = json.load(file)

print(data)

Here, we import the json library, open the file, use the load() function to read it, and print the data.

## Working with XML Files

The **XML** (Extensible Markup Language) format is used to store and transport data. Although Pandas cannot read XML directly, Python provides tools to parse it.

### Reading an XML File

Example:

import xml.etree.ElementTree as ET

tree = ET.parse('FileExample.xml')

root = tree.getroot()

for child in root:

print(child.tag, child.attrib)

In this example, we import the xml library, use the etree module to parse the XML file, and loop through elements to extract information.

### Creating a DataFrame from XML Data

After parsing the XML, you can assign column headers, collect the required data, and append it to a Pandas DataFrame for analysis.

## Summary

In this video, you learned:

* How to recognize different file types (**CSV**, **JSON**, **XML**).
* How to use Python libraries such as **Pandas**, **JSON**, and **XML** to extract and process data.
* How to use **DataFrames** to organize and analyze data effectively.

# Summary

## Introduction

Simple APIs in Python are application programming interfaces that provide straightforward and easy-to-use methods for interacting with services, libraries, or data, often with minimal configuration or complexity. An API lets two pieces of software communicate with each other.

## Using APIs in Python

Using an API library in Python involves:

1. Importing the library.
2. Calling its functions or methods to make HTTP requests.
3. Parsing responses to access data or services provided by the API.

### Pandas API

The Pandas API processes data by communicating with other software components.

#### Creating a Pandas Instance

An instance is created when you form a dictionary and then use the **DataFrame** constructor to create a Pandas object.

#### Common Methods

* **head()** – Displays the mentioned number of rows from the top (default 5) of DataFrames.
* **mean()** – Calculates the mean and returns the values.

## REST APIs and HTTP Communication

REST APIs allow communication over the internet, providing access to resources such as storage, data, and AI algorithms.

### HTTP Protocol

The **HTTP (HyperText Transfer Protocol)** transfers data, including web pages and resources, between a client (web browser) and a server on the World Wide Web. It is the foundation for implementing REST APIs.

### HTTP Methods

* **GET** – Requests information from the server.
* **POST** – Submits data to the server.
* **PUT** – Updates existing data on the server.
* **DELETE** – Removes data from the server.

### HTTP Messages and Responses

HTTP messages transmit data over the internet, typically containing JSON files with operational instructions. These responses include details like resource type, resource length, and other metadata.

## URL Structure

A **Uniform Resource Locator (URL)** identifies and locates resources on the web. It is divided into three parts:

1. **Scheme** – Defines the protocol (e.g., HTTP or HTTPS).
2. **Internet Address/Base URL** – The domain name or IP address.
3. **Route** – The specific path to the resource.

You can modify query results with the **GET** method and obtain multiple parameters (like name, ID, etc.) from a URL using a **Query string**.

## Working with the Requests Library

**Requests** is a Python library that simplifies sending HTTP/1.1 requests. It allows developers to easily interact with web services using GET, POST, PUT, and DELETE methods.

## Time Series Data and Plotting

When dealing with time series data, the **Pandas time series functions** help process and analyze it. You can visualize daily candlestick data using **Plotly** to plot interactive charts.

## Web Scraping in Python

Web scraping involves extracting and parsing data from websites to gather information for analysis. Libraries like **Beautiful Soup** and **Requests** are commonly used.

### Understanding HTML

HTML comprises text enclosed within elements called **tags**, represented inside angular brackets (e.g., <p>, <div>). Web pages often also contain **CSS** and **JavaScript**.

Each HTML document can be viewed as a **tree structure** with nested elements, including strings and tags. HTML tables use elements like <table>, <tr>, <th>, and <td> to organize data.

### Extracting Tabular Data

You can extract tabular data from web pages using Pandas’ read\_html() method.

### Beautiful Soup Library

**Beautiful Soup** is a Python library for parsing and navigating HTML or XML documents.

#### Parsing a Document

Pass an HTML document to the **BeautifulSoup** constructor to create a structured object for navigation and data extraction.

#### Navigable String and find\_all Method

* A **NavigableString** behaves like a Python string but supports Beautiful Soup operations.
* The **find\_all()** method extracts content based on tag names, attributes, or text. It searches through tag descendants and retrieves all matching elements, returning them as a Python iterable (e.g., list).

## Working with File Formats

**File formats** define the structure and encoding of stored data. Common examples include:

* .txt for plain text
* .csv for comma-separated values
* .xml for Extensible Markup Language
* .json for JavaScript Object Notation
* .xlsx for Excel spreadsheets

Python supports multiple file formats and provides libraries to handle them efficiently.

### Accessing Different File Types

* **CSV**: Use Pandas to read and process CSV files.
* **JSON**: Use Python’s json module to parse and handle JSON data.
* **XML**: Use xml.etree.ElementTree for XML file parsing.

The file extension indicates the type of file and the appropriate tool required to open or process it.

## Summary

* APIs enable software systems to communicate effectively.
* REST APIs use HTTP methods like GET, POST, PUT, and DELETE for operations.
* The Requests library allows interaction with APIs easily.
* Web scraping extracts data using Beautiful Soup and Pandas.
* Python supports multiple file formats, including CSV, JSON, and XML, through specialized libraries.

# Next Steps

## Next Steps

As our next step, we can take one of the following Python Project courses to apply our new skills in real-world scenarios:

### Python Project Courses

* **Python Project for Data Science**
* **Python Project for Data Engineering**
* **Developing AI Applications with Python and Flask**

**Note:** Successful completion of this course is a pre-requisite for enrolling in these Python Project courses.

## Additional Learning Opportunities

We can further develop your skills in Python and data handling by exploring the following course:

* **Databases and SQL for Data Science with Python**

## Professional Certificates

If you are interested in starting or advancing a career in **Data Science**, **Data Engineering**, or **AI & Application Development**, consider pursuing one of the following IBM Professional Certificates and Specializations:

### IBM Professional Certificates

* IBM Applied AI Professional Certificate
* IBM Data Analyst Professional Certificate
* IBM Data Science Professional Certificate
* IBM Data Engineering Professional Certificate
* IBM Full Stack Software Developer Professional Certificate
* IBM DevOps and Software Engineering Professional Certificate
* IBM Back-End Development Professional Certificate

### IBM Specializations

* Applied Software Engineering Fundamentals Specialization
* Data Science Fundamentals with Python and SQL Specialization
* Applied Data Science Specialization