MsC Python Programming Lecture Notes

Prepared by: Gereltsetseg Altangerel
Assistant Professor

Class Coordinator & Proofreader: Máté Tejfel
Associate Professor

Department of Programming Languages and Compilers
Eötvös Loránd University (ELTE)

2026

Contents

About the Course

1. Overview

This MSc Python course is designed to help students build a strong foundation in Python programming while gradually applying Python in practical domains such as web development, databases, data science, and machine learning.

The course content progressively guides students from Python foundations toward practical real-world applications. The course emphasizes not only programming knowledge, but also computational thinking, problem solving, and practical coding skills.

Course Structure

The curriculum is structured into three interconnected pillars, each targeting a critical dimension of programming success:

1. **Lectures:** Focus on understanding fundamental concepts, programming principles, and computational thinking. → **KNOWLEDGE**
2. **Practice Labs:** Develope practical programming skills through hands-on coding exercises and guided examples. → **SKILL**
3. **Assignments:** Strengthen problem-solving abilities by applying concepts and programming skills to larger tasks. → **PROBLEM SOLVING**

Special attention was given to reducing unnecessary conceptual burden by introducing topics gradually and connecting each lecture naturally to the next.

The software proficiency is realized only at the intersection of all three:

Knowledge + Coding Skill + Problem Solving

2. Lecture Summary

The following table summarizes the lecture materials that have been completed for the MSc Python Programming course.

The lectures were designed with special attention to conceptual progression, pedagogical clarity, and practical applicability. The overall structure aims to guide students gradually from Python foundations toward more advanced software development concepts and real-world applications.

The course tries to emphasize the following characteristics:

- **Academic Structure:** The topics are organized into a coherent progression from basic computational concepts toward advanced programming practices.

- **Pedagogical Flow:** Each lecture is connected conceptually to the next in order to reduce unnecessary cognitive burden and support smoother learning transitions.
- **Content Coverage:** The course covers core Python programming concepts including computation, control flow, abstraction, recursion, object-oriented programming, error handling, file processing, and database interaction, api developement and some practical application.
- **Content Coverage:** Core topics include imperative control flow, procedural abstraction, recursion, and robust object-oriented programming (OOP). Students will also master real-world engineering essentials, including defensive error handling, file I/O operations, relational database integration, RESTful API development, and practical introductions to modern domain-specific libraries (e.g., data science and machine learning).
- **MSc-Level Appropriateness:** The materials emphasize computational thinking, abstraction, and problem solving in addition to Python syntax and implementation details.
- **Practical Balance:** The course combines theoretical understanding with hands-on programming exercises and practical applications.

#	**Lecture**
1	Lec 1-1 Introduction to Programming with Python
2	Lec 1-2 About the Course and Administration
3	Lec 2-1 The Programming Pipeline: From Data to Computation
4	Lec 2-2 From Data to Computation
5	Lec 3-1 From Computation to Control Flow
6	Lec 3-2 From Control Flow to String Processing
7	Lec 4 From Processing to Abstraction: Functions
8	Lec 5-1 Recursive Thinking in Programming
9	Lec 5-2 Dictionary
10	Lec 6 Object-Oriented Programming (OOP)
11	Lec 7 Advanced OOP
12	Lec 8-1 Modules, Packages, and Project Structure
13	Lec 8-2 Managing Python Environments
14	Lec 9 File Handling
15	Lec 10 Exception Handling
16	Lec 11-1 Pythonic Idioms
17	Lec 11-2 Managing Databases with Python
18	Lec 12 Using python in API Developement
19	Lec 13 Using Python in Practical Domains
20	Lec 14 Python Recap & Summary

Table 1: Summary of lecture materials prepared for the MSc Python course.

The goal of the course is not only to teach Python syntax, but also to develop structured

computational thinking and practical programming skills.

The concepts introduced in these lectures will be reinforced during the accompanying laboratory session through warm-up questions and guided programming exercises.

2. References and Acknowledgment

The core structure and several conceptual foundations of this course were inspired by open-source Python lectures provided by the Massachusetts Institute of Technology (MIT), particularly materials related to: *"Programming Fundamentals with Python"*, *"Functions"*, *"Dictionaries"*, *"Recursive Functions"*, *"Object-Oriented Programming (OOP)"*, *"Advanced OOP Concepts"*, and *"Testing, Debugging, and Exceptions"*

In particular, several concepts and organizational ideas were influenced by the following course:

> MIT OpenCourseWare.
> *6.0001 Introduction to Computer Science and Programming in Python.*
> Massachusetts Institute of Technology, Fall 2016.
> Available at: `https://ocw.mit.edu`

However, the lecture slides and notes have been substantially redesigned, reorganized, simplified, and extended based on my teaching experience and the learning needs of master level students. The course structure, topic progression, conceptual explanations, naming of lectures, and practical integration were customized to provide a smoother and more accessible learning experience.

The course materials were also improved according to discussions, feedback, and guidance from Professor Máté Tejfel.

For consistency and readability, individual references were not included on every slide or lecture note. Instead, this section serves as the primary acknowledgment of the educational sources that influenced the development of the course materials.

The course materials were adapted not only from academic sources, but also through practical teaching experience and continuous refinement.

4. Teaching and Learning Philosophy: Message to Students

Learning programming is an active process that requires continuous participation, experimentation, and practice. Students are encouraged to engage actively during lectures and labs, experiment with code, ask questions, and continuously improve their programming abilities.

Programming is not a subject that can be mastered passively by only reading slides or watching examples. Real understanding develops through writing programs, making mistakes, debugging errors, and trying different solutions repeatedly.

According to the Learning Pyramid, active practice is one of the most effective ways to develop long-term programming skills. Therefore, if you are new to programming:

Practice. Practice. Practice.

Use lectures, labs, assignments, and exams not only to collect grades, but also as opportunities to strengthen your technical skills, problem-solving abilities, and personal learning strategies.

The responsibility of the **teacher** is to guide, support, challenge, and create a structured learning environment. However, meaningful progress ultimately depends on the student's own effort, consistency, and curiosity.

The learning journey is yours.

Together, we work as a team toward your success throughout the semester.

This course is built on the idea of continuous improvement rather than perfection. Learning programming requires patience, persistence, and active engagement over time.

Success in programming is not about being perfect; it is about improving consistently.

Lecture 1: Introduction to Programming with Python

Foundations of Computation

The course content is structured progressively. We begin with foundational concepts, move toward abstraction and structured programming, and finally apply Python in real-world domains such as data science, web development, and machine learning. This layered approach ensures that students not only learn syntax but also develop computational thinking skills.

In this lecture, we begin with the fundamental ideas behind programming and examine how computers execute instructions. To build this understanding, we first consider a basic question:

What does a computer actually do?

What Does a Computer Do?

At its core, a computer is a machine that performs two primary functions:

- Execution of instructions
- Storage and retrieval of data

Modern computers are capable of executing billions of instructions per second while managing vast amounts of data in memory and storage systems. However, despite this impressive capability, a computer has no inherent intelligence. It cannot think, reason, or make decisions on its own.

A computer only performs the instructions that are explicitly given to it.

This observation leads to an important question:
What kinds of instructions can a computer perform?

1. Built-in Instructions

   Built-in instructions are the fundamental operations that a computer can perform directly. These include:

   - Arithmetic operations (addition, subtraction, multiplication, division)
   - Logical operations (comparisons, boolean evaluation)
   - Data movement (storing and retrieving values)

   These operations are limited in scope but form the foundation upon which all programs are built.

2. Custom Instructions

   Custom instructions are defined by the programmer. They allow us to:

   - Solve specific problems

- Automate repetitive tasks
- Build complex systems

Programming is essentially the process of designing and implementing these custom instructions.

At this point, an important question arises:
How do we create custom instructions, and what tools do we need to do so?

To answer this, we must understand how to transform a real-world problem into a form that a computer can execute.

From Problem to Solution: Goals and Steps

To create custom instructions, we must translate a problem into a form that a computer can understand. This transformation involves two key components.

1. Defining the Goal (Declarative Thinking)

   The goal specifies what we want the computer to achieve.

   For example:

   Determine whether a given number is a palindrome.

   This represents a high-level description of the desired outcome.

2. Defining the Steps (Imperative Thinking)

   The steps describe how the computer should achieve the goal.

   - Reverse the number
   - Compare it with the original
   - Return the result

   This step-by-step procedure leads us to one of the most important concepts in programming.

   However, these steps alone are not sufficient. To make the computer execute them, we need a programming language that can express these steps in a precise and structured way.

So, what do we call this sequence of steps?

Algorithms

An algorithm is a finite sequence of well-defined instructions used to solve a problem.

$$\text{Algorithm} = \text{Step}_1 + \text{Step}_2 + \cdots + \text{Step}_n$$

Algorithms are independent of programming languages. They can be expressed in natural language, diagrams, or pseudocode. However, for a computer to execute an algorithm, it must be written in a programming language.

Algorithms represent the logic of a solution, while programs represent its implementation.

Programming is the process of designing and expressing instructions that a computer can execute to solve problems.

Programming Languages

Programming languages serve as a bridge between human thinking and machine execution. They allow us to express algorithms in a structured and precise manner.

Components of a Language

Every programming language consists of the following components, similar to those found in natural languages:

- **Alphabet**: The set of valid characters (letters, digits, symbols)
- **Lexis**: Keywords, identifiers, and operators
- **Syntax**: Rules for forming valid statements
- **Semantics**: The meaning and behavior of those statements

Syntax determines whether a program follows the grammatical rules of the language, while semantics defines the meaning and behavior of the program when it is executed. Similar to natural languages, a sentence can be grammatically correct but meaningless; likewise, a program may be syntactically correct but semantically incorrect, leading to unexpected behavior.

From High-Level Code to Machine Execution

Computers do not understand high-level programming languages such as Python directly. Instead, they operate using machine code, a low-level representation consisting of binary instructions.

To bridge this gap, special programs are used.

Compiler

- Translates entire source code at once
- Produces an executable file
- Enables faster execution
- Detects errors before execution

Examples include C, C++, and Java.

Interpreter

- Translates and executes code line by line
- Does not produce a separate executable file
- Allows easier debugging
- Results in slower execution

Examples include Python and JavaScript.

Python uses an interpreter that translates and executes code line by line. Unlike compiled languages, this approach allows programmers to run and test code incrementally, receive immediate feedback, and identify errors more easily. As a result, Python is particularly well-suited for learning, experimentation, and rapid development.

Having seen how Python executes code through an interpreter, we now examine how these instructions are ultimately executed at the hardware level.

How a Computer Executes Instructions

Once code is translated into machine instructions, execution follows a well-defined cycle within the computer system. Execution Cycle:

1. The program is loaded into memory
2. The CPU fetches instructions one by one
3. Each instruction is decoded
4. The CPU executes the instruction
5. Results are stored back in memory

This process is known as the *fetch-decode-execute cycle*. Execution is inherently sequential, although modern systems may optimize certain operations.

Program Execution Flow: High-Level Code to Machine Execution

While the previous section focused on how instructions are executed at the hardware level, it is also important to understand the overall flow of a program from problem definition to final result.

Problem → Algorithm → Python Code → Interpreter → Machine Code → Execution → Result

This flow illustrates how an abstract problem is gradually transformed into executable instructions and ultimately into a concrete result.

Having understood how programs are constructed and executed, we are now ready to explore how to express these instructions effectively in Python.

Throughout this course, we will examine Python's syntax, semantics, and fundamental mechanisms, building the skills necessary to develop programs ranging from simple scripts to sophisticated applications.

Summary

In this lecture, we discussed how computers execute instructions and store data, and how programming involves defining custom instructions to solve problems. We introduced algorithms as structured descriptions of solutions and examined how programming languages are used to implement these algorithms. Finally, we saw how translators convert high-level code into machine instructions that can be executed by the computer. In the next lecture, we will explore the fundamentals of Python programming, including basic syntax, data types, and the development of simple programs.

A strong foundation leads to clear thinking.
Clear thinking leads to good programs.

Lecture 2

Lecture 2-1: The Programming Pipeline: From Data to Computation

2.1.1 Introduction

In the previous lecture, we introduced the fundamental idea that computers execute instructions written in a high-level programming language and translated into machine-level operations. We discussed how instructions are defined, how they are structured, and how they are executed.

In this lecture, we move one step further and examine how programs are organized in practice. In particular, we focus on how data flows through a program and how it is transformed into meaningful results.

2.1.2 Decomposing a Program

Let us consider the following Python program. Which logical parts can we identify in it?

```
a = 10
b = 20
sum = a + b
average = (a + b) / 2
print(sum)
print(average)
```

This program can be divided into three logical parts:

Input	Processing	Output
`a = 10`	`sum = a + b`	`print(sum)`
`b = 20`	`average = (a + b) / 2`	`print(average)`

This example illustrates how programs transform input values into meaningful output.

2.1.3 The Programming Pipeline

The fundamental structure of any program is straightforward: an algorithm takes input data and produces a result. This represents the core flow of computation. Along the way, the algorithm may analyze, manipulate, and transform data into a more meaningful form.

A program can be conceptually divided into three main stages:

- **Input** – receiving data
- **Processing** – transforming data
- **Output** – producing results

This model is known as the **Input–Processing–Output (IPO) pipeline**, and it provides a structured way of understanding how programs operate.

Input Stage: The input stage is responsible for receiving data. In this example, the values are hard-coded, meaning they are directly written into the program. In real-world applications, however, input can come from various sources:

- Command-line input
- Files
- Databases
- Networks
- External systems or sensors

The choice of input source and data type directly influences how the program processes the data.

Processing Stage: The processing stage is the core of any program. It transforms input data into meaningful output. This may involve:

- Arithmetic computations
- Logical evaluations
- Data transformations
- Algorithmic manipulation

This stage represents the **algorithmic logic** of the program.

Output Stage: The output stage communicates the results of the program. In this example, results are printed to the console. In practice, output may be directed to:

- Files
- Databases
- Network systems
- Graphical user interfaces
- Other applications

This process can be compared to a factory assembly line. A factory receives raw materials, processes them through a sequence of operations, and produces a final product.

$$\text{Steel} \rightarrow \text{Cutting} \rightarrow \text{Welding} \rightarrow \text{Painting} \rightarrow \text{Final Car}$$

Raw materials are processed step by step, with each stage performing a specific operation. The final product is the result of these transformations.

Similarly, in programming:

Input (Data/Data Object) → Processing → Output

Each stage transforms data, and the output of one stage becomes the input of the next.

In the programming pipeline, input may consist of data or data objects.
What is the difference between them?

2.1.4 Data vs Data Objects

In the programming pipeline, input may consist of either raw data or structured entities known as data objects. This distinction is central to understanding Python's programming model.

Data

- Represents raw values stored in memory
- Has no associated behavior
- Examples: 42, "hello", 3.14

Data Objects

- Combine a value with its type and associated behavior
- Support operations and methods

For example, an integer not only stores a value but also supports operations such as addition and subtraction.

In Python, do we work with raw data or with data objects?
In contrast, how does C represent and manipulate data?

To answer these questions, let us analyze how the same operation is performed at three different levels: hardware, C, and Python.

```
a = 10
b = 11
print(a + b)
```

Although the operation `a + b` appears the same, it is handled differently at each level:

- **Hardware level**: The CPU executes a simple addition using a built-in instruction (ADD).
- **C language**: The expression `a + b` is compiled into the corresponding machine instruction.
- **Python**: The + operator invokes a method associated with objects, implemented in C, and then executed by the hardware.

Python → C Implementation → Hardware

Python introduces an additional layer of abstraction between the programmer and the hardware, simplifying development by expressing operations in terms of data objects and their behavior. However, this abstraction introduces overhead and can reduce performance.

In contrast, C operates closer to the hardware, working directly with raw data types and translating operations into machine-level instructions. As a result, C provides higher performance and efficiency, while Python prioritizes readability, flexibility, and ease of use.

2.1.5 Summary

In this lecture, we explored the Input–Processing–Output pipeline and examined how programs transform data into meaningful results. Programming is fundamentally about transforming data into meaningful outcomes. We also introduced the concept of data objects and discussed Python's abstraction and execution model. In the next part of this lecture, we will explore data types, type casting, and the use of expressions and operators.

You now know the pipeline and the raw material—data.
Programming is about transforming it into meaningful results.

Lecture 2-2: From Data to Computation – Expressions, Data, and Variables

2.2.1 Introduction

In the previous lecture, we examined how programs transform data through the Input–Processing –Output pipeline and introduced the concept of data objects. We emphasized that programming is fundamentally about transforming input data into meaningful results.

In this lecture, we focus on how this transformation is actually carried out in practice. In particular, we study the mechanisms that allow programs to manipulate data, namely expressions, operators, and variables. These elements form the core computational layer of any program and enable the transition from static data to dynamic computation.

2.2.2 Types of Python Instructions

In Python, instructions can be broadly categorized into three types: definitions, expressions, and commands.

1. **Definitions** introduce new elements into a program. For example, assigning a value to a variable or defining a function creates structures that can be used later. Definitions do not necessarily produce immediate output; instead, they establish the environment in which computation takes place.
2. **Expressions** are responsible for computation. They combine data objects and operators to produce new values. Whenever a program performs a calculation, it is evaluating an expression. Thus, expressions represent the fundamental mechanism through which data is transformed.
3. **Commands, or statements, perform actions.** These actions may include printing output, controlling program flow, or interacting with the external environment. While expressions compute values, commands determine how and when those computations are executed.

Taken together, these three types of instructions define the structure and behavior of a program.

2.2.3 Expressions and the Nature of Computation

An expression is a combination of data objects and operators that evaluates to a value. Every meaningful computation in a program is expressed through expressions.

For example:

```
3 + 5    # 8
7.5 * 2 # 15.0
```

In the expression `3 + 5`, the numbers are data objects and the symbol + defines how they are combined. The result is a new data object.

Computation is therefore the process of generating new data from existing data. Every expression produces a value, and every value has a type, which determines how it can be used.

Data Objects and Their Behavior

In Python, all values are represented as data objects. Each object consists of a value and a type.

The type determines:

- what operations are allowed
- how the object behaves

For example:

```
3 + 5       # numeric addition
"3" + "5"   # string concatenation
```

Although the same operator is used, the behavior differs because the types are different.

Data Types: Scalar and Non-Scalar Objects

Data objects can be divided into two main categories.

Scalar objects (single value):

```
5       # int
3.14    # float
True    # bool
None    # NoneType
```

Non-scalar objects (structured data):

```
[1, 2, 3]       # list
"hello"         # string
{"a": 1}        # dictionary
```

Scalar objects are atomic, while non-scalar objects provide structure.

2.2.4 Operators and Data Transformation

Operators define how data objects interact. Common arithmetic operators include:

```
+   -   *   /   //   %   **
```

The result type depends on the operands:

```
5 + 3       # 8
5 + 3.0     # 8.0
```

Python automatically ensures type consistency.

Operator Precedence

Python evaluates expressions according to a defined order:

- **
- *, /
- +, -

Example:

```
2 + 3 * 4     # 14
(2 + 3) * 4   # 20
```

Parentheses can be used to control the evaluation.

2.2.5 Variables and Binding

A variable is a name that refers to an object in memory.

```
x = 10
```

Here:

- 10 is an object
- x is a reference to that object

Variables do not store values directly; they refer to objects.

The Role of Variables

Variables improve readability and flexibility.

```
pi = 3.14159
radius = 2.2
area = pi * (radius**2)
```

Meaningful names make programs easier to understand and maintain.

Changing Bindings

Variables can be reassigned:

```
radius = 2.2
area = pi * (radius**2)

radius = radius + 1
```

The value of `area` does not update automatically. It must be recomputed.

Type Conversion: Type conversion changes the type of an object.

Explicit conversion:

```
int(3.9)   # 3
float(3)   # 3.0
```

Implicit conversion:

```
5 + 3.2    # 8.2
```

Python performs automatic conversions when needed.

2.2.6 Dynamic vs Static Typing

Python is a dynamically typed language, which means that the type of a variable is determined at runtime rather than being explicitly declared.

```
x = 5
x = "Hello"
```

In this example, the variable `x` is first bound to an integer object and later re-bound to a string object. This is valid in Python because variables are not restricted to a fixed type.

In contrast, statically typed languages such as C or Java require variables to have a fixed type that must be declared before use.

Example in C:

```
int x = 5;
x = "Hello"; // Error: incompatible types
```

Example in Java:

```
int x = 5;
x = "Hello"; // Compile-time error
```

In these languages, the type of a variable cannot change during execution, and type errors are detected before the program runs.

Key Comparison:

Dynamic Typing (Python):

- Type is determined at runtime
- Variables can refer to objects of different types
- More flexible and easier to use

Static Typing (C / Java):

- Type is declared and fixed
- Type checking happens at compile time
- More strict but often more efficient

Thus, in Python, a variable is not a container with a fixed type but a reference to an object. The type belongs to the object, not to the variable. This distinction enables Python's dynamic typing, allowing the same variable to be associated with different types of objects at different points in a program.

2.2.7 Summary

In this lecture, we explored how programs perform computation through expressions, operators, and variables. We examined data objects, expressions, and the role of variables as references to objects. We also discussed type conversion and Python's dynamic typing system. Programming is fundamentally the process of transforming data into meaningful results. In the next lecture, we will explore control flow structures such as conditionals and loops, which allow programs to make decisions and repeat actions.

You now have more ingredients.
What will you cook?

Lecture 3

Lecture 3-1: From Computation to Control Flow

3.1.1 Introduction

In the previous lectures, we explored how programs perform computations using data, expressions, and variables. We learned how to transform data into meaningful results through the programming pipeline.

However, computation alone is not sufficient to build realistic programs. Real-world problems require programs to make decisions and repeat actions. This leads to an important question:

How can we control the flow of execution in a program?

In this lecture, we introduce control flow, which allows programs to move beyond simple sequential execution and handle more complex behavior.

Straight-Line Programs

A straight-line program executes statements sequentially, one after another, without any decision-making or repetition.

```
hour = int(input("Starting time (hours): "))
mins = int(input("Starting time (minutes): "))
dura = int(input("Event duration (minutes): "))

total_mins = hour * 60 + mins + dura
end_hour = (total_mins // 60) % 24
end_min = total_mins % 60

print(f"End time: {end_hour}:{end_min}")
```

Such programs are simple and easy to understand, but they are limited in complexity because they cannot adapt to different conditions or repeat actions.

3.1.2 From Computation to Control

So far, we have learned how to perform computations using expressions and operators. However, an important question arises:

- When should a computation be executed?
- Which computation should be selected?

To answer these questions, we need mechanisms to control the execution of programs.

Control Flow: Control flow refers to the order in which instructions are executed in a program. Instead of always executing statements sequentially, control flow allows programs to:

- make decisions
- repeat actions

There are two main types of control flow used not only in Python, but also in many programming languages.

- **Branching (Conditionals)** – decision-making using `if`, `elif`, `else`
- **Iteration (Loops)** – repetition using `while` and `for`

1. Conditional Statements: Conditional statements allow a program to make decisions based on conditions. A condition is an expression that evaluates to either `True` or `False`. Based on this result, the program decides which block of code to execute.

```
if num1 > num2:
    print("num1 is greater")
elif num1 < num2:
    print("num2 is greater")
else:
    print("They are equal")
```

Execution is no longer fixed; it depends on conditions.

Different ways of writing conditional logic may produce the same result but differ in readability.

Simple `if-elif-else` structures are generally preferred over deeply nested conditions, as they improve clarity and maintainability.

Note: Importance of Indentation

Python uses indentation to define code structure. Indentation determines which statements belong to a block and controls the flow of execution. Incorrect indentation can lead to errors or unintended behavior.

2. Iteration: Repetition in Programs: Many programming problems require repeating actions. Writing the same code multiple times is inefficient, difficult to maintain, and prone to errors.

Loops allow programs to execute a block of code repeatedly and automatically. Like many programming languages, Python provides two main looping structures.

1. **While Loops:**

   A `while` loop repeats a block of code as long as a condition remains true.

```
    num_x = 5
while num_x > 0:
    print("X")
    num_x -= 1
```

The loop continues until the condition becomes false.

2. **For Loops:**

   A `for` loop iterates over a sequence of values.

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

This is commonly used when the number of iterations is known.

The range() Function: `range()` is a built-in Python function that generates a sequence of numbers.

General form: range(start, stop, step)

It can take up to three parameters:

- **start**: starting number
- **stop**: ending number (not included)
- **step**: increment between numbers

Example 1 — Using `break`

```
for i in range(10):
    if i == 5:
        break
    print(i, end = " ")

```

Output:

```
0 1 2 3 4
```

The loop stops completely when `i == 5`.

Example 2 — Using `continue`

```
for i in range(10):
    if i == 5:
        continue
    print(i, end = " ")

```

Output:

```
0 1 2 3 4 6 7 8 9
```

The value `5` is skipped, but the loop continues.

`break` stops the loop entirely, while `continue` skips only the current iteration.

For vs While Loops: Both loop types can perform similar tasks but are used in different situations.

- `for` loop: used when the number of iterations is known
- `while` loop: used when repetition depends on a condition

3.1.3 Summary

In this lecture, we introduced control flow as a fundamental concept in programming. We moved beyond simple computation and learned how programs can make decisions and repeat actions.

We explored conditional statements, loops, indentation, and different loop control mechanisms.

Programs are not just sequences of computations; they are dynamic systems that make decisions and repeat actions.

In the next lecture, we will explore more advanced program structuring techniques, including functions and abstraction.

Programs, like us, make decisions, repeat actions, and skip steps when needed.

Lecture 3-2: Working with Strings in Python

In the previous lecture, we introduced control flow structures, learning how programs make decisions and repeat actions to move beyond simple sequential execution. In this lecture, we apply these concepts to an essential built-in data type: strings. Strings represent textual data and serve as the foundation for real-world applications such as processing user input, parsing files, and analyzing unstructured text data.

3.2.1 Strings in Python

A string is an ordered sequence of characters, which can include letters, digits, symbols, and whitespace. Strings in Python are case-sensitive, meaning uppercase and lowercase characters are treated as distinct entities.

```
s = "abc"
# len() returns the total
   character count
print(len(s))
```

Output:

```
3
```

Strings can also be compared using standard relational operators (e.g., ==, !=, <, >), which evaluate strings based on their lexicographical (alphabetical) order via underlying character Unicode values.

Indexing and Slicing Individual components of a string can be accessed or extracted using structural offsets. Python supports both forward indexing (starting at `0`) and negative indexing (counting backward from the end).

Positive Indexing

```
s = "abc"
print(s[0])  # First element
print(s[2])  # Last valid index

```

Output:

```
a
c
```

Negative Indexing

```
s = "abc"
print(s[-1]) # Last element
print(s[-2]) # Second to last

```

Output:

```
c
b
```

Note: Attempting to access an index outside the range `[-len(s), len(s) - 1]` raises an `IndexError`.

To extract a subsequence (substring), Python provides the slicing syntax: `[start : stop : step]`. The `start` boundary is inclusive, the `stop` boundary is exclusive, and the optional `step` argument dictates the stride interval.

Slicing Implementations

```
s = "abcdefgh"

print(s[3:6])   # Extraction
print(s[3:6:2]) # Stride
    interval
print(s[::-1])  # Reversal
    sequence

```

Output:

```
def
df
hgfedcba
```

3.2.2 Core String Properties

Python strings possess three core structural traits that govern how they behave in algorithms:

(a) **Iterable:** They can be stepped through sequentially, character by character, using a loop structure.

(b) **Immutable:** Once instantiated in memory, their state cannot be mutated. Modifying a character in-place results in a runtime error. To alter text, a brand new string object must be constructed.

(c) **Membership Testing:** They support the evaluation of substrings using native `in` and `not in` collection operations.

Immutability & Assignment

```
s = "hello"
# s[0] = 'y'  # Raises
    TypeError

# Correct approach: Re-
    allocation
s = 'y' + s[1:]
print(s)

```

Membership Verification

```
word = "apple"

print("a" in word)
print("x" not in word)

```

Output:

```
True
True
```

Because strings are iterable sequences, we can programmatically traverse them. While a manual tracker can be maintained within a `while` loop, an idiomatic `for` loop is the preferred implementation for cleaner readability and design.

For Loop (Idiomatic)

```
word = "hello"
for char in word:
    print(char, end=" ")
```

While Loop (Manual Pointer)

```
word = "hello"
i = 0
while i < len(word):
    print(word[i], end=" ")
    i += 1
```

Output (Both Approaches): `h e l l o`

3.2.3 Summary

In this lecture, we explored strings as sequential collections of characters. We analyzed their fundamental architectural constraints—namely immutability, sequence iteration, and sequence membership validation. Additionally, we mastered positional indexing and range slicing to isolate data within a string, and demonstrated sequential collection traversal strategies.

In the next lecture, we will dive deeper into advanced string methods and introduce modular code organization via custom procedural abstractions and functions.

Now that you can control execution, you can process and transform real data—like text.

Lecture 4: From Processing to Abstraction — Functions

Introduction and Motivation

Why Do We Need Structure?

Firstly, let us begin with a simple question:

Which text is easier to read?

Consider two types of text. The first one is written as a continuous block. It contains multiple ideas, but they are not separated. As a result, it is difficult to follow and mentally exhausting.

The second one is divided into paragraphs. Each paragraph expresses a clear idea. The structure makes it easier to read and understand.

Obviously, the second one is easier to read. Why?

Because paragraphs organize ideas and improve readability. Usually, one paragraph expresses one main idea. If a text contains multiple ideas, it is divided into multiple paragraphs.

This observation leads to an important insight:

Programs, like text, also need structure.

Secondly, now consider the following program and try to identify problem:

```
pi = 3.14159

r1 = 2
area1 = pi * (r1 ** 2)

r2 = 5
area2 = pi * (r2 ** 2)

r3 = 10
area3 = pi * (r3 ** 2)
```

Let us ask:

Do you notice a problem in this code?

Although the program works correctly, the same computation is repeated multiple times. Only the input values change.

Let us ask a follow-up question:

What happens if we need to modify the formula?

If the computation changes, it must be updated in multiple places throughout the program. This leads to repetitive code, makes maintenance more difficult, and increases the risk of errors.

Now we can ask a more general question:

How do we manage such complexity?

As programs grow, they become harder to read, understand, and maintain. Therefore, we need a way to organize them. The key idea is **decomposition**, which means breaking a program into smaller, manageable pieces.

So far, we have focused on writing programs step by step, performing individual computations. We now move from processing to **abstraction**, asking:

How can we organize and reuse our code?

This marks a turning point in learning programming. Up to now, we have focused on writing and executing instructions. From this point forward, we focus on structuring and organizing those instructions—moving from computation to organization, and from processing to abstraction.

Python provides several ways to structure programs, including functions, classes, and modules. Functions are reusable blocks of code that perform a specific task, classes serve as blueprints for creating objects that combine data and behavior, and modules organize related code into separate files to improve structure and reuse. Among these, functions are the most fundamental tool.

Functions

A function is a reusable block of code that performs a specific task. It allows us to organize programs into smaller, manageable units and avoid repetition. Functions are defined once and executed when they are called.

A function typically includes a name that describes its purpose, optional parameters that represent inputs, a body containing the instructions, and an optional return value that produces a result.

Functions help structure programs in the same way paragraphs structure text. Each function represents a single logical unit, making programs easier to read, understand, and maintain.

Good programming is not about writing more code, but about writing better code. Well-designed programs emphasize clarity, maintainability, and reusability.

4.2.1 Defining Functions in Python

After understanding the role of functions in organizing programs, the next step is to learn how to define them.

A function is created using the `def` keyword, followed by the function name, a pair of parentheses, and a colon. The body of the function is written on the following lines and must be indented.

```
def function_name(parameters):
    # function body
    return result
```

A function definition includes a name that identifies the operation, optional parameters that represent inputs, a body containing the computation, and an optional `return` statement that sends the result back to the caller.

Example. Let us revisit the earlier example where the area of a circle was computed multiple times. Instead of repeating the same computation, we can define a function:

Function Definition

```
def area(radius):
    return 3.14159 * (radius ** 2)

```

Execution Reusability

```
area(2)
area(5)
area(10)

```

Here, `area` is the function name, `radius` is the input parameter, and the expression inside `return` defines the computation. Once defined, the function can be reused.

A key observation is that defining a function does not execute it; the function runs only when it is called. By encapsulating a computation into a function, we avoid repetition and improve readability and maintainability.

This directly connects to the idea of abstraction. When we use a function such as `area(5)`, we do not need to think about how the computation is performed—we only need to understand what the function does.

Focus on what a function does, not how it works.

4.2.2 Return vs Print: Understanding the Difference

When working with functions in Python, it is important to distinguish between `return` and `print`. Although both may appear to produce output, they serve fundamentally different purposes.

Let us begin with a guiding question:

Does displaying a value mean the program can use it later?

The answer is no. Displaying a value and returning a value are not the same.

The `print` function is used to display information to the user. It outputs a value to the screen, but does not send anything back to the program. In contrast, the `return` statement is used inside functions to send a value back to the caller, making it available for further computation.

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

```

When a `return` statement is executed, the function stops immediately and sends the result back. This allows the returned value to be stored, reused, or combined with other computations.

The difference can be summarized as follows: `return` is used for computation within the program, while `print` is used for displaying information to the user.

return is for the program, print is for the user.

Consider the following example:

```python
def square(x):
    print(x * x)

result = square(5)
print(result)

```

Output:

```
25
None
```

Here, the function prints the result but does not return it, so Python automatically returns `None`, indicating the absence of a value.

To build reusable and composable programs, functions should return values rather than only printing them.

Function Calls and Scope

So far, we have learned how to define functions. We now ask a more important question:

What actually happens when a function is called?

A function call transfers control from the main program to the function. The program pauses, executes the function body, and then returns to its original point of execution.

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

result = square(5)
```

During this process, the argument `5` is passed to the function and assigned to the parameter `x`. The expression `x * x` is evaluated, and the result is returned.

Key Idea: A function call creates a separate execution context.

This leads to the concept of **scope**, which determines where variables exist and where they can be accessed.

Variables defined inside a function belong to the **local scope** and are accessible only within that function. Variables defined outside functions belong to the **global scope** and can be accessed throughout the program.

```python
x = 10

def test():
    x = 5
    print(x)
test()
print(x)

```

Output:

```
5
10
```

In this example, the local variable inside the function overrides the global variable, while the global variable remains unchanged outside the function.

Understanding scope is essential because it prevents unintended interference between variables and allows functions to operate as independent units.

A function is not just a reusable block of code—it is an independent unit with its own local environment.

A common misconception is that variables inside a function affect variables outside. This is not true unless explicitly specified.

4.3.1 Parameters and Return Model

Functions follow a simple transformation model:

Input (argument) $\longrightarrow$ *Function* $\longrightarrow$ *Output (return)*

Values are passed into functions through **arguments**, which are received by **parameters**. The function processes these values and returns the result using the `return` statement.

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

The returned value can be stored and reused:

```
result = square(5)
print(result)   # 25

```

```
def f():
    print("Hello")

```

If a function does not explicitly return a value, Python returns `None`, representing the absence of a result.

Arguments provide input, parameters receive them, and `return` produces the output.

A function can therefore be understood as a transformation that maps input values to output values, forming the foundation of structured and reusable program design.

Objects in Python (Recap)

In Python, every value is represented as an **object**, including numbers, strings, lists, and even functions. Each object has a value, a type, and an identity, which represents its location in memory.

```
x = 10
print(id(x))
```

The function `id()` returns the identity of an object.

At this point, it is useful to distinguish between two kinds of objects. **Data objects** represent values such as numbers, strings, and lists, while **function objects** represent computations—executable code that can be invoked. In other words, data answers the question *"What is the value?"*, while functions answer *"What should be done?"*.

Since functions are also objects, they behave like values in Python. This means they can be assigned to variables, passed as arguments, and returned from other functions.

```
def greet():
    return "Hello"

f = greet
print(f())

```

In this example, the function `greet` is assigned to the variable `f`, and calling `f()` executes the function.

This idea leads to a powerful question:

If functions are values, can they be returned from other functions?

The answer is yes.

```
def outer():
    def inner():
        return "Hello from inner function"
    return inner

f = outer()
print(f())
```

Here, the function `outer` returns another function, which can then be executed through the variable `f`.

We can now generalize this idea:

Value → Function → Value

A function can take values as input and produce values as output, where the values themselves can be either data or other functions. This highlights a fundamental principle of Python: functions are not just code—they are values that represent computations and can be manipulated like any other object.

Summary

In this lecture, we moved from writing programs as sequences of instructions to understanding how to organize and structure them. We introduced functions as reusable units of computation that help

reduce repetition, improve readability, and support modular program design. We explored how functions are defined, how they receive input through parameters and arguments, and how they produce results using the `return` statement. We also examined the difference between returning values and printing output, emphasizing that well-designed functions should return values for reuse.

Furthermore, we studied how function calls create their own execution context and how variables are managed through local and global scope. This led to a deeper understanding of how programs behave during execution. Finally, we introduced the concept that everything in Python is an object, including functions. This allows functions to behave like values—they can be assigned, passed, and returned—enabling more flexible and powerful program design.

You are no longer just writing code—you are organizing ideas.

Lecture 5

Lecture 5-1: Recursive Thinking in Programming

5.1.1 Introduction

Recursion is the process of repeating a structure in a self-similar way. This idea appears frequently in nature. For example, the branching of trees, the structure of snowflakes, and fractal patterns such as ferns or Romanesco broccoli all exhibit similar shapes at different scales. Each part resembles a smaller version of the whole.

In programming, recursion follows the same principle. It is a technique in which a function calls itself in order to solve a problem.

This leads to a fundamental idea:

> *A complex problem can be solved by reducing it to smaller versions of the same problem.*

Recursive Thinking

Recursion is not only a programming technique but also a way of thinking about problem solving.

- From a programming perspective, recursion involves a function calling itself to solve a problem step by step, gradually reducing it to simpler cases.
- From an algorithmic perspective, recursion is closely related to strategies such as divide-and-conquer and decrease-and-conquer, where a complex problem is broken down into smaller, more manageable subproblems that follow the same structure as the original.

5.1.2 Recursion vs Iteration

Repetition in programming can be expressed in two fundamental ways: iteration and recursion.

- Iteration relies on looping constructs such as `for` and `while`, where the computation is carried out step by step using variables to track the current state.
- In contrast, recursion expresses repetition through function calls, where a problem is defined in terms of smaller instances of itself. Rather than focusing on the sequence of steps, recursion emphasizes the structure of the problem and how it can be reduced.

This leads to an important conceptual distinction: iteration reflects an **imperative** style of thinking, where we describe how to perform a computation step by step, while recursion reflects a more **declarative** style, where we describe what the problem is and let the recursive structure handle the process of solving it.

5.1.3 Defining Recursive function

A correct recursive function must include two essential parts: a **base case**, which defines the simplest instance of the problem and stops the recursion, and a **recursive step**, in which the function calls itself with a smaller input, moving toward the base case.

Example: Multiplication

The difference between iteration and recursion can be clearly illustrated using multiplication.

Iterative Version

```
def mult_iter(a, b):
    result = 0
    while b > 0:
        result += a
        b -= 1
    return result
```

Key Characteristics

- Uses a loop (`while`)
- Maintains state using variables
- Explicit step-by-step computation

Recursive Version

```
def mult(a, b):
    if b == 1:
        return a
    else:
        return a + mult(a, b-1)
```

Key Characteristics

- Base case: `b == 1`
- Recursive step: reduces `b`
- Problem defined in terms of itself

5.1.4 Understanding Recursion

Example: Factorial

The factorial function provides a classic example of recursion. It illustrates how a problem can be expressed in terms of a smaller version of itself.

Mathematically, factorial is defined as:

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$$

This definition can be naturally translated into a recursive function:

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

A recursive function always consists of two essential components:

- **Base case:** `n == 1`, which stops the recursion
- **Recursive step:** `n * factorial(n-1)`, which reduces the problem

This example demonstrates how a complex computation can be defined in terms of a simpler instance of the same problem.

When a recursive function is executed, each function call creates a new execution context, known as a **stack frame**. These frames are organized in a structure called the **call stack**.

For example, evaluating `factorial(4)` produces the following sequence:

```
factorial(4)
-> 4 * factorial(3)
-> 4 * 3 * factorial(2)
-> 4 * 3 * 2 * factorial(1)
-> return 1
```

Once the base case is reached, the computation proceeds in reverse order. Each function call returns its result to the previous one, a process known as **stack unwinding**.

Each recursive call operates within its own local environment. This means that every function call has its own variables, and these variables do not interfere with one another.

Key observations:

- Each call maintains its own independent state
- Variables are local to each function call
- Control returns to the previous call after execution

This separation ensures that recursive functions behave correctly and predictably.

Iteration vs Recursion (Summary)

Repetition in programming can be expressed using either iteration or recursion. While both approaches can solve the same problems, they differ in how they model computation.

Iteration	**Recursion**
Uses loops	Uses function calls
Stores state in variables	Uses call stack
Step-by-step execution	Problem decomposition
Stops via condition	Stops via base case

Conceptually, iteration focuses on *how* to compute (step-by-step), while recursion focuses on *what* the problem is by expressing it in terms of smaller subproblems.

Both approaches can solve many of the same problems, but they reflect different ways of thinking.

Example: Reverse Number (Iteration vs Recursion)

Recursion can also be applied to transform numerical data. The difference between iteration and recursion becomes clear when we compare how each approach manages computation.

Iterative Version

```python
def reverse_num(num):
    rev = 0
    while num > 0:
        rev = rev * 10 + num % 10
        num = num // 10
    return rev
```

Key Idea

- Uses a loop to control repetition
- State stored in a variable (`rev`)
- State is updated explicitly

Recursive Version

```python
def reverse_num(num, rev=0):
    if num == 0:
        return rev
    else:
        return reverse_num(
            num // 10,
            rev * 10 + num % 10)
```

Key Idea

- Uses function calls instead of loops
- Managed by the **call stack**
- State is passed implicitly

Iteration stores state explicitly in variables, while recursion relies on the call stack to manage state implicitly.

Recursion on Strings: Palindrome

Recursion is not limited to numerical problems; it can also be applied to strings.

A **palindrome** is a string that reads the same forwards and backwards.

The recursive idea is simple:

- Compare the first and last characters
- Recursively check the substring in between

The base case occurs when the string has length 0 or 1, in which case it is a palindrome. In the recursive case, if the first and last characters match, the function continues checking the inner substring; otherwise, the string is not a palindrome.

Divide and Conquer

Recursion often follows the **divide-and-conquer** paradigm.

Break → Solve → Combine

This approach works by:

- Breaking a complex problem into smaller subproblems
- Solving each subproblem independently
- Combining the results to obtain the final solution

The key idea is that each subproblem is similar to the original problem, but simpler to solve.

5.1.5 Summary

Recursion is a powerful technique in which a function calls itself to solve a problem. It works by reducing a complex problem into smaller subproblems of the same form.

A correct recursive function must include:

- A **base case**, which stops the recursion
- A **recursive step**, which moves the problem toward the base case

Recursion operates by building a sequence of function calls and then resolving them through stack unwinding.

Most problems that can be solved using recursion can also be solved using iteration (and vice versa).

Break the problem into smaller parts, solve them, and combine the results.

Lecture 5-2: Python Dictionaries

5.2.1 Introduction to Compound Data Types

Python provides several built-in compound (non-scalar) data types such as lists, tuples, and dictionaries. Some of these structures were already introduced during the practice labs.

The main goal is not only to learn their syntax, but also to understand their similarities, differences, and appropriate use cases. Choosing the correct data structure can simplify programming problems significantly and lead to clearer, more efficient solutions.

Each compound data type organizes and accesses data differently:

- Lists store ordered collections of elements
- Tuples store immutable ordered data
- Dictionaries map keys to values

Understanding how and when to use these structures is an important step toward writing more effective Python programs.

5.2.2 What is a Dictionary?

A dictionary is a built-in Python data structure that stores data as **key–value pairs**. Instead of accessing elements by integer positions like lists, dictionaries allow direct access using meaningful keys.

```
student = {
    "name": "Alex",
    "id": 1023,
    "grade": "A"
}

print(student["name"])
```

Output:

```
Alex
```

Dictionaries help organize related information into one unified structure.

Creating Dictionaries

An empty dictionary can be created using curly braces:

```
my_dict = {}
```

Example dictionary:

```
grades = {
    'Ana': 'B',
    'John': 'A+',
    'Denise': 'A'
}
```

Each entry consists of:

- A **key**
- A corresponding **value**

Dictionary Lookup

Dictionary lookup is similar to indexing in a list, but dictionaries use keys instead of integer indices.

```
grades['John']
```

Output:

```
A+
```

If the key does not exist, Python raises a `KeyError`.

```
grades['Sylvan']
```

Dictionary Operations

Dictionaries support common operations such as adding, updating, deleting, and checking keys.

Adding or Updating

```
grades['Sylvan'] = 'A'
```

Checking Membership

```
'John' in grades
```

Output:

```
True
```

Deleting Elements

```
del(grades['Ana'])
```

Keys and Values

Dictionary keys and values have different properties.

Keys

- Must be unique
- Must be immutable
- Common types:
  - integers
  - strings
  - tuples
  - booleans

Values

- Can be any type
- Can be mutable or immutable
- Duplicates are allowed

Example:

```
d = {
    4: {1:0},
```

```
    (1,3): "twelve",
    'const': [3.14, 2.7]
}
```

Iterating Through Dictionaries

Python provides methods for accessing dictionary keys and values.

```
grades.keys()
grades.values()
```

Example execution loop:

```
for name in grades:
    print(name, grades[name])
```

Example Application: Word Frequency

Dictionaries are commonly used for counting and grouping data.

Example problem:

- Count how many times each word appears
- Find the most frequent word

A dictionary can map:

$$\texttt{word} \rightarrow \texttt{frequency}$$

This type of structure makes many data analysis tasks simpler and more efficient.

Comparison of List, Tuple, and Dictionary

Feature	**List**	**Tuple**	**Dictionary**
Structure	Ordered collection	Ordered collection	Key–value pairs
Mutability	Mutable	Immutable	Mutable
Can be modified?	Yes	No	Yes
Access Method	Integer index	Integer index	Key lookup
Duplicate Elements	Allowed	Allowed	Keys: not allowed
Syntax	`[ ]`	`( )`	`{ key:value }`
Typical Usage	Sequence of items	Fixed data	Fast lookup using labels
Example	`[1,2,3]`	`(1,2,3)`	`{"a":1}`

Although lists, tuples, dictionaries, and strings organize data differently, they all share one important property: they are iterable objects. This means we can use loops to access their elements one by one.

Summary

Python provides several built-in compound (non-scalar) data types such as lists, tuples, and dictionaries. Although these structures organize and access data differently, they all help simplify programming problems by storing related information efficiently.

Dictionaries are especially useful when data should be accessed using labels instead of numeric indices. They support common operations such as adding, updating, deleting, and searching for elements using keys.

Although lists, tuples, and dictionaries differ in structure and usage, they all support iteration. This means loops can access their elements one by one, which makes them convenient for processing collections of data.

More generally, many Python objects are iterable, including lists, tuples, dictionaries, strings, files, and more. Iteration is therefore one of the fundamental concepts used throughout Python programming.

Selecting the appropriate data structure is an important step toward effective problem solving and program design.

Lecture 6: Object-Oriented Programming in Python

Introduction

In this lecture, we move from simply using Python objects to understanding how they are structured and how we can create our own.

Python supports multiple programming paradigms, including imperative, functional, declarative, object-oriented, and event-driven programming. This flexibility allows us to choose the most suitable approach depending on the problem.

Python is a multi-paradigm programming language.

In this lecture, we focus on the object-oriented paradigm, which allows us to organize programs using objects and their interactions.

Everything in Python is an Object (Recap)

So far, we have worked with values such as integers, strings, lists, and dictionaries. However, an important conceptual shift is required:

Every value in Python is an object.

Each object consists of:

- A **type** (class)
- **Attributes** (data)
- **Methods** (behavior)

For example:

```
x = 10
type(x)          # <class 'int'>
isinstance(x, int)  # True
```

An object can be understood as a combination of data, behavior, and identity.

Objects and Abstraction

Objects provide a way to manage complexity through abstraction. The internal representation of an object is hidden, and only a set of operations (methods) is exposed.

For example, a list may have a complex internal structure, but we interact with it using methods such as `append()`, `pop()`, or `sort()`.

Focus on how to use an object, not how it is implemented.

Why Object-Oriented Programming?

Object-oriented programming helps us:

- Combine data and behavior
- Improve modularity
- Reuse code
- Reduce complexity
- Create multiple independent objects from one class

Each class represents an independent unit, which allows better organization and maintainability.

From Using Types to Creating Types

Until now, we have used built-in types such as `int`, `str`, and `list`. The next step is to create our own types.

In object-oriented programming, working with classes involves two complementary perspectives: **creating a class** and **using a class**. Understanding this distinction is essential for designing and working with custom types in Python.

(a) **Creating a Class (Defining a New Type)**

When we **create a class**, we define a new type. This involves specifying the class name, defining its attributes (data), and implementing its methods (behavior). In other words, we describe how objects of this type should behave and what operations can be performed on them.

For example, Python developers have already defined built-in classes such as `list`, which encapsulate both data storage and operations.

A class definition provides a blueprint:

```
class Coordinate(object):
    pass
```

Every class in Python ultimately derives from the base class `object`, even if it is not explicitly written.

Defining Attributes and Methods

A class defines both state and behavior:

- **Attributes** – represent the state of an object
- **Methods** – define the behavior of an object

Attributes define what an object knows, methods define what it can do.

Creating Data Attributes

To create data attributes for objects, we define a special method called `__init__`. This method serves two important purposes. First, it is automatically called when a new object of the class is created. Second, it is used to initialize the object's data attributes.

In other words, the `__init__` method defines how an object is constructed and what data it should contain when it is created.

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Here, `self` refers to the current object being created. The attributes `x` and `y` are assigned to the object, meaning that each instance of the class will store its own values for these attributes.

Here, `self` refers to the current object being created. Each object will have its own values for `x` and `y`.

At this point, since we have defined the constructor (`__init__`), we can now create objects of this class.

(b) **Using a Class (Working with Objects)**

Once a class is defined, we can use it to create objects (instances) and perform operations on them.

Creating an object:

```
c = Coordinate(3, 4)
```

Here, `c` is an instance of the `Coordinate` class. When this statement is executed, Python creates a new object and automatically calls the `__init__` method to initialize its attributes. As a result, the object `c` stores its own values for `x` and `y`.

Each object created from the class maintains its own independent state, even though all objects share the same class definition.

Creating/Defining Methods

Now we extend the class definition by adding behavioral methods. Previously, we defined the special method `__init__` to initialize objects. In addition to initialization, a class can define methods that describe what objects can do.

A method is a function defined inside a class that operates on objects. Every method must have at least one parameter, conventionally named `self`, which refers to the current instance. It allows the method to access and manipulate the data of that specific object, thereby distinguishing between different instances created from the same class.

The following example shows a complete class definition with a method:

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, other):
```

```
        return ((self.x - other.x)**2 +
                (self.y - other.y)**2)**0.5
```

In this method:

- `self` refers to the current object
- `other` refers to another object of the same class

This method computes the distance between two `Coordinate` objects by accessing their attributes.

Using Methods (Method Calls)

Once a method is defined inside a class, it can be called using an object (instance) of that class.

```
c1 = Coordinate(3, 4)
c2 = Coordinate(0, 0)

d = c1.distance(c2)
print(d)
```

When we call `c1.distance(c2)`, Python automatically passes `c1` as the first argument (i.e., `self`) and `c2` as the second argument (i.e., `other`).

In other words, the call:

```
c1.distance(c2)
```

is internally interpreted as:

`Coordinate.distance(c1, c2)`

This means that methods are simply functions where the object is passed automatically as the first argument.

The method then computes the distance between the two objects by accessing their attributes.

Visualizing Method Execution

```
c1 (3,4)          c2 (0,0)

c1.distance(c2)

self  -> c1
other -> c2

distance = sqrt((3-0)^2 + (4-0)^2) = 5
```

Objects interact by calling methods, and `self` is passed automatically by Python.

With these examples, we can clearly understand how classes are both defined and used. Creating a class establishes a new type by describing its structure and behavior, whereas using a class means creating objects and working with them through their methods.

Design the blueprint first, then create and use its instances.

Type and Instance Checking

We can inspect objects using the following functions, just as we used with built-in types:

```
type(c)
isinstance(c, Coordinate)
```

- `type()` returns the class of the object
- `isinstance()` checks if an object belongs to a class

These are built-in functions provided by Python, and they can be applied to both built-in and user-defined objects. This consistency arises from Python's unified object model, where every value is an object and every class ultimately derives from the base class `object`. Consequently, these operations behave uniformly across all types.

Improving Object Representation

Special Methods and Custom Behavior

Python provides a set of special methods (also called *dunder methods*) that allow us to customize how objects behave and interact with built-in operations.

For example:

- `__add__` defines behavior for the + operator
- `__eq__` defines equality comparison (==)
- `__len__` defines the behavior of `len()`
- `__str__` defines how objects are displayed

By implementing these methods, custom classes can behave similarly to built-in types.

These methods are part of Python's object model and are available because all classes ultimately derive from the base class `object`. However, their behavior is not automatically meaningful—we must explicitly define or override them to control how our objects interact with Python's syntax and built-in functions.

(a) **Example: Improving Object Representation with `__str__`**

By default, printing an object produces an uninformative result:

```
print(c)
# <__main__.Coordinate object at ...>
```

This output only shows the object's type and memory location, which is not useful for understanding its content.

We can improve this by defining the `__str__` method inside the class:

```
def __str__(self):
    return f"<{self.x},{self.y}>"
```

Now it prints:

```
print(c)  # <3,4>
```

The `__str__` method defines how an object is converted to a string when it is printed. It must always return a string value.

(b) **Example: Defining Equality with `__eq__`**

In addition to controlling how objects are displayed, we can also define how they are compared.

By default, comparing two objects using == checks whether they refer to the same object in memory, not whether their contents are equal.

```
c1 = Coordinate(3, 4)
c2 = Coordinate(3, 4)

print(c1 == c2)   # False (by default)
```

Although `c1` and `c2` have the same values, they are different objects, so the result is `False`.

We can redefine this behavior by implementing the `__eq__` method:

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y
```

Now:

```
c1 = Coordinate(3, 4)
c2 = Coordinate(3, 4)

print(c1 == c2)   # True
```

The `__eq__` method defines what it means for two objects to be equal by comparing their attributes. Equality is no longer based on identity, but on the data stored in the objects.

Special methods allow custom objects to integrate naturally with Python's syntax and built-in operations.

Summary

Python supports multiple programming paradigms, but at its core, everything is built around objects. Every value in Python is an object that combines both data and behavior. Classes define new types, while objects are instances of those types, representing real, independent entities in a program. Methods describe how these objects behave, and special methods allow us to customize how objects interact with each other and with the language itself.

You are no longer just writing code—you are organizing ideas into interacting objects.

Lecture 7: Advanced Concepts in Object-Oriented Programming

Introduction

In the previous lecture, we introduced the fundamentals of object-oriented programming, including classes, objects, attributes, and methods. In this lecture, we build upon those foundations and explore more advanced concepts.

These concepts enable us to design programs that are more robust, maintainable, and reusable. As programs grow in complexity, it becomes increasingly important to structure code in a way that supports extension, clarity, and modularity.

In particular, we revisit key ideas from earlier lectures and extend them to understand how objects behave and interact in more complex systems.

OOP Basics(Recap)

In Python, everything is an object. Each object has:

- a type (class)
- data (attributes)
- behavior (methods)

Objects are instances of classes, and classes act as blueprints. A single class definition can be used to create multiple independent objects that share the same structure but hold different data.

Design once, create many, use independently.

Two Perspectives: Implementing vs Using a Class

When working with classes, we can view them from two perspectives:

Implementing a Class

- Define a new object type
- Specify attributes and methods
- Focus on design and structure

Using a Class

- Create objects (instances)
- Call methods
- Perform operations

Building the tool vs using the tool

Modeling the Real World

Object-oriented programming allows us to model real-world entities such as students, animals, or coordinates.

Each class represents a concept, and objects represent specific instances of that concept.

Real world → program model

Objects: State and Behavior

An object consists of two main components:

Data (State)

- Represents what the object is
- Example: `x, y` for coordinates

Behavior (Methods)

- Defines what the object can do
- Example: `distance()`

Object = Data + Behavior

Getter and Setter Methods

In object-oriented design, direct access to data attributes is discouraged. Instead, we define methods inside the class to control access to these attributes.

- **Getters** retrieve values
- **Setters** modify values

These methods are defined within the class:

```
class Animal(object):
    def __init__(self, age):
        self.age = age

    def get_age(self):
        return self.age

    def set_age(self, newage):
        self.age = newage
```

They are then used outside the class to access and modify data:

```
a = Animal(3)
print(a.get_age())
a.set_age(5)
```

Why Use Getters and Setters?

Getters and setters provide controlled access to an object's data, allowing us to manage how attributes are read and modified. Instead of exposing internal data directly, these methods act as an interface between the object and the outside world. This design offers flexibility, as the internal implementation of the class can change without affecting external code that relies on these methods. As a result, programs become easier to maintain, less error-prone, and more adaptable to future modifications.

As we know, objects are created through instantiation:

```
a = Animal(3)
```

Attributes and methods are accessed using dot notation:

```
a.age
a.get_age()
```

Although direct access is allowed, using getters and setters is considered better practice because they provide controlled access to object data.

However, explicitly calling methods such as:

```
a.get_age()
a.set_age(10)
```

can make code less natural and less Pythonic.

Python provides the **`@property`** decorator to solve this problem. It allows methods to be accessed like regular attributes while still preserving the benefits of getters and setters.

```
class Animal(object):
    def __init__(self, age):
        self._age = age

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, newage):
        self._age = newage
```

Now the object can be used naturally:

```
a = Animal(3)

print(a.age)    # getter
a.age = 10      # setter
```

In this approach, the attribute appears to be accessed directly, but Python internally calls the corresponding getter and setter methods.

Use attributes like variables, but control them like methods.

Information Hiding

Information hiding is a key principle in OOP.

If internal representation changes, external code should not break.

For example, if:

```
self.age  ->  self.years
```

then direct access `a.age` will fail.

Using:

```
a.get_age()
```

ensures that the interface remains stable.

Hide implementation details, expose a stable interface

Python Limitation: Python does not strictly enforce information hiding. In practice, object attributes can be accessed and modified directly from outside the class. While this flexibility makes Python easy to use, it also means that the responsibility for maintaining proper design lies with the programmer. Directly accessing or modifying internal data is generally considered poor programming practice, as it can lead to fragile code and make future changes more difficult.

Default Arguments

Methods in a class can define default parameter values, just like regular functions in Python. A method is essentially a function defined inside a class, so it follows the same rules for parameters.

```
class Person(object):
    def __init__(self, name=""):
        self.name = name

    def set_name(self, newname=""):
        self.name = newname
```

If no argument is provided when the method is called, the default value is used.

```
p = Person()
p.set_name()       # uses default value ""
p.set_name("John") # overrides default
```

This behavior is identical to functions defined outside a class. The only difference is that methods include `self` as the first parameter, which refers to the instance of the class.

Inheritance and Class Hierarchies

Inheritance allows a class to derive from another class, enabling code reuse and hierarchical organization. In Python, all classes ultimately inherit from the built-in `object` class. This creates a hierarchy where more specific classes extend more general ones. For example, `Animal` can be a parent class, and `Cat` can be a child class that inherits from it.

Parent Class (Superclass)

- Defines general behavior
- Serves as a base for other classes

Child Class (Subclass)

- Inherits data and behavior from the parent
- Can add new features
- Can override existing behavior

A child class extends and specializes the parent class.

Example: Subclass

```
class Cat(Animal):
    def speak(self):
        print("meow")
```

The `Cat` class inherits all methods from `Animal` but also defines its own specific behavior.

Method Resolution

When a method is called:

- Python first looks in the current class
- Then moves up the class hierarchy
- Uses the first matching method found

Inheritance helps organize code into reusable and maintainable structures.

Multiple Inheritance

Python also supports **multiple inheritance**, where a class can inherit from more than one parent class.

```
class A(object):
    def hello(self):
        print("A")
```

```
class B(object):
    def hello(self):
        print("B")

class C(A, B):
    pass
```

In this example, class `C` inherits from both `A` and `B`.

A natural question arises:

> *If both parent classes define the same method, which one should Python use?*

Python resolves this using **Method Resolution Order (MRO)**.

Method Resolution Order (MRO)

MRO defines the order in which Python searches classes for methods and attributes.

```
c = C()

c.hello()
```

Output:

```
A
```

Python chooses the method from class `A` because `A` appears before `B` in the inheritance list.

The search order can be inspected using:

```
print(C.__mro__)
```

Example output:

```
(<class '__main__.C'>,
 <class '__main__.A'>,
 <class '__main__.B'>,
 <class 'object'>)
```

This means Python searches in the following order:

$$C \rightarrow A \rightarrow B \rightarrow object$$

> *MRO ensures that Python resolves method conflicts in a consistent and predictable way.*

Class Variables

Class variables are variables defined directly inside a class (not inside a method). They are shared among all instances of that class.

```
class Rabbit(object):
    tag = 1   # class variable

    def __init__(self, age):
        self.age = age
        self.rid = Rabbit.tag
        Rabbit.tag += 1
```

In this example, `tag` is a class variable that keeps track of the number of objects created. Each time a new `Rabbit` object is instantiated, it is assigned a unique identifier stored in `rid`, and the shared `tag` value is incremented.

Usage Example:

```
r1 = Rabbit(2)
r2 = Rabbit(3)

print(r1.rid)  # 1
print(r2.rid)  # 2
```

Class variables are shared across all instances, while instance variables belong to individual objects.

Operator Overloading

As we discussed previous class, we can customize how objects behave with built-in operators by defining special methods inside a class. This allows user-defined objects to interact naturally using standard syntax such as + and ==.

Consider the following example:

```
class Rabbit(object):
    def __init__(self, age, parent1=None, parent2=None):
        self.age = age
        self.parent1 = parent1
        self.parent2 = parent2

    def __add__(self, other):
        return Rabbit(0, self, other)

    def __eq__(self, other):
        return (self.parent1 == other.parent1 and
                self.parent2 == other.parent2)
```

The `__add__` method defines how two objects interact using the + operator. In this example, adding two `Rabbit` objects creates a new `Rabbit` instance, representing an offspring whose parents are the two operands.

The `__eq__` method defines how objects are compared using the == operator. Here, two `Rabbit` objects are considered equal if they have the same parents.

Usage Example:

```
r1 = Rabbit(2)
r2 = Rabbit(3)

baby = r1 + r2       # calls __add__
print(baby.age)      # 0

print(r1 == r2)      # calls __eq__
```

Special methods allow custom objects to behave like built-in types by integrating with Python's operators.

Summary

Object-oriented programming allows us to organize data and behavior together into cohesive units, model real-world systems more naturally, and build scalable and reusable code. By structuring programs around objects and their interactions, we move beyond writing isolated instructions and instead design systems composed of well-defined components. Classes provide abstraction and decomposition, similar to how functions structure programs, enabling us to manage complexity effectively.

Object-oriented programming models real-world entities by combining data and behavior within classes. Getters and setters provide controlled access to object data, supporting the principle of information hiding, which improves maintainability and reduces the risk of errors. Inheritance enables code reuse and extension by allowing new classes to build upon existing ones. Class variables allow data to be shared across instances, while special methods enable custom behavior and seamless integration with Python's built-in operations. Together, these concepts form a powerful approach to designing clear, modular, and maintainable programs.

You are not just writing programs—you are designing structured systems of interacting objects.

Lecture 8

Lecture 8-1: Code Organization and Modularity in Python

At its core, a Python module is simply a file containing source code. It serves as a logical container that can define functions, classes, and variables, as well as any executable statements. By organizing an application into distinct modules, developers can break down a monolithic codebase into a structured collection of reusable, maintainable units.

8.1.1 Types of Python Modules

Python groups modules into three main categories:

(a) **Built-in Modules**: These are baked directly into Python's standard library. They are always available to you without any extra installation.

- *Examples*: `math`, `random`, `os`, `sys`.

(b) **Third-party Modules**: Created by external developers and the community. These must be downloaded and installed using package managers like `pip`.

- *Examples*: `requests` (for HTTP requests), `pandas` (for data manipulation), `matplotlib` (for visual plotting).

(c) **Custom Modules**: Created by *you*! Any `.py` file you write can be imported as a custom module in another file.

8.1.2 Using built-in Modules

Operating System Interactivity: The `os` Module

Python's built-in `os` module provides a cross-platform way to interact directly with your underlying operating system (Windows, macOS, or Linux). However, because Unix-like systems and Windows handle security and storage paths differently, you should always test your path and permission-related code on the target OS.

Here are the most common ways to navigate and manage directories programmatically:

```
import os

# 1. Get the current process ID
print(os.getpid()) # Returns the OS process ID for this Python
    script

# 2. Get Current Working Directory (Equivalent to 'pwd')
print(os.getcwd())

# 3. List items inside a specific directory (Equivalent to 'ls')
print(os.listdir('/home/p4'))
```

```

# 4. Create a new directory
os.mkdir("/home/p4/dir_directory")

# 5. Remove an empty directory
os.rmdir("/home/p4/dir_directory")
```

You can also run raw shell commands using `os.system()`:

```
import os
status = os.system("ls /home/p4") # Executes the command in the
    system shell
```

> **Note on Exit Codes**: `os.system()` returns an integer code. An exit code of `0` means the command ran successfully. Any non-zero value indicates a system error or warning.

8.1.3 Using a third-party module

1. Intalling module

A **package manager** is a system tool responsible for installing, updating, configuring, and removing software packages on your operating system or within your runtime environment.

- **System Package Managers (e.g., `apt` on Ubuntu)** operate at the OS level. For instance, installing or removing system-wide tools look like this:

```
sudo apt install ssh          # Installs SSH
sudo apt remove openssh-server # Removes OpenSSH Server
```

- **Language-Specific Package Managers (e.g., `pip`)** are dedicated entirely to downloading and managing external libraries and frameworks for Python.

```
pip install pygame      # Downloads and installs Pygame
pip list                # Lists all currently installed
   Python packages
pip uninstall pygame    # Uninstalls Pygame
```

2. Importing Modules & Troubleshooting

Once installed, you can bring these tools into your namespace using different syntaxes based on your preferences:

- **Method 1: Importing the entire module**

```
import requests
response = requests.get("https://www.google.com")
print(response.status_code) # Output: 200
```

- **Method 2: Importing specific attributes/functions**

```python
from requests import get
response = get("https://www.google.com")
print(response.status_code) # Output: 200
```

Note: Like this, we can also import a specific part of the built-in module.

Fixing `ModuleNotFoundError`

If you attempt to import a library that hasn't been installed yet, Python raises a `ModuleNotFoundError`.

```python
import pygame
# Raises: ModuleNotFoundError: No module named 'pygame'
```

The Fix: Verify if the package is truly missing using `pip list`, and then install it:

```bash
pip list | grep pygame   # Check if installed
pip install pygame       # Install the missing module
```

8.1.4 Creating and Importing Custom Modules

Let's look at how easily you can split your application across files.

Imagine we have a file called `mymodule.py`:

Listing 1: mymodule.py

```python
# mymodule.py
var = 5
lst = [1, 2, 3]

def func(var):
    print(var)
```

We can import and use these elements inside a separate executable script, `main.py`:

Listing 2: main.py

```python
# main.py
import mymodule
from mymodule import lst

print(mymodule.var)  # Prints: 5
print(lst)           # Prints: [1, 2, 3]
mymodule.func(6)     # Prints: 6
```

Lecture 8-2: Managing Python Environments

In modern software development, managing external dependencies is a critical challenge. When developing multiple Python projects simultaneously on a single machine, you will inevitably

encounter dependency version conflicts. For example, `Project A` might rely on an older legacy version of a library like NumPy (e.g., version 1.15) to maintain backward compatibility, while `Project B` requires a modern release (e.g., version 1.26) to utilize new performance optimizations.

Installing these conflicting libraries globally creates a system-wide clash, as a single Python installation can only map to a single version of a package at any given time. To resolve this "dependency hell," developers use **isolated virtual environments**. A virtual environment is a self-contained directory tree that contains its own Python installation, its own standard library, and its own set of isolated packages.

To create and manage these isolated spaces, Python developers rely on a variety of built-in and third-party tools. These environment managers generally fall into two distinct categories:

- **Lightweight Virtual Environment Creators (e.g., `venv`, `virtualenv`):** These tools focus strictly on isolating the Python runtime environment. They rely on external package installers like `pip` to download and install packages from the Python Package Index (PyPI) inside the isolated directory.
- **Full-Scale Package and Environment Managers (e.g., Conda, Poetry):** These comprehensive systems manage both the runtime environment and the complex dependency resolution process. They often handle non-Python binary dependencies, library conflicts, and project packaging pipelines within a single, unified workflow.

8.2.1 Step-by-Step Practical Workflows

To cement these concepts, let us look at two hands-on examples. We will perform the exact same workflow in both tools: creating an isolated environment, installing a specific version of a library, exporting our configuration, and cleaning up.

Workflow A: Using the Python Standard (`venv` & `pip`)

This workflow uses only standard Python tools built directly into your installation.

(a) **Create the project folder and navigate inside:**

```
mkdir my_project && cd my_project
```

(b) **Create a new virtual environment:** Here, we use the standard library module `venv` (which works identically to `virtualenv`) to create an environment folder named `env_pip`.

```
python3 -m venv env_pip
```

(c) **Activate the environment:** This tells your terminal shell to look inside your virtual environment folder for Python executable binaries and package files.

- On **Linux / macOS**:

```
source env_pip/bin/activate
```

- On **Windows** (Command Prompt):

```
env_pip\Scripts\activate.bat
```

Note: Once activated, your terminal prompt will change to show `(env_pip)` *at the beginning.*

(d) **Install a package:** Let us install a specific, historical version of the popular data processing library `pandas`.

```
pip install pandas==1.5.3
```

(e) **Verify the installation:** We can write and run a quick Python inline test command to verify that the library is correctly installed and showing the version we requested.

```
python -c "import pandas; print(pandas.__version__)"
```

(f) **Export the dependency list (Reproducibility):** Save your environment setup to share it with team members.

```
pip freeze > requirements.txt
```

(g) **Exit the virtual environment:** When you are done working, return your terminal settings back to normal.

```
deactivate
```

Workflow B: Using the Data Science Standard (`conda`)

This workflow uses Conda, which manages both Python itself and non-Python system binary packages.

(a) **Create a new environment:** We will create a fresh, isolated sandbox explicitly configured with a specific Python version (e.g., Python 3.9) and name it `env_conda`.

```
conda create --name env_conda python=3.9
```

Note: Conda will ask you to confirm installing standard default tools; simply type `y` *(yes) to proceed.*

(b) **Activate the environment:** This modifies your system path variables to target Conda's virtual environment tree.

```
    conda activate env_conda

```

Note: Your terminal prompt will now change to show `(env_conda)`.

(c) ***Install a package:*** *Let us install the specialized visualization plotting library* `matplotlib`.

```
conda install matplotlib=3.7.1
```

(d) ***Verify the installation:*** *Just like before, we verify the package version dynamically using an inline Python command.*

```
python -c "import matplotlib; print(matplotlib.__version__)"
```

(e) ***Export the dependency list (Reproducibility):*** *Export both the packages and the core system specifications (like the core Python version) into a YAML file.*

```
conda env export > environment.yml
```

(f) ***Deactivate the environment:*** *Exit back to your machine's base environment.*

```
conda deactivate
```

(g) ***Clean up (Optional):*** *If you no longer need the project, you can delete the entire environment from your computer's storage with a single command.*

```
conda env remove --name env_conda
```

Note: While `pip` is a language-specific system built exclusively to distribute Python packages, `conda` is a cross-language manager designed to handle complex system dependencies across multiple programming applications.

Summary

In this lecture, we explored how Python achieves maintainability, scalability, and stability through modular design and isolated execution environments.

Key takeaways from this session include:

- **Modular Architecture**: Python code is structured across built-in modules (`os`, `sys`), third-party packages installed via package managers, and custom `.py` scripts. Modularizing code promotes reusability and keeps large software projects organized.
- **System Package Management**: Tools like `pip` handle language-specific dependencies from PyPI, whereas system managers (`apt`) and cross-language managers (`conda`) handle low-level OS dependencies and binary packages.
- **Environment Isolation**: Virtual environment tools (`venv`, `conda`) solve "dependency hell" by isolating Python runtimes, package versions, and libraries per project.
- **Reproducibility**: Tracking project dependencies via specification files (`requirements.txt` for `pip` or `environment.yml` for `conda`) ensures that code behaves consistently across different machines and deployment environments.

  > **Best Practice for Upcoming Lectures**: As we transition into specialized practical domains—such as web development, data science, and machine learning—always create a dedicated virtual environment for each new project. This simple habit keeps your local machine clean, prevents version collisions between heavy libraries (e.g., PyTorch, TensorFlow, Pandas), and ensures your projects remain fully reproducible.

Lecture 9: File Handling in Python

Introduction

In previous lectures, we learned that everything in Python is an object. This idea extends naturally to file handling as well.

A file in Python is also an object.

This means that files follow the same principles we have already studied: they are created, used, and eventually removed from memory. Understanding file handling becomes easier when we see it as part of Python's unified object model

Object Lifecycle Revisited

So far, we have seen that objects go through a lifecycle:

- Creation
- Usage
- Destruction (garbage collection)

File objects follow exactly the same pattern, which allows us to connect new knowledge with what we already understand.

File Objects in Python

A file object represents a connection between a Python program and a file stored on disk.

The lifecycle of a file object consists of three main steps:

- **Open (Create)**: establish a connection to the file.
- **Process (Use)**: read from or write to the file
- **Close (Cleanup)**: release the connection

```
f = open("data.txt", "w")
f.write("Hello")
f.close()
```

If a file is not properly closed, it may lead to resource issues. Therefore, managing this lifecycle correctly is important.

Safer File Handling with `with` in Python

Python provides a safer and cleaner way to manage files using the `with` statement. This approach handles all three steps automatically.

```
with open("data.txt", "w") as f:
    f.write("Hello")
```

In this structure, the file is opened, used, and automatically closed when the block is exited. Even if an error occurs during execution, the file will still be properly closed. Even if an error occurs, the file will be properly closed.

$$\textit{Open} \rightarrow \textit{Process} \rightarrow \textit{Automatic Cleanup}$$

This makes file handling more reliable and easier to write.

9.3.1 File Handling Methods

Python provides several methods for reading and writing files. The choice depends on how much data we want to process.

Reading methods:

- `read()` reads the entire file as a single string
- `readline()` reads one line at a time
- `readlines()` reads all lines into a list

Writing methods:

- `write()` writes a string to the file
- `writelines()` writes multiple lines

For large files, iterating line by line is recommended, as it is more memory-efficient.

9.3.2 File Paths

To open a file, Python must know where the file is located. This location is specified using a **file path**.

There are two main types of file paths:

- **Relative path**: based on the current working directory
- **Absolute path**: full path starting from the root directory

Example of a relative path:

```
with open("data/file.txt")
```

Incorrect paths are a common source of errors. Therefore, it is important to understand where the program is running and how paths are interpreted.

Cross-Platform Path Issues

Different operating systems use different path formats:

- Windows commonly uses \
- Linux and macOS commonly use /

Because of these differences, file paths may break when code is moved between systems.

Solution

Python provides modules such as `os.path` and `pathlib` to create portable paths that work across operating systems.

Example Using `os.path`

```
import os

path = os.path.join("data", "file.txt")

with open(path, "r") as f:
    print(f.read())
```

Example Using `pathlib` (recommended)

```
from pathlib import Path

path = Path("data") / "file.txt"

with open(path, "r") as f:
    print(f.read())
```

These modules automatically handle operating-system-specific path formats and help create more portable and reliable programs.

> *Always know where your program is running and how file paths are constructed.*

File Modes

When opening a file, Python requires a mode that specifies how the file will be used.

- `"r"` — read an existing file
- `"w"` — write (create or overwrite)
- `"a"` — append to a file
- `"rb"` — read in binary mode

```
with open("data.txt", "a") as f:
    f.write("New line")
```

Choosing the correct mode is important for correct program behavior.

Iterable Objects and Files

An iterable object can return its elements one by one. Common iterable objects include lists, tuples, strings, dictionaries, and files.

A file object is also iterable, allowing efficient line-by-line processing:

```
with open("file.txt") as f:
    for line in f:
        print(line)
```

This approach is memory efficient and widely used for processing large files.

Managing Files with the `os` Module

The `os` module allows Python to interact with the operating system. It can be used to manage files and directories.

Common operations include:

- renaming files
- removing files
- creating directories
- executing system commands

```python
import os
os.rename("example.txt", "example2.txt")

# Remove a file
os.remove("example2.txt")

# Create a new directory
os.mkdir("new_folder")

# Check if a file exists
os.path.exists("data.txt")

# Execute a system command
os.system("echo Hello from OS")
```

These operations allow Python programs to interact directly with the underlying operating system and manage files dynamically.

The `os` module bridges Python programs and the operating system.

Summary

File handling in Python follows the same principles as working with any other object. A file is created, used, and eventually removed, following a clear lifecycle. Python provides structured tools such as `with open(...)` to manage this process safely and efficiently.

Files are not special—they are iterable objects that can be processed using the same techniques as lists or strings. Understanding this connection simplifies the learning process and reinforces the idea that Python is built on consistent design principles.

A file is just another object—once you recognize this, file handling becomes a natural extension of what you already know.

Lecture 10: Testing, Debugging, Exceptions, and Assertions

Introduction: Why Do Programs Fail?

Programs may fail for many reasons, including incorrect syntax, unexpected runtime conditions, or logical mistakes in the implementation. Writing correct code is only part of programming; ensuring that code behaves reliably under different conditions is equally important.

To achieve this, programmers use a combination of techniques:

- Prevent errors during development
- Detect errors through testing
- Identify and fix errors through debugging
- Handle unexpected situations using exceptions

Reliable programming is a systematic process: detect, understand, and prevent errors.

Types of Errors in Python

There are three main categories of errors:

- **Syntax Errors:** These occur when the program violates Python's grammar rules and prevent execution.

```
if x > 5
    print(x)
```

(Missing colon results in a syntax error)

- **Runtime Errors (Exceptions):** These occur during execution and may cause the program to terminate.

```
x = 10 / 0
```

(Division by zero raises a `ZeroDivisionError`)

- **Logical Errors:** The program runs without crashing but produces incorrect results.

```
def square(x):
    return x * 2   # incorrect logic
```

(The function runs, but returns wrong result)

Each type requires a different strategy for detection and correction.

3 Three Related Activities in Software Reliability

Ensuring program correctness is not a single step, but a continuous process involving three closely related activities: defensive programming, testing, and debugging. Each plays a distinct role in improving the reliability of software.

10.3.1 Defensive Programming

Defensive programming focuses on preventing errors before they occur. The goal is to design code that anticipates potential problems and handles them proactively.

- Write clear specifications for functions (inputs, outputs, assumptions)
- Break programs into smaller, modular components
- Validate inputs and check assumptions
- Use assertions to detect violations early

This approach reduces the likelihood of bugs and makes programs easier to understand and maintain.

10.3.2 Testing

Testing is the process of verifying that a program behaves as expected by comparing its actual results with the intended specification.

- Design test cases based on expected inputs and outputs
- Consider normal cases, boundary cases, and edge cases
- Use systematic approaches such as unit and integration testing
- Ask: *"How can I break this program?"*

Testing is often associated with the phase after development, when the completed program is evaluated. However, in practice, testing should be performed continuously throughout the development process.

- During development, test individual components (**unit testing**)
- After combining components, test their interaction (**integration testing**)
- After implementation, validate the system against requirements (**system testing**)

This continuous approach helps detect errors early, reduces debugging effort, and improves overall software quality.

Types of Testing

- **Unit Testing:** Tests individual functions or components in isolation
- **Regression Testing:** Ensures previously fixed bugs do not reappear
- **Integration Testing:** Tests how different components work together

Testing Approaches

- **Black-box Testing:** Based on the specification without inspecting the implementation
- **Glass-box Testing:** Based on the internal structure and logic of the code

Examples: Black-box vs Glass-box Testing

Consider the following function:

```
def abs_value(x):
    """Returns the absolute value of x"""
    if x < 0:
        return -x
    else:
        return x
```

Black-box Testing (Specification-based)

We design test cases based only on the expected behavior (what the function should do), without looking at the implementation.

- Input: `5` Expected output: `5`
- Input: `-3` Expected output: `3`
- Input: `0` Expected output: `0`

Focus: Does the function return correct results for different types of inputs?

Glass-box Testing (Implementation-based)

We design test cases by examining the code structure and ensuring all branches are executed.

- Test case for `x < 0`: `x = -3`
- Test case for `x >= 0`: `x = 5`
- Boundary test: `x = 0`

Focus: Does every path and condition in the code execute correctly?

Black-box testing focuses on expected behavior, while glass-box testing focuses on code structure.

Testing helps detect errors but does not explain why they occur.

10.3.3 Debugging

Debugging begins after an error has been detected. It is the process of identifying the root cause of incorrect behavior and fixing it.

- Analyze program behavior and error messages

- Form hypotheses about possible causes
- Test hypotheses through controlled experiments
- Modify the code and verify the fix

Debugging is a systematic and iterative process that requires careful reasoning. It is similar to a scientific method:

- Observe the program's behavior
- Form a hypothesis about the cause
- Test the hypothesis and refine the solution

Common debugging tools:

- Print statements
- Debuggers
- Step-by-step execution tools

Defensive programming prevents errors, testing reveals errors, and debugging explains and fixes errors.

From Debugging to Error Handling

Debugging helps us find and fix errors during development. However, not all errors can be predicted or eliminated. Some issues arise only during program execution, often due to unexpected input or external conditions.

How can a program handle unexpected errors at runtime?

This leads to the concept of **exception handling**.

Exceptions

Exceptions are runtime errors raised when a program encounters unexpected or invalid conditions during execution.

- `ValueError`: invalid value
- `TypeError`: incompatible types
- `IndexError`: invalid index

Instead of crashing, programs can use exceptions to respond to such situations in a controlled manner.

10.4.1 Exception Handling in Python

Python provides structured error handling using the `try/except` mechanism. Built-in exception classes (such as `ValueError`, `TypeError`, and `ZeroDivisionError`) allow programs to handle specific types of runtime errors.

- **try**: contains code that may fail

- **except**: handles the error
- **else**: executes if no error occurs
- **finally**: always executes (cleanup)

Example:

```
try:
    x = int(input("Enter a number: "))
    y = int(input("Enter another number: "))
    result = x / y
except ValueError:
    print("Invalid input")
except ZeroDivisionError:
    print("Cannot divide by zero")
else:
    print("Result:", result)
finally:
    print("Execution finished")
```

Explanation:

- The `try` block contains operations that may raise exceptions
- The `except` blocks handle specific errors
- The `else` block runs only if no exception occurs
- The `finally` block runs regardless of success or failure

Exception handling allows programs to remain stable and respond gracefully to runtime errors.

Beyond Handling Exceptions

So far, we have focused on handling exceptions raised by Python. However, errors are not only generated by the system—sometimes they arise from the logic of our own programs.

What should we do if our program detects an invalid situation?

For example, a function may receive input for which it cannot produce a meaningful result. In such cases, the program should explicitly signal the error rather than continue with incorrect behavior.

This leads to the concept of **raising exceptions**.

10.5.1 Raising Exceptions

Exceptions can be triggered manually using the `raise` statement.

- Used when a function cannot produce a valid result
- Clearly communicates an error condition to the caller

Example:

```
def sqrt(x):
    if x < 0:
        raise ValueError("Cannot compute square root of
   negative number")
    return x ** 0.5
```

Explanation:

- The function checks whether the input is valid
- If the input is invalid, it raises an exception
- The caller is notified that the operation cannot be performed

Raising exceptions ensures that errors are reported clearly instead of producing incorrect results.

Custom Exceptions

Built-in Exception Classes in Python: Python provides many built-in exception classes, each representing a specific type of runtime error.

Examples:

- `ValueError`: invalid value
- `TypeError`: incompatible type
- `ZeroDivisionError`: division by zero
- `IndexError`: invalid index

These predefined exceptions allow programs to detect and handle common error conditions.

While built-in exceptions cover many general cases, they may not fully capture the logic of every application.

What if our program has its own rules and constraints?

In such cases, we need a way to define our own error types.

Custom exceptions: Python allows us to define custom exception classes to represent application-specific errors.

- Custom exceptions are created by inheriting from `Exception`
- They make error handling more precise and meaningful

Example: Defining and Using a Custom Exception

```
class NegativeAgeError(Exception):
    pass

def set_age(age):
    if age < 0:
```

```
        raise NegativeAgeError("Age cannot be negative")
    return age
```

Handling the custom exception:

```
try:
    set_age(-5)
except NegativeAgeError:
    print("Invalid age provided")
```

Explanation:

- A custom exception `NegativeAgeError` is defined
- The function raises this exception when a rule is violated
- The caller handles the exception explicitly

Built-in exceptions describe general errors, while custom exceptions represent application-specific logic.

From Handling Errors to Preventing Bugs

Exception handling allows programs to respond to runtime errors and continue execution gracefully. However, not all errors come from external conditions—some indicate flaws in our program logic.

How can we ensure that our assumptions about the program are always valid?

This leads to the concept of **assertions**.

Assertions vs Exceptions: Exceptions are used to handle unexpected situations that occur during program execution, such as invalid input or runtime failures. They allow programs to respond to errors gracefully and continue operating when possible. In contrast, assertions are used to check assumptions about the program during development. They help detect logical errors by stopping execution when a condition that should always be true is violated. In summary, exceptions are designed to handle runtime errors, while assertions are intended to detect bugs in program logic.

Exceptions handle errors; assertions detect bugs.

Assertions are used to verify that certain conditions hold true during execution.

- If the condition is false, Python raises an `AssertionError`
- They are useful for detecting bugs early in development

Common uses of assertions:

- Validate input assumptions
- Check invariants in data structures
- Ensure correctness of intermediate results

Example: Using Assertions

```python
def avg(grades):
    assert len(grades) > 0, "no grades data"
    return sum(grades) / len(grades)

# Valid input
print(avg([80, 90, 100]))

# Invalid input
print(avg([]))
```

Output: 90.0 Traceback (most recent call last): ... AssertionError: no grades data

Explanation:

- The assertion checks that the input list is not empty
- For valid input, the function returns the correct average
- For invalid input, the assertion fails and raises an `AssertionError`

Summary

This lecture introduced the fundamental concepts required to write reliable and robust programs. Writing correct code is not only about implementing functionality, but also about preventing, detecting, and handling errors effectively.

Key Concepts

- **Types of Errors:**
  - Syntax errors prevent execution
  - Runtime errors (exceptions) occur during execution
  - Logical errors produce incorrect results without crashing
- **Defensive Programming:** Writing code that anticipates potential issues by validating inputs, documenting assumptions, and structuring programs into smaller components.
- **Testing:** Verifying that the program behaves according to its specification. Testing should be performed continuously during development and includes unit, regression, and integration testing.
- **Debugging:** A systematic process of identifying, analyzing, and fixing the causes of incorrect behavior. It involves forming hypotheses and validating them through experimentation.
- **Exceptions and Exception Handling:** Exceptions are runtime errors raised when unexpected or invalid conditions occur. Python provides structured handling using `try`, `except`, `else`, and `finally`.
- **Raising Exceptions:** The `raise` statement allows programs to signal error conditions explicitly when a valid result cannot be produced.

- **Custom Exceptions:** Custom exception classes enable programmers to represent application-specific errors, improving clarity and maintainability.
- **Assertions:** Assertions are used during development to check assumptions and detect bugs early. They are not intended for handling user-facing runtime errors.

Errors → Prevention → Testing → Debugging → Exceptions → Assertions

High-quality software is achieved through systematic design, testing, and error handling.

Lecture 11

Lecture 11-1: Pythonic Idioms

Introduction

Pythonic idioms are common and preferred ways of writing Python code. They emphasize readability, simplicity, and clarity, reflecting Python's design philosophy.

Using Pythonic idioms allows programmers to write cleaner, shorter, and more maintainable code while reducing unnecessary complexity.

Write code that is simple, readable, and expressive.

11.1.1 Truthy and Falsy Values

In Python, every value can be converted to a boolean using the `bool()` function. Values are implicitly interpreted as either `True` or `False`, so explicit comparisons are often unnecessary.

Example:

```python
# Non-Pythonic
if len(my_list) > 0:
    print("Not empty")

# Pythonic
if my_list:
    print("Not empty")
```

Common falsy values:

- `None, False, 0, "", [], {}`

Empty values are False; non-empty values are True.

11.1.2 Membership Operator

The membership operator `in` allows checking whether a value exists in a collection(or iterable objects).

Example:

```python
# Non-Pythonic
numbers = [1,2,4,5,3]
found = False
for n in numbers:
    if n == 3:
        found = True

# Pythonic
```

```python
if 3 in numbers:
    print("Found")
```

This approach is more concise and readable.

11.1.3 Chained Comparisons

Python allows combining multiple comparisons into a single expression.

Example:

```python
# Non-Pythonic
if x > 0 and x < 10:
    print("In range")

# Pythonic
if 0 < x < 10:
    print("In range")
```

Chained comparisons improve readability and reduce redundancy.

11.1.4 While–Else Control Flow

The `else` block in a loop executes only if the loop completes without interruption (i.e., no `break`).

```python
x = 5
while x > 0:
    print(x)
    x -= 1
else:
    print("Loop finished")
```

The else block runs when the loop ends normally.

11.1.5 List Comprehension

List comprehensions provide a compact way to create lists.

Example:

```python
# Non-Pythonic
squares = []
for x in range(5):
    squares.append(x**2)

# Pythonic
squares = [x**2 for x in range(5)]
```

They combine looping and expression into a single readable statement.

11.1.6 zip() Function

The `zip()` function combines multiple iterables into pairs.

```
names = ["Alice", "Bob"]
scores = [85, 90]

for name, score in zip(names, scores):
    print(name, score)
```

Iterate over multiple sequences simultaneously.

11.1.7 Swapping Variables

Python allows swapping values without a temporary variable.

```
# Non-Pythonic
temp = a
a = b
b = temp

# Pythonic
a, b = b, a
```

This uses tuple unpacking internally.

11.1.8 Unpacking Values

Unpacking extracts elements from a collection into variables.

```
point = (3, 5)
x, y = point
```

Functions can also return multiple values:

```
def get_user():
    return "Alice", 25

name, age = get_user()
```

Sequences can be unpacked into variables based on position.

11.1.9 Lambda Functions

Lambda functions are small anonymous functions used for simple operations.

```
add = lambda a, b: a + b
print(add(2, 3))
```

They are commonly used with higher-order functions:

```python
numbers = [1, 2, 3]
result = list(map(lambda x: x * 2, numbers))
```

Summary

Pythonic idioms help write code that is:

- Clear and readable
- Concise and expressive
- Efficient and maintainable

In this lecture, we explored several common and widely used Pythonic idioms. These represent only a subset of Pythonic idioms. Python provides many more patterns and techniques that promote clean and elegant code.

Clarity over complexity. Readability over optimization.

Lecture 11-2: Managing Database with Python

11.2.1 From Foundations to Applications

In previous lectures, we focused on core Python concepts. In this lecture, we transition from foundational knowledge to real-world applications.

Python is widely used in:

- Web development
- Data analysis
- Data-driven applications

To support these applications, we need a reliable way to store and manage data. This leads us to databases.

11.2.2 Quick Overview on Database

A database is an organized and structured collection of data that enables efficient storage, management, and retrieval of information. It allows us to handle large volumes of data effectively while ensuring quick access when needed. Databases are widely used by different roles, including end users who interact with data, developers who design and build applications, and database administrators (DBAs) who manage, maintain, and ensure the reliability of the system.

A database is the backbone of modern data-driven systems.

Why Do We Need Databases?: To understand the importance of databases, consider a real-world scenario such as storing student grades. A system may need to manage information including student ID, name, course, assignment or exam, and corresponding scores. As the number of students grows, managing such data manually or using simple files becomes inefficient and error-prone. Databases provide a structured and scalable solution for organizing, storing, and retrieving this information efficiently.

Databases enable efficient management of complex and growing data.

Database Management Systems (DBMS): Databases are managed using a Database Management System (DBMS), which provides tools and functionality for storing, organizing, and accessing data. Popular DBMS platforms include MySQL, PostgreSQL, and SQLite. Users interact with a DBMS using Structured Query Language (SQL), which enables them to create, read, update, and delete data. These four fundamental operations are collectively known as CRUD operations.

SQL Basics: SQL (Structured Query Language) is the standard language used to communicate with databases. It allows users to define database structures and manipulate data through commands such as creating databases and tables, inserting records, and querying information. For example:

```
CREATE DATABASE database_name;
```

In practice, interaction with a database follows a simple flow: users write SQL commands, these commands are processed by the DBMS, and the database stores or retrieves the requested data. Once the logic of SQL is understood, it can be applied across different database systems such as MySQL, PostgreSQL, and SQLite, as the core concepts remain largely consistent.

Databases vs Files: Databases provide several advantages over simple files such as CSV or Excel. They are designed to handle large and complex datasets efficiently, ensure data consistency, support multiple users working at the same time, and allow powerful querying and data management. In contrast, files are more suitable for small datasets and are convenient for quick manual editing and sharing, but they lack scalability and advanced data management capabilities.

11.2.3 How Python work with Database?

Python can interact with databases using modules (connectors) such as `sqlite3`, MySQL connectors, and PostgreSQL libraries (e.g., `psycopg2`). These modules are not database management systems themselves; instead, they provide an interface that allows Python programs to connect to and communicate with a DBMS.

In this class, we use **SQLite**, which is a lightweight database management system (DBMS), because it is especially simple and practical for learning database concepts. It requires no installation or separate server setup, as it stores data directly in a local file on the computer. This makes it lightweight, easy to use, and ideal for beginners who want to focus on understanding how databases work without dealing with complex configuration.

Python interacts with SQLite according to the following architecture:

Python Program → `sqlite3` module → SQLite Engine → Database File

Components:

- **Python program**: sends SQL commands and processes results
- **`sqlite3` module**: acts as an interface between Python and SQLite
- **SQLite engine**: executes SQL queries and manages data
- **Database file**: stores the actual data on disk

Working with SQLite in Python

To work with a database using `sqlite3` in Python, we follow a structured sequence of steps. These steps are carried out through two main objects: the **connection** object and the **cursor** object.

First, we establish a connection to the database:

```
conn = sqlite3.connect("example.db")
```

The **connection object** (`conn`) represents the link between the Python program and the database file. If the database file does not exist, SQLite will create it automatically.

Next, we create a cursor:

`cursor = conn.cursor()`

The **cursor object** (`cursor`) is used to execute SQL commands and interact with the database.

We can then execute SQL statements:

`cursor.execute("CREATE TABLE students(name TEXT)")`

The `execute()` method sends SQL commands to the SQLite engine for processing.

After making changes to the database, we commit them:

`conn.commit()`

The `commit()` method ensures that all changes are saved permanently in the database file.

Finally, we close the connection:

`conn.close()`

Closing the connection releases resources and ensures that the database is properly updated.

Connection manages the database, while the cursor performs operations on it.

Summary of Objects

- **Connection object (`conn`)**: manages the database connection and controls transactions
- **Cursor object (`cursor`)**: executes SQL queries and retrieves results

These objects provide essential methods such as `execute()`, `commit()`, and `close()`, forming the core workflow for database interaction in Python.

These steps form the basic workflow for database interaction.

These steps will be carried out during the practical session, where you will see how a Python program can interact with persistent data in a real-world setting in depth.

11.2.4 Summary

In this lecture, we introduced databases and how Python interacts with them.

- Databases store and manage structured data
- DBMS tools organize and process data using SQL
- CRUD operations define basic data manipulation
- Python connects to databases using modules like `sqlite3`
- Database operations follow a structured workflow (connect, execute, commit, close)

Data → Storage → Query → Application

Python enables programs to move from computation to data-driven systems.

Lecture 12: API development in Python

Introduction to APIs

An API(Application Programming Interface) is a set of rules and protocols that enables different software components (function, application, system) to communicate with each other. APIs allow applications to exchange data and functionality without exposing their internal implementation details.

Why APIs Are Important

- Enable secure communication and seamless integration between applications. APIs provide a standardized way to exchange data and services while controlling access to resources, improving both interoperability and security. This standardized interaction model naturally supports client-server architectures.
- Promote code reusability by allowing developers to use existing functionalities and services without rewriting the same code. This reduces development time, minimizes duplication, and improves maintainability.

Real-World Example: Consider an online shopping application:

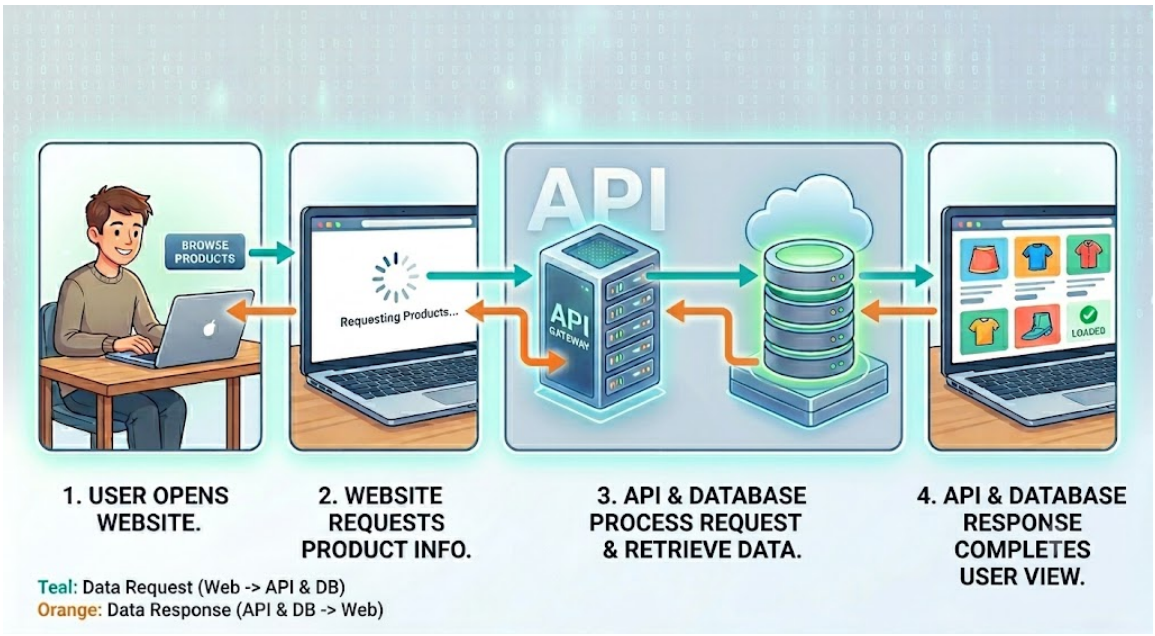

Figure 1: End-to-End Client-Server Request-Response Lifecycle

The API acts as an intermediary between the client and the database.

API Development in Python

There are several frameworks available for API development in Python, such as FastAPI, Flask, and Django REST Framework. In this course, we will use Flask because it is a lightweight and widely adopted web framework for building web applications and RESTful APIs.

Flask is easy to learn, requires minimal setup, and provides a flexible architecture that allows developers to choose the components they need. It also has a large ecosystem of extensions and is well suited for small- to medium-sized projects, making it an excellent framework for learning API development concepts.

The fundamental principles of API development are similar across most Python frameworks. Therefore, once you learn how to develop APIs using Flask, transitioning to other frameworks such as FastAPI or Django becomes much easier, as the main differences are typically in syntax, configuration, and framework-specific features.

1. Simple Hello World Example on API Development

Main Steps and Configurations

Step 1: Importing the Flask Framework

Before building a web application, the fundamental components must be imported from the core package. The `Flask` class provides the foundation required to instantiate a web server, declare routing paths, and process incoming HTTP payloads.

```
from flask import Flask
```

Step 2: Creating a Flask Application Instance

Once imported, a central application object must be instantiated to manage configuration settings, routing maps, and execution contexts.

```
app = Flask(__name__)
```

The variable `app` serves as the primary gateway for the application framework. The mandatory argument `__name__` is a built-in Python variable reflecting the current module's namespace. Flask relies on this identifier to dynamically map and locate resources—such as HTML templates, static assets, and distinct configuration files—relative to the root directory of the application:

- **Direct Execution:** If the script is run directly (e.g., via terminal), `__name__` evaluates to `"__main__"`.
- **Module Import:** If the script is imported by another file, `__name__` retains the actual filename of the module.

Step 3: Defining Routes and Endpoints

Routing mechanisms govern application behavior by dictating which Python function executes when a client hits a specified URL pattern.

```
@app.route('/')
def home():
    return "Hello World"
```

The decorator `@app.route('/')` registers the root URL path (`/`) directly to the immediately proceeding function, `home()`. When a client accesses the server's root address, Flask interceptively catches the request, executes the target function, and wraps its string evaluation into an outgoing HTTP response wrapper.

- `@app.route('/')` acts as the explicit configuration blueprint for the URL path.
- `home()` serves as the designated **endpoint function** managing the logic payload for that path.
- `return "Hello World"` generates the plain-text body sent back to the requesting client.

For example, when a client browser targets the local development server URL (`http://localhost:5015/`), the application coordinates the execution workflow sequentially as follows:

Client Request (`GET /`) → Flask Routing Layer → `home()` Execution → `"Hello World"` Payload → HTTP Response to Client

Figure 2: Sequential Request Processing Flow for the Root Endpoint.

Core Terminology Glossary

Route The specific URL string pattern matching an inbound web address.

Endpoint The physical Python block or function tasked with processing an active route.

Request The structured data payload and headers transmitted from the client to the server environment.

Response The state codes and data objects returned from the server back to the client environment.

2. Advanced Example: Connecting Databases and Web Clients Using a Flask API

In real-world enterprise architectures, an API rarely acts as an isolated layer; it serves as a secure data proxy sitting safely between clients and persistent storage networks.

By implementing this mid-tier isolation, the database remains strictly closed off from direct public network access. The Flask API coordinates security permissions, input sanitation, transaction operations, and data transformations before transmitting records to or from the client.

Before building the system, we must understand two fundamental concepts:

(a) **Data Lifecycle Operations (CRUD)**
The bedrock of persistent software storage relies on CRUD functionality. In contemporary REST APIs, these fundamental operations map directly to specific HTTP standard verbs to cleanly declare intent.

Table 2: Mapping CRUD Operations to RESTful HTTP Methods

CRUD Operation	HTTP Method	System Functional Description
Create	`POST`	Inserts and provisions entirely new records into the database.
Read	`GET`	Fetches existing data entries without modifying system state.
Update	`PUT / PATCH`	Alters existing database records safely.
Delete	`DELETE`	Permanently expunges records from targeted database tables.

(b) **How Communication is Handled**

Web API communication relies entirely on the stateless **Request-Response Cycle** over HTTP. The architecture operates strictly under a client-initiated pattern:

- **The Request Phase:** A client (such as a web application or mobile interface) issues a structured HTTP request containing a target URL, an explicit HTTP Method (such as `GET` or `POST`), and optional parameters or JSON body data.
- **The Processing Phase:** The Flask web server captures this stream, matches the route parameters to the proper endpoint function, executes internal business or validation logic, and queries downstream data tiers if required.
- **The Response Phase:** The endpoint outputs a response format (typically JSON or HTML strings) along with an explicit HTTP Status Code (e.g., `200 OK` for success, or `404 Not Found`), completing the transmission loop.

To build a production-ready system, we must bridge our persistent storage (SQLite) with network protocols. Standard web clients cannot interface with database engines directly; the Flask API acts as our secure gateway.

Step 1: Importing Necessary Modules

To bridge our local SQLite instance with network protocols, initialize your script with the required Flask components and database driver utilities:

```
import sqlite3
from flask import Flask, jsonify, request

app = Flask(__name__)
```

- `sqlite3`: Provisions the localized relational database storage driver.
- `Flask`: Constructs the primary web service framework environment.
- `request`: Intercepts and parses incoming payload states (headers, JSON attributes).
- `jsonify`: Serializes Python arrays and dictionaries into valid HTTP JSON response streams.

Step 2: Defining operations to interact with the database through the API

(a) **Retrieving All Records (GET Request)**

The following route maps an inbound HTTP `GET` request to execute database operations. To conform to REST safety criteria, database connections are explicitly spun up, executed, and immediately severed inside the thread handler to avoid thread deadlock.

```
@app.route('/employees', methods=["GET"])
def get_all_employees():
    conn = sqlite3.connect('employee.db')
    cursor = conn.cursor()

    cursor.execute("SELECT id, name, department, salary
   FROM employees")
    rows = cursor.fetchall()

    cursor.close()
    conn.close()

    # Structure raw database rows into serialized JSON
   objects
    employee_list = []
    for row in rows:
        employee_list.append({
            "id": row[0],
            "name": row[1],
            "department": row[2],
            "salary": row[3]
        })

    return jsonify(employee_list)
```

(b) **Retrieving a Specific Record Using Dynamic Routing (GET Request)**

Flask supports structural parameter mapping variables within route rules using specific variable converter segments: <converter:variable_name>.

```
@app.route('/employees/<int:emp_id>', methods=["GET"])
def get_employee_by_id(emp_id):
    conn = sqlite3.connect('employee.db')
    cursor = conn.cursor()

    cursor.execute("SELECT id, name, department, salary
   FROM employees WHERE id = ?", (emp_id,))
    row = cursor.fetchone()

```

```
    cursor.close()
    conn.close()

    if row:
        return jsonify({"id": row[0], "name": row[1], "department": row[2], "salary": row[3]})
    return jsonify({"error": "Employee record not found"}), 404
```

The value injected directly into the target URL string pattern (e.g., `/employees/1`) evaluates automatically to an integer and passes into the argument payload of `get_employee_by_id()`.

(c) **Creating a New Record (POST Request)**

To capture and write inbound database states safely, target configurations capture client strings from an external `POST` request payload body.

```
@app.route('/employees/new', methods=["POST"])
def create_employee():
    # Extract the inbound request JSON payload body
    new_user = request.json

    if not new_user or "name" not in new_user:
        return jsonify({"error": "Invalid payload format"}), 400

    conn = sqlite3.connect('employee.db')
    cursor = conn.cursor()

    cursor.execute(
        '''
        INSERT INTO employees (name, department, salary)
        VALUES (?, ?, ?)
        ''',
        (new_user["name"], new_user["department"], new_user["salary"])
    )

    conn.commit()
    cursor.close()
    conn.close()

    # 201 is the standard HTTP status code for successful creation
    return jsonify({"message": "Employee record created successfully"}), 201
```

(d) **Updating an Existing Record (PUT Request)**

To safely update persistent records, we map a PUT request to target a specific resource by ID. This route extracts the incoming JSON packet and executes an update query on the database.

```
@app.route('/employees/<int:emp_id>', methods=["PUT"])
def update_employee(emp_id):
    update_data = request.json
    if not update_data:
        return jsonify({"error": "Invalid payload"}), 400

    conn = sqlite3.connect('employee.db')
    cursor = conn.cursor()

    # Dynamically update the targeted record details
    cursor.execute(
        '''
        UPDATE employees
        SET name = ?, salary = ?
        WHERE id = ?
        ''',
        (update_data.get("name"), update_data.get("salary
    "), emp_id)
    )

    conn.commit()
    cursor.close()
    conn.close()

    return jsonify({"message": "Employee record updated
    successfully"}), 200
```

(e) **Destroying an Existing Record (DELETE Request)**

To permanently drop a transaction state, we expose a route using the DELETE verb which securely executes a record deletion query on a verified target integer.

```
@app.route('/employees/<int:emp_id>', methods=["DELETE"])
def delete_employee(emp_id):
    conn = sqlite3.connect('employee.db')
    cursor = conn.cursor()

    cursor.execute("DELETE FROM employees WHERE id = ?",
    (emp_id,))

```

```
    conn.commit()
    cursor.close()
    conn.close()

    return jsonify({"message": "Employee record deleted
   successfully"}), 200
```

Note: Structured JSON Body Payload Requirement: Inbound POST requests targeted to the route above must explicitly submit data confirming to standard structured object notation:

```
{
    "name": "Alice",
    "department": "Sales",
    "salary": 300
}
```

Step 3: Running and Testing the Unified API Architecture

To instantiate and bind the application lifecycle daemon safely across host interfaces, append the execution guard block to your main script entrypoint:

```
if __name__ == '__main__':
    app.run(debug=True, port=5015)
```

Once running, the endpoint services are actively bound to your local daemon loop environment at: `http://localhost:5015/employees`

API Integration Verification Using cURL

You can test database mutations and request operations directly from an active shell interface using the standard command-line utility `cURL`:

(a) Create Employee Record (POST)

```
curl -X POST http://127.0.0.1:5015/employees/new \
     -H "Content-Type: application/json" \
     -d '{"name":"Alice","department":"Sales","salary":300}'
```

(b) Retrieve Global Employee Registry (GET)

```
curl -X GET http://127.0.0.1:5015/employees
```

(c) Update Targeted Record (PUT)

```
curl -X PUT http://127.0.0.1:5015/employees/1 \
     -H "Content-Type: application/json" \
     -d '{"name":"Alice Smith","salary":310}'
```

(d) Destroy Target Record (DELETE)

```
curl -X DELETE http://127.0.0.1:5015/employees/1
```

How Data Flows Under the Hood

To understand how our code functions in practice, we can visualize how data flows through our three-tier system:

$$\textbf{[Client UI / cURL Shell]} \overset{\text{HTTP Request / JSON Response}}{\rightleftharpoons} \textbf{[Flask API Routing Engine]}$$

$$\textbf{[Flask API Routing Engine]} \overset{\text{Parameterized SQL Query}}{\underset{\text{Data Records Mapping}}{\rightleftharpoons}} \textbf{[SQLite Relational Engine]}$$

Figure 3: Three-Tier Relational Application Architectural Lifecycle.

The structured sequence of an inbound call traces through the system across six definitive steps:

(a) **Issuance:** The client triggers a network command (e.g., an HTTP `GET` request).

(b) **Ingress Interception:** The Flask server captures the transaction and routes it to the matching endpoint decorator function.

(c) **Database Connection:** The targeted Python function safely opens a connection handle to our database file using the SQLite driver.

(d) **Query Processing:** The SQL script runs directly inside the database engine on disk to retrieve or mutate records.

(e) **Transformation:** Python converts the raw SQL query tuples back into a native dictionary and serializes it using `jsonify()`.

(f) **Egress Response:** The finalized JSON payload, packaged with the correct HTTP status code, is sent back to complete the transaction cycle.

Summary

Throughout this unit, we have dissected and built the core tenets of database-driven API development:

- **API Framework Fundamentals:** Initializing a lightweight web server and managing basic routing paths using Flask.
- **Dynamic Routing:** Capturing and using URL path parameters (like `<int:emp_id>`) to query specific resources.
- **RESTful CRUD Mapping:** Aligning data lifecycle operations with standard HTTP methods (`GET`, `POST`, `PUT`, and `DELETE`).
- **Data Tier Integration:** Establishing secure database connections and executing SQL queries inside active Flask endpoint functions.
- **API Verification:** Validating system endpoints and testing request payloads from the command line using `cURL`.

Lecture 13: Using Python in Practical Domains

Now we will implement data exploration and machine learning using core Python libraries at a beginner level, as there are separate, dedicated classes for data analysis and machine learning later in your curriculum. Our goal here is simply to give you a gentle first taste of these domains at an introductory level, highlighting how Python serves as their underlying structural foundation.

Python in Data Science

What is Data Analysis? Data analysis is the systematic process of examining raw data to uncover useful patterns, trends, and insights. Its ultimate goal is to convert unstructured data into actionable intelligence:

$$\text{Raw Data} \rightarrow \text{Analysis} \rightarrow \text{Insights} \tag{1}$$

Every data project generally moves through these major milestones:

(a) **Defining the Objective**: Identify the problem or question you need to solve.

(b) **Data Collection**: Gather relevant data from various sources.

(c) **Data Cleaning & Preparation**: Deal with missing values, format inconsistencies, and errors.

(d) **Data Exploration**: Inspect the core characteristics, shape, and structure of your dataset.

(e) **Exploratory Data Analysis (EDA)**: Use descriptive statistics and visual plots to isolate trends.

(f) **Advanced Analysis**: Implement predictive modeling or machine learning algorithms if necessary.

The Core Data Science Stack

Python's power in this field comes from a highly integrated ecosystem of third-party libraries:

- **NumPy**: The foundation for high-performance array manipulation and vector operations.
- **Pandas**: Built for structured tabular data manipulation and analysis.
- **Matplotlib & Seaborn**: Libraries used to generate static data visualizations and charts.
- **Scikit-Learn**: The foundational framework for classical Machine Learning.

Deep Dive into Pandas

Pandas introduces two primary data structures that make data manipulation fast and intuitive:

(a) **Series**: A one-dimensional, labeled array.

(b) **DataFrame**: A two-dimensional, labeled table (resembling an Excel spreadsheet or a SQL database table).

These data structures can be easily initialized from in-memory Python objects (like lists or dictionaries), or loaded directly from external files (such as CSV, Excel, or SQL databases).

1. Creating a DataFrame from a Dictionary

The following example shows how we create a DataFrame from a standard Python dictionary:

```python
import pandas as pd

# Creating a DataFrame from a standard Python dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [21, 20, 22]
}

df = pd.DataFrame(data)
print(df)
```

2. Data Manipulation: Pandas DataFrame vs. Native Python Dictionaries

Why do we use Pandas instead of writing native Python loops over standard dictionaries? Pandas is optimized for speed and provides simple, expressive syntax to perform complex operations. Let us compare common tasks:

Example A: Calculating the Mean Age

- **Using a Native Dictionary:**

```python
total_age = 0
for age in data['Age']:
    total_age += age
mean_age = total_age / len(data['Age'])
print(mean_age) # Output: 21.0
```

- **Using a Pandas DataFrame:**

```python
print(df['Age'].mean()) # Output: 21.0
```

Example B: Transforming Text to Uppercase

- **Using a Native Dictionary:**

```python
# List comprehension to reassign values in the dictionary
data['Name'] = [name.upper() for name in data['Name']]
```

- **Using a Pandas DataFrame:**

```
df['Name'] = df['Name'].str.upper()
```

Example C: Filtering Data (Selecting Rows Where Age ≥ 21)

- **Using a Native Dictionary:**

```
for age in data['Age']:
    if age >= 21:
        print(age) # Outputs: 21, 22
```

- **Using a Pandas DataFrame:**

```
print(df[df['Age'] >= 21])
# Outputs a clean, filtered sub-table!
```

3. Computing Descriptive Statistics

Pandas makes summarizing numerical distributions effortless with its built-in functions:

```
import pandas as pd

lst = [10, 2, 3, 4, 5, 7, 9, 1, 1]
df_lst = pd.DataFrame(lst)

# Easily run statistical functions
print(df_lst.mean())     # Mean
print(df_lst.median())   # Median
print(df_lst.std())      # Standard Deviation
print(df_lst.describe()) # Prints complete summary (count,
   mean, std, min, quartiles, max)
```

Pandas is one of the core libraries for data cleaning, manipulation, transformation, and exploration. By comparing it directly to native data structures above, we can clearly observe its flexibility, expressive power, and syntactic simplicity.

Once data has been cleaned and structured using Pandas, the next natural step in Exploratory Data Analysis (EDA) is visualization. Python's visualization ecosystem is dominated by two highly complementary libraries:

- **Matplotlib**: A low-level, highly customizable plotting library. It provides total control over every graphical element (axes, labels, colors, and line styles) but can require verbose boilerplate code.
- **Seaborn**: A high-level visualization library built on top of Matplotlib. It integrates natively with Pandas DataFrames and provides beautiful, statistically-oriented default styles and complex plots with minimal code.

We will explore these visualization techniques in detail during our practical lab sessions.

Python in Machine Learning

Once you have prepared, cleaned, and explored your data, you can use it to build predictive patterns. This is where **Scikit-Learn (`sklearn`)** comes in.

What is Scikit-Learn?

Scikit-Learn is an open-source Python library designed specifically for machine learning. It is seamlessly integrated with the rest of your data science stack, building on top of **NumPy** (for numerical operations) and **Matplotlib** (for visualizations) while playing beautifully with **Pandas DataFrames**.

The Two Pillars of Machine Learning

Machine Learning algorithms generally fall into two categories, depending on how the data is structured:

1. Supervised Learning (Guided Pattern Discovery)

Supervised learning operates on **labeled data**, meaning every training sample contains both the input features and the correct target label. Think of it like learning with an answer key.

- **Regression**: Used for predicting continuous, numerical quantities.
  - *Example*: Predicting a house's market price based on its size.
- **Classification**: Used for predicting discrete, categorical labels.
  - *Example*: Classifying whether an email is "Spam" (1) or "Normal" (0).

2. Unsupervised Learning (Exploration & Discovery)

Unsupervised learning operates on **unlabeled data**. The algorithm's goal is to find hidden structures, natural clusters, or patterns on its own.

- **Clustering**: Grouping items together based on mathematical similarities.
- **Dimensionality Reduction**: Shrinking the number of input features while preserving vital information.
- **Anomaly Detection**: Spotting unusual outliers that deviate significantly from standard groupings.

How to Select a Machine Learning Model

When starting a new project, how do you decide which algorithm to choose? You can follow these criteria:

(a) **Data Scale**: Scikit-Learn generally requires at least 50 samples to start training. If you have high-dimensional features where the number of features D is larger than

your sample size N ($D > N$), prioritize regularized models like Ridge or L1/L2 lasso methods.

(b) **Target Variable Type**:

- Discrete Categories → **Classification**.
- Continuous Quantities → **Regression**.
- No Target Labels → **Clustering / Dimensionality Reduction**.

(c) **Performance vs. Explainability**:

- If you need to easily explain *why* a decision was made, choose simpler linear models or Decision Trees.
- If raw predictive accuracy is your only priority, use complex ensemble models like Random Forests or Gradient Boosting.

The Strategic Estimator Heuristic

A pro-tip for machine learning workflows is to **start simple**.

- Establish a quick baseline model using a simple estimator (like a linear support vector classifier or Ridge regression) and a dummy benchmark.
- Inspect your dataset size:
  - If you have $< 100k$ samples, try kernel approximations or ensemble models.
  - If you have $\geq 100k$ samples, pivot to out-of-core solvers like SGD Classifiers.
- Increase complexity (like switching to non-linear kernels or boosting architectures) only if your simple baseline's accuracy is underperforming.

The End-to-End Supervised ML Pipeline

A clean machine learning workflow follows a highly structured, standard lifecycle:

Phase 1: Initialization & Training

(a) **Data Preparation (Splitting)**: Divide your main dataset into a **Training Set** (used for pattern extraction) and an independent **Testing Set** (reserved for final validation).

(b) **Model Selection**: Select the algorithm that best matches your data limitations and objective.

(c) **Training the Model (Fitting)**: Feed the training data to your model using the `.fit()` method so the algorithm can adjust its internal weights to minimize error.

Phase 2: Evaluation & Deployment

4 **Model Prediction & Evaluation**: Test the model's performance on unseen data using `.predict()` and evaluate performance metrics (such as Accuracy or Mean Squared Error).

5 **Iterative Optimization**: Boost your model's performance by fine-tuning hyperparameters and experimenting with feature adjustments.

6 **Model Persistence**: Save your trained model to a file using serialization libraries so it can be deployed into production systems later without retraining.

First Taste: Code Example of a Classifier

To show you how simple it is to program this workflow in Python, here is a complete supervised machine learning pipeline using a dummy dataset:

```python
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
import numpy as np

# 1. Create a dummy dataset (100 samples: features X, and binary labels y)
X = np.random.randn(100, 2)  # Two continuous features
y = np.where(X[:, 0] + X[:, 1] > 0, 1, 0)  # Simple classification rule

# 2. Train-Test Split (Phase 1)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 3. Model Selection & Fitting (Phase 1)
model = LogisticRegression()
model.fit(X_train, y_train)  # Training step!

# 4. Predict & Evaluate (Phase 2)
predictions = model.predict(X_test)
accuracy = model.score(X_test, y_test)

print(f"Predictions: {predictions}")
print(f"Testing Accuracy: {accuracy * 100:.1f}%")
```

Summary

This lecture introduced the foundational role Python plays in modern data science and machine learning. Rather than serving as a replacement for deep statistical theory, Python acts as the unified glue code that ties together data ingestion, manipulation, visual exploration, and predictive modeling into a single seamless pipeline.

Key takeaways from this section include:

- **Data Stack Synergy**: Standard Python structures (lists and dictionaries) are convenient, but third-party libraries like `Pandas` and `NumPy` provide vectorized operations and specialized data structures (`Series` and `DataFrames`) necessary for performant, expressive data manipulation.
- **Exploratory Visual Workflow**: Cleaning raw data and visualizing trends using `Matplotlib` and `Seaborn` form the necessary prerequisite stage (EDA) before feeding data into predictive algorithms.
- **Structured ML Lifecycle**: Frameworks like `Scikit-Learn` standardize machine learning workflows into a unified API pattern—initializing models, fitting on training data (`.fit()`), evaluating on unseen test data (`.predict()`), and optimizing baseline estimators.

As you progress through your curriculum, you will build upon this foundation by exploring the deep mathematical principles—such as linear algebra, calculus, and probability theory—that govern these algorithms. For now, understanding which tool to use and when provides a practical framework to begin tackling real-world data problems.

Lecture 14: Python Recap & Summary

In the study of software engineering, a programming language is far more than a mere vehicle for syntax; it represents a conceptual framework through which computer scientists reason about and solve various problems. This chapter explores the multi-paradigm nature of the Python programming language, establishing how it serves as an adaptable bridge across divergent coding methodologies. By understanding these paradigms, software developers can move beyond syntax-level fluency and develop the critical skills required for language selection, allowing them to choose the optimal tool for any given domain or architectural goal. By understanding these paradigms, software developers can move beyond syntax-level fluency and develop the critical skills required for language selection, allowing them to choose the optimal tool for any given domain or architectural goal. The history of computer science has produced several distinct methodologies for designing and executing program instructions. To analyze these designs comprehensively, we contrast the paradigms of some programming languages. Each of these languages embodies a unique approach to computational logic, state management, and abstraction. We categorize these approaches into four primary paradigms: imperative, declarative, functional, and object-oriented.

The Imperative Style: How to Solve

The imperative paradigm represents the most traditional and intuitive approach to programming. At its core, the imperative style focuses heavily on the explicit steps required to accomplish a task. A developer writing imperative code describes the exact step-by-step control flow of a program, manually mutating states and defining explicit iteration counters and loops. This paradigm is highly direct and remains popular because its instructions map directly to physical CPU operations and hardware memory cells.

(a) **Low-Level Imperative Execution in C**

The C programming language is a classic example of a purely imperative model. To perform a simple mathematical operation, such as squaring a list of numbers, the C compiler requires the programmer to manage memory bounds, array indices, and loop termination conditions manually. Consider the following implementation:

```c
#include <stdio.h>
int main() {
    int nums[] = {1, 2, 3, 4}; // Explicit array initialization
    int squared[4]; // Manual allocation for output

    // Explicit loop structure with an index counter
    for(int i = 0; i < 4; i++) {
        squared[i] = nums[i] * nums[i]; // assigning/updating state
        printf("%d ", squared[i]); // Side effect
```

```
    }
    return 0;
}
```

In this code, the CPU is directed through each iteration of the array. The index counter i must be explicitly declared, incremented, and validated against the array boundary ($i < 4$). There is no abstraction layer separating the mathematical goal from the underlying execution sequence.

(b) **Imperative Flow Control in Python**
While Python is renowned for high-level abstractions, it fully supports the imperative paradigm. Developers can easily write procedural instructions that mirror the mechanical state transitions found in C. Here is the equivalent imperative implementation in Python:

```
numbers = [1, 2, 3, 4]
squared = []

# Manually looping and managing the list state
for num in numbers:
    squared.append(num ** 2)

print(squared) # Output: [1, 4, 9, 16]
```

In this snippet, although Python abstracts away the explicit index variable i, the logic remains strictly imperative. The programmer still declares an empty output array, manually iterates over the sequence, and appends mutated elements to the list step-by-step.

The Declarative Style: What to Solve

In contrast to the imperative style, the declarative paradigm shifts the developer's focus from *how* to solve a problem to *what* the desired outcome should be. In a declarative framework, the precise mechanisms of state transitions, loop bounds, and element accumulation are completely abstracted away and managed automatically by the language's underlying execution engine. For example, this cuts down on extra code and prevents the common counting mistakes that can occur when managing loops manually, as seen in the previous imperative examples.

Let's explore some declarative coding examples in Python.

(a) List Comprehensions
Python implements declarative principles elegantly through its list comprehension syntax. By collapsing loops and conditional accumulation into a single expressive statement, the code specifies the mathematical mapping of input to output directly:

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

```

```
# Expressing "what" the list contains, rather than "how"
    to build it
squared = [num ** 2 for num in numbers]

print(squared) # Output: [1, 4, 9, 16]
```

Notice that this declarative implementation contains no manual appends, state definitions, or variable assignments during the loop. The engine interprets the comprehension and constructs the resulting collection implicitly, allowing the developer to express the transformation as a direct mathematical projection.

(b) Set and Dictionary Comprehensions

Python extends declarative collection building to sets and dictionaries. Instead of initializing an empty container and manually calling `add()` or updating keys inside an iterative loop, the developer declares the mapping and filtering rules directly within the curly braces:

```
# Declarative Set: Automatically ensures unique, squared
    values
numbers = [1, -1, 2, -2]
unique_squares = {num ** 2 for num in numbers}
print(unique_squares) % Output: {1, 4}

% Declarative Dictionary: Directly mapping numbers to
    their cubes
cubes_map = {num: num ** 3 for num in range(1, 4)}
print(cubes_map) % Output: {1: 1, 2: 8, 3: 27}
```

In both cases, the execution engine handles container allocation, hashing collision resolution, and element insertion implicitly without explicit procedural steps.

(c) Declarative Filtering (Conditional Comprehensions)

Another highly popular and practical application of the declarative style in Python is filtering collections based on conditional criteria. In an imperative model, selecting specific elements from a dataset (such as extracting only even numbers) requires setting up a loop, executing a conditional branch, and manually appending matching items to a pre-allocated collection. Python abstracts this process into a single declarative expression where you define exactly *what* criteria elements must meet to be included:

```
numbers = [1, 2, 3, 4, 5, 6]

% Imperative style (for comparison):
even_numbers = []
for num in numbers:
    if num % 2 == 0:
        even_numbers.append(num)

```

```
% Declarative style:
% We declare that "even_numbers" contains "num" for each
    "num" in "numbers" if it is even
even_numbers = [num for num in numbers if num % 2 == 0]

print(even_numbers) % Output: [2, 4, 6]
```

By expressing the filtering condition directly within the collection builder, the underlying execution engine handles loop state, sequence index tracking, and memory resizing automatically. This declarative syntax eliminates the boilerplate of procedural conditional branching, preventing the common counting or index-tracking mistakes that can occur when managing loops manually in imperative layouts.

(d) Declarative Logical Reductions

In the imperative style, checking if a collection meets a specific condition requires setting up state flags and manually managing control flow with breaks. Python abstracts this logic using declarative reductions like `any()` and `all()`, which evaluate a collection against a predicate in a single logical assertion:

```
scores = [75, 82, 91, 43, 88]

% Imperative style (for comparison):
has_failing_score = False
for score in scores:
    if score < 50:
        has_failing_score = True
        break

% Declarative style:
% We state "what" we are checking (if any score is less
    than 50)
has_failing_score = any(score < 50 for score in scores)

print(has_failing_score) % Output: True
```

This declarative approach reads almost exactly like a mathematical predicate formula:

$$\exists x \in \text{scores} : x < 50$$

The engine optimizes the underlying search execution and stops evaluation (short-circuiting) as soon as the condition is satisfied, without requiring the developer to write explicit control-flow branches.

Contrasting Python Declarative Style with SQL

To truly appreciate the nature of declarative programming, it is valuable to contrast Python's syntax with a purely declarative, domain-specific language like SQL (Structured Query Language). In SQL, there are no concepts of loops, array indices, or procedural state manipulation. The developer simply describes the criteria for the final dataset, and the database execution engine determines the optimal path to retrieve and transform the data.

Consider the task of taking a table of numbers, filtering for even values, and squaring them.

```sql
-- Purely Declarative Query in SQL
SELECT value * value AS squared
FROM numbers_table
WHERE value % 2 = 0;
```

We can compare this directly to Python's declarative conditional list comprehension:

```python
# Python Declarative Equivalent
squared = [num ** 2 for num in numbers if num % 2 == 0]
```

Key Paradigm Comparisons

- **Underlying Execution Engines:** In SQL, the query compiler analyzes the `SELECT`, `FROM`, and `WHERE` clauses to build an execution plan (often involving index scans or hash joins) without any step-by-step instructions from the programmer. Similarly, in Python, the list comprehension is compiled into optimized bytecode (`LIST_APPEND` instructions at the C level) that executes much faster than a manual procedural loop.
- **Mathematical Projections:** Both languages treat data transformation as a mathematical projection. The SQL expression `value * value` maps directly to Python's expression `num ** 2`, acting as a transformation function $f(x) = x^2$ applied to a filtered subset.
- **Readability and Intent:** In both examples, the code reads as a description of the final result set rather than a recipe on how to construct it. This reduces cognitive load and eliminates the potential for manual loop-management bugs (such as off-by-one errors or incorrect state tracking) across both languages.

The Functional Style: Abstracting Logic and Preserving Purity

The functional paradigm is rooted in mathematical lambda calculus. It views computation as the evaluation of pure, mathematical functions while strictly avoiding mutable data and side effects. In functional programming, functions are treated as first-class citizens. This means they can be dynamically assigned to variables, passed into other functions as arguments, and returned as output from computations.

Functional Programming in Python

Python incorporates key functional elements through higher-order functions like `map()` and anonymous expressions known as `lambda` functions. The following code demonstrates how

to calculate squares using functional principles:

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

# Using a higher-order function that takes an anonymous
    lambda expression
squared = list(map(lambda num: num**2, numbers))

print(squared) # Output: [1, 4, 9, 16]
```

In this model, the transformation is abstracted into the anonymous function `lambda num: num**2`. The higher-order function `map()` applies this lambda across the entire collection. The operation leaves the original `numbers` array completely unmodified, illustrating the functional design principle of immutability.

Pure Functional Implementation: Haskell

While Python allows functional programming style as an option, a purely functional language like Haskell strictly enforces these principles by default. In Haskell, all variables are immutable by design, and functions are mathematically ”pure”—meaning they cannot produce side effects, modify global state, or alter their inputs.

To achieve the same goal of squaring a list of numbers, Haskell uses an anonymous function syntax (where the backslash \ represents the Greek letter λ) mapped over a list:

```
-- Define a list of integers
numbers :: [Int]
numbers = [1, 2, 3, 4]

--- Map an anonymous lambda function over the list
squared :: [Int]
squared = map (\num -> num ^ 2) numbers

--- Output when evaluated: [1, 4, 9, 16]
```

This example illustrates several key structural differences from Python’s hybrid model:

- **Enforced Immutability:** In Python, you *could* accidentally mutate a state inside a function. In Haskell, state mutation is mathematically impossible; the compiler will reject any code that attempts to change the values within `numbers`.
- **Mathematical Syntax:** The backslash \ is a direct syntactic nod to λ calculus. The expression `\num -> num ^ 2` is a pure mathematical abstraction that transforms input to output without any execution boilerplate.
- **Lack of Side Effects:** Since Haskell segregates pure computation from input/output actions (using monads), the evaluation of `squared` is guaranteed to be computationally clean, making the execution predictable, easy to parallelize, and simple to verify mathematically.

Object-Oriented Style: Modelling the Real World

While basic programming focuses on processing individual pieces of data like numbers or text, real-world problems require a higher level of abstraction. The Object-Oriented Programming (OOP) paradigm solves this by organizing software design around 'objects' that represent real-world entities. Instead of keeping data and functions separate, objects bundle them together—encapsulating both data (attributes) and behavior (methods) into cohesive, reusable structures. This design enforces modularity and a clean separation of concerns, helping developers easily manage complexity as codebases grow larger.

The Rigid OOP Architecture of Java

Java is built upon a strict object-oriented design. In Java, everything must reside inside a class structure, and even the entry point of the program demands class instantiation or static encapsulation. This design pattern enforces clean boundaries but introduces significant boilerplate code:

```java
public class Squarer {
    private int number; // Private state

    public Squarer(int number) {
        this.number = number; // Enforced encapsulation
    }

    public int getSquare() {
        return this.number * this.number; // Bound behavior
    }

    public static void main(String[] args) {
        Squarer s = new Squarer(4); // Strict object
   instantiation
        System.out.println(s.getSquare()); // Invocation
    }
}
```

The Flexible OOP Model of Python

Python offers a highly versatile and dynamic object-oriented model. It features explicit self-references, dynamic type determination, and minimizes boilerplate code. This allows developers to write clean object-oriented architectures without being forced into rigid structures when they are unnecessary:

```python
class Squarer:
    def __init__(self, number):
```

```
        self.number = number # Dynamically allocated attribute

    def get_square(self):
        return self.number ** 2

# Simplified instantiation and invocation
s = Squarer(4)
print(s.get_square()) # Output: 16
```

Summary

Real-life analogy. Imagine you want a clean room.

- You can give step-by-step physical instructions (**Imperative**).
- You can point to a picture of a clean room and say "Make it look like this" (**Declarative**).
- You can delegate specific, isolated tasks to different helpers (**Functional**).
- You can define what a "Room" object is and call its `.clean()` method (**Object-Oriented**).

Python is a multi-paradigm language, meaning it lets you choose the most flexible tool for the job.

The Multi-Paradigm Trade-off. In contrast to languages designed around a single, strict paradigm, Python's multi-paradigm approach offers ultimate adaptability. It gives you the freedom to blend styles—such as declarative data processing and object-oriented domain modeling—to write clean, efficient code. However, this freedom requires discipline; without it, you risk creating inconsistent, "hybrid" codebases where different programming styles clash, making large systems complex to maintain.

Readability and Pragmatism. Ultimately, Python's greatest asset is its deliberate refusal to force you into a single cognitive mold. By prioritizing readability and simplicity, Python minimizes developer overhead, valuing rapid prototyping and structural clarity over low-level, micro-optimized execution. This pragmatism has made Python the foundational tool of choice across modern computing domains, including Web Development, Data Science, and Machine Learning.

Languages as Tools. As you move forward, remember that there is no universally "best" programming language. The optimal choice depends entirely on your project's requirements, performance budgets, and long-term goals. As modern software engineers, we must look beyond syntax and deeply understand the paradigms that shape computational logic. A programming language is simply a tool used to bring abstract ideas to life; your true value lies in your ability to critically evaluate and apply the correct paradigm to solve real-world problems.