Python for Beginners

readbytes

Python for Beginners

From Novice to Advanced Programmer

readbytes.github.io

2025-07-28

This page is intentionally left blank.

Contents

Chapter 1.

Introduction to Python

1. What is Python and Why Learn It?
2. Installing Python and Setting Up Your Environment
3. Writing and Running Your First Python Program
4. Understanding Python Syntax and Indentation

1 Introduction to Python

1.1 What is Python and Why Learn It?

Python is a high-level, interpreted programming language known for its simplicity, readability, and versatility. Created by Guido van Rossum and first released in 1991, Python was designed with the philosophy of making code easy to write and understand. Its guiding principles—captured in "The Zen of Python"—emphasize clarity, simplicity, and explicitness, making it one of the most accessible programming languages for beginners and professionals alike.

1.1.1 The Origin and Design Philosophy of Python

Python's name is inspired by the British comedy group Monty Python, reflecting its creator's desire for a language that is fun and approachable. Unlike many languages developed primarily for machine efficiency or complex features, Python focuses on human readability and straightforward syntax.

Key design philosophies include:

- **Readability counts**: Python code looks clean and resembles plain English.
- **Simple is better than complex**: Python encourages writing clear and maintainable code.
- **Explicit is better than implicit**: Code behavior should be clear to readers and avoid hidden magic.
- **Batteries included**: Python ships with a large standard library, offering tools and modules for various tasks out of the box.

1.1.2 Key Features That Make Python Popular

- **Easy to Learn and Use**: Python's syntax is concise and intuitive, allowing new programmers to start coding quickly without wrestling with complex rules.
- **Interpreted Language**: Python runs code line-by-line, making debugging easier and speeding up development cycles.
- **Cross-Platform**: Python runs on Windows, macOS, Linux, and many other platforms, ensuring portability.
- **Dynamic Typing**: Python handles variable types automatically, reducing the need for verbose declarations.
- **Extensive Libraries and Frameworks**: Python has rich libraries for web development (Django, Flask), data science (Pandas, NumPy), machine learning (TensorFlow, Scikit-learn), automation (Selenium, PyAutoGUI), and more.

- **Strong Community Support**: An active global community contributes to Python's ecosystem, providing tutorials, open-source projects, and help forums.

1.1.3 Versatility Across Domains

Python's flexibility allows it to shine in many areas:

- **Web Development**: Frameworks like Django and Flask help developers build scalable websites and APIs efficiently.
- **Automation and Scripting**: Python scripts automate repetitive tasks such as file management, data scraping, and testing.
- **Data Science and Machine Learning**: With powerful libraries, Python is the preferred language for analyzing large datasets, creating visualizations, and developing AI models.
- **Education**: Its beginner-friendly nature and interactive environments make Python ideal for teaching programming fundamentals.
- **Game Development, Networking, IoT, and More**: Python's adaptability makes it suitable for diverse applications beyond these core areas.

1.1.4 Comparison with Other Popular Languages

- **Python vs. Java**: Java is statically typed and requires verbose syntax, which can be daunting for beginners. Python's dynamic typing and simpler syntax allow learners to focus on programming concepts without boilerplate code. Java excels in large-scale, performance-critical applications, but Python's rapid prototyping is preferred for many projects.
- **Python vs. C++**: C++ is powerful and close to hardware, but it has a steep learning curve with complex syntax and manual memory management. Python abstracts these details, enabling faster development and reducing bugs, which is especially valuable for beginners.
- **Community and Ecosystem**: While Java and C++ have mature ecosystems, Python's open-source libraries and frameworks grow rapidly, supported by a vibrant community that continually expands its capabilities.

1.1.5 Real-World Applications Built with Python

Many prominent companies rely on Python for critical parts of their technology stacks:

- **Instagram and Spotify** use Python for backend services and data analysis.

- **Dropbox** built its desktop client largely in Python.
- **NASA** employs Python for scientific computing and data visualization.
- **Google** incorporates Python for system building, automation, and AI research.
- **YouTube** uses Python for video processing and website management.

1.2 Installing Python and Setting Up Your Environment

Welcome to the practical part of your Python journey! Before writing any code, you need to install Python and set up your working environment. This section will guide you step-by-step through installing Python on Windows, macOS, and Linux, introducing some popular tools to write Python code, verifying your installation, and writing your very first Python script.

1.2.1 Installing Python on Different Platforms

Windows

1. **Download Python Installer**
   - Go to the official Python website: https://www.python.org/downloads/
   - Click the **Download Python 3.x.x** button (where 3.x.x is the latest version).
2. **Run the Installer**
   - Open the downloaded installer file.
   - **Important:** Make sure to **check the box** that says **Add Python 3.x to PATH** at the bottom of the window.
   - Click **Install Now**.
   - Wait for the installation to complete.
3. **Verify Installation**
   - Open the Command Prompt (press `Win + R`, type `cmd`, and press Enter).
   - Type the following and press Enter:

     ```
     python --version
     ```

   - You should see the Python version printed, for example:

     ```
     Python 3.11.2
     ```

   - If it doesn't work, you may need to add Python to your PATH manually (see the Environment Variables section below).

macOS

1. **Check if Python is Pre-installed**
    - Open the Terminal (found in Applications > Utilities).
    - Type:

      ```
      python3 --version
      ```

    - macOS often comes with Python 2.x installed by default; Python 3 may not be installed.
2. **Install Python 3 via Installer**
    - Go to https://www.python.org/downloads/
    - Download the latest macOS installer `.pkg` file.
    - Run the installer and follow the instructions.
3. **Verify Installation**
    - Open Terminal and type:

      ```
      python3 --version
      ```

    - You should see the Python version displayed.
4. **Alternative: Use Homebrew**
    - If you have Homebrew installed, you can install Python using:

      ```
      brew install python
      ```

    - Then verify with:

      ```
      python3 --version
      ```

Linux (Ubuntu/Debian)

1. **Check if Python is Installed**
    - Open Terminal.
    - Type:

      ```
      python3 --version
      ```

    - If installed, you will see the version number.
2. **Install Python 3**
    - Update your package list:

      ```
      sudo apt update
      ```

    - Install Python 3:

      ```
      sudo apt install python3
      ```

- You may also want to install the package manager for Python (pip):

  ```
  sudo apt install python3-pip
  ```

3. **Verify Installation**
   - Type:

     ```
     python3 --version
     ```

1.2.2 Setting Environment Variables

Most modern installers set the PATH environment variable automatically. However, if you encounter issues running `python` or `python3` from the terminal or command prompt, you may need to add Python to your PATH manually.

Setting Environment Variables on Windows

1. Open **Start Menu**, search for **Environment Variables**, and select **Edit the system environment variables**.
2. In the **System Properties** window, click **Environment Variables**.
3. Under **System variables**, select **Path** and click **Edit**.
4. Click **New** and add the path to your Python installation folder, e.g., `C:\Users\YourUsername\AppDat` and also the `Scripts` folder inside it, e.g., `C:\Users\YourUsername\AppData\Local\Programs\Pytho`
5. Click **OK** on all windows.
6. Restart Command Prompt and verify with:

   ```
   python --version
   ```

1.2.3 Introducing Python Development Tools

Once Python is installed, you need a way to write and run your programs. Here are some popular tools:

IDLE (Integrated Development and Learning Environment)

- Comes pre-installed with Python.
- To open, search for **IDLE** in your OS's Start menu or applications.
- It provides a simple Python shell and editor.
- Great for beginners to quickly write and run scripts.

Visual Studio Code (VS Code)

- A powerful, free code editor.
- Download from https://code.visualstudio.com/.
- Install the **Python extension** from Microsoft inside VS Code for enhanced Python support.
- Offers features like syntax highlighting, debugging, and integrated terminal.

Jupyter Notebook

- Interactive web-based environment popular in data science.
- Install using pip:

  ```
  pip install notebook
  ```

- Run with:

  ```
  jupyter notebook
  ```

- Opens in your browser, allowing you to write and run Python code in small blocks (cells).

1.2.4 Writing Your First Python Script

Let's create a simple program to verify your setup!

1. Open your preferred editor (IDLE, VS Code, or any text editor).
2. Create a new file named `hello.py`.
3. Type the following code:

   ```
   print("Hello, Python world!")
   ```

4. Save the file.
5. Run the script:
   - **Using Command Line:**
     - Open Command Prompt or Terminal.
     - Navigate to the folder where you saved `hello.py`:

       ```
       cd path/to/your/folder
       ```

     - Run the script with:

       ```
       python hello.py
       ```

       or on some systems:

```
python3 hello.py
```

- **Using IDLE:**
  - Open `hello.py` in IDLE.
  - Press `F5` or select **Run > Run Module**.

6. You should see the output:

```
Hello, Python world!
```

1.2.5 Summary

- Download and install Python for your operating system.
- Ensure Python is added to your system PATH.
- Choose an editor or IDE like IDLE, VS Code, or Jupyter Notebook.
- Verify the installation by running `python --version` or `python3 --version`.
- Write and run your first Python program.

Now that Python is installed and your environment is ready, you're set to explore the language and start programming!

1.3 Writing and Running Your First Python Program

Now that you have Python installed and your environment set up, it's time to write and run your very first Python program! This section will walk you through creating a simple `"Hello, world!"` script, running it using both the command line and an IDE, and understanding the basic structure of a Python program. We'll also explore how to modify your script to interact with the user and perform simple calculations.

1.3.1 Creating Your First Python Script: "Hello, world!"

Step 1: Write Your Code in a `.py` File

1. Open your preferred text editor or IDE (such as IDLE, Visual Studio Code, or any simple text editor like Notepad).
2. Create a new file and type the following code exactly:

```
print("Hello, world!")
```

3. Save the file as `hello.py`. The `.py` extension tells your computer this is a Python script.

Step 2: Run the Script

You can run your Python program either via the command line or using an IDE.

Running via Command Line

1. Open your terminal or command prompt.
2. Navigate to the folder where you saved `hello.py`. For example:
   ```
   cd path/to/your/folder
   ```
3. Run the script by typing:
   ```
   python hello.py
   ```
   or on some systems:
   ```
   python3 hello.py
   ```
4. You should see the output:
   ```
   Hello, world!
   ```

Running in IDLE

1. Open IDLE.
2. From the menu, select **File > Open**, then open `hello.py`.
3. Run the program by pressing `F5` or selecting **Run > Run Module**.
4. The output will appear in the IDLE shell window:
   ```
   Hello, world!
   ```

1.3.2 Anatomy of a Basic Python Script

Let's understand what happens in this simple script:

Full runnable code:

```
print("Hello, world!")
```

- `print()` is a built-in Python function that outputs text or other data to the console.
- The text inside the parentheses and quotes — `"Hello, world!"` — is called a **string**.
- When Python runs this script, it executes commands **top to bottom**. Here, it reads the `print()` function and displays the string on your screen.

1.3.3 Modifying the Program

Example 1: Print User Input

Let's make the program interactive by asking for the user's name and greeting them.

Update your script to:

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

Explanation:

- `input()` pauses the program and waits for the user to type something.
- Whatever the user types is stored in the variable `name`.
- The `print()` function combines the string `"Hello, "` with the user's input and an exclamation mark to greet them.

Try running the script now. You will see:

```
Enter your name: Joe
Hello, Joe!
```

Example 2: Perform a Basic Calculation

Let's write a script that asks the user for two numbers and prints their sum.

```
num1 = input("Enter the first number: ")
num2 = input("Enter the second number: ")

# Convert input strings to integers
num1 = int(num1)
num2 = int(num2)

total = num1 + num2
print("The sum is:", total)
```

Explanation:

- The `input()` function returns text, so we convert the input to integers using `int()`.
- We add the two numbers and store the result in `total`.
- Finally, we print the sum.

Example run:

```
Enter the first number: 5
Enter the second number: 7
The sum is: 12
```

1.3.4 Summary

- Python scripts are saved with a `.py` extension.

- You can run scripts from the command line using `python filename.py` or from an IDE like IDLE.
- Python executes code from top to bottom.
- The `print()` function outputs data to the screen.
- You can interact with users using `input()`.
- Python lets you perform calculations after converting input strings to numbers.

Congratulations! You've written and run your first Python programs. In the next sections, you will dive deeper into Python syntax and how to write more complex code.

1.4 Understanding Python Syntax and Indentation

Python is known for its clean and readable syntax, which makes it an excellent language for beginners. One of the unique features of Python compared to many other programming languages is its use of **indentation** (spaces or tabs at the start of a line) to define blocks of code instead of braces `{}` or keywords.

This section will introduce you to the basics of Python syntax, explain why indentation is crucial, and show how it controls the flow of your programs.

1.4.1 Pythons Syntax Rules and Indentation

What Is Indentation?

Indentation is the space at the beginning of a line of code. In Python, indentation is **not optional**—it is part of the syntax. The amount of indentation you use is up to you (usually 4 spaces per level is the convention), but it must be **consistent within a block**.

Why Does Python Use Indentation?

Unlike many languages that use curly braces `{}` or keywords to mark the start and end of blocks of code, Python uses indentation to:

- Improve code readability.
- Enforce a clean and uniform coding style.
- Define where blocks of code begin and end.

For example, the body of an `if` statement, a loop, or a function is defined by indentation.

1.4.2 How Indentation Defines Code Blocks

Correct Indentation Example

Full runnable code:

```
if 5 > 2:
    print("Five is greater than two!")
    print("This line is also part of the if block.")

print("This line is outside the if block.")
```

Output:

```
Five is greater than two!
This line is also part of the if block.
This line is outside the if block.
```

Explanation:

- The two `print` statements indented under the `if` line belong to the `if` block.
- The last `print` is not indented and therefore runs independently.

Incorrect Indentation Example

```
if 5 > 2:
print("Five is greater than two!")
    print("This will cause an error.")
```

This code will raise an **IndentationError**:

```
IndentationError: expected an indented block
```

Because Python expects the lines inside the `if` block to be indented consistently.

1.4.3 Indentation in Different Structures

Indentation in `if` Statements

Full runnable code:

```
age = 18

if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")
```

- Both the `if` and `else` blocks have their own indented code.
- Indentation defines which statements belong to which block.

Indentation in Loops

Full runnable code:

```
for i in range(3):
    print("Iteration:", i)
    print("Inside the loop")

print("Outside the loop")
```

- The two print statements inside the loop are indented.
- The last print is not indented, so it runs after the loop finishes.

Indentation in Functions

Full runnable code:

```
def greet(name):
    print("Hello, " + name + "!")
    print("Welcome to Python.")

greet("Alice")
```

- The two `print` statements form the function body and are indented.
- The call to `greet()` is not indented, so it runs after the function is defined.

1.4.4 Tips for Working with Indentation

- **Use spaces, not tabs:** The Python community recommends using 4 spaces per indentation level. Mixing tabs and spaces can cause errors.
- **Be consistent:** Indent all lines in a block by the same number of spaces.
- **Your editor helps:** Most code editors and IDEs (like VS Code or IDLE) automatically handle indentation for you.
- **IndentationError** means you have inconsistent or missing indentation—check your code carefully.

1.4.5 Summary

- Python uses indentation to define blocks of code instead of braces.
- Consistent indentation is mandatory and controls the program flow.
- Indentation affects `if` statements, loops, functions, and other structures.
- Always use the same number of spaces for each indentation level (4 spaces is standard).
- Incorrect indentation causes errors and stops your program from running.

Mastering indentation is essential to writing clear, error-free Python code. It helps your

programs run correctly and makes your code easier to read and maintain.

Chapter 2.

Basic Python Syntax and Data Types

1. Variables, Constants, and Basic Data Types (`int`, `float`, `str`, `bool`)
2. Operators and Expressions
3. Comments and Naming Conventions
4. Simple Input and Output

2 Basic Python Syntax and Data Types

2.1 Variables, Constants, and Basic Data Types (`int`, `float`, `str`, `bool`)

In Python, understanding variables and basic data types is essential as they form the foundation of your programs. This section will explain how to declare and initialize variables, introduce dynamic typing, discuss mutable vs. immutable types, and cover the core data types: integers, floats, strings, and booleans. We'll also talk about naming conventions and how to simulate constants in Python.

2.1.1 Variables in Python

Declaring and Initializing Variables

Unlike some other programming languages, Python **does not require explicit declaration** of variables. You simply assign a value to a variable name, and Python takes care of the rest.

Example:

```
age = 25
name = "Alice"
temperature = 36.6
is_raining = False
```

Here:

- `age` is assigned an integer value `25`.
- `name` is assigned a string `"Alice"`.
- `temperature` is assigned a floating-point number `36.6`.
- `is_raining` is assigned a boolean `False`.

Dynamic Typing

Python is **dynamically typed**, meaning the type of a variable is determined at runtime and you can change the variable's type by assigning a new value.

```
x = 10          # x is an int
x = "hello"     # now x is a str
```

You don't have to declare variable types explicitly — Python figures it out automatically.

2.1.2 Mutable vs. Immutable Types

- **Immutable types** cannot be changed after they are created. When you modify them, a new object is created.

- **Mutable types** can be changed in place.

Core immutable types include:

- `int`
- `float`
- `str`
- `bool`
- `tuple` (not covered in this chapter)

Mutable types include:

- `list`
- `dict`
- `set`

For this chapter, focus on the immutable types (`int`, `float`, `str`, `bool`).

2.1.3 Basic Data Types in Python

Integers (`int`)

Integers represent whole numbers without a fractional part.

Full runnable code:

```
x = 42
print(type(x))  # Output: <class 'int'>
```

You can perform arithmetic on integers:

```
a = 5
b = 3
print(a + b)  # Output: 8
print(a * b)  # Output: 15
```

Floating-Point Numbers (`float`)

Floats represent numbers with decimal points.

Full runnable code:

```
pi = 3.14159
temperature = -4.5
print(type(pi))  # Output: <class 'float'>
```

You can convert between `int` and `float`:

```
num = 7
print(float(num))  # Output: 7.0
```

```
f = 3.8
print(int(f))       # Output: 3 (decimal part truncated)
```

Strings (`str`)

Strings are sequences of characters enclosed in quotes.

Full runnable code:

```
message = "Hello, Python!"
print(type(message))  # Output: <class 'str'>
```

Strings can use single (`'...'`) or double quotes (`"..."`):

```
s1 = 'Single quotes'
s2 = "Double quotes"
```

You can combine (concatenate) strings using `+`:

Full runnable code:

```
greeting = "Hello, " + "world!"
print(greeting)  # Output: Hello, world!
```

Booleans (`bool`)

Booleans represent truth values: `True` or `False`.

Full runnable code:

```
is_sunny = True
print(type(is_sunny))  # Output: <class 'bool'>
```

Booleans are often used in conditions and comparisons:

Full runnable code:

```
print(5 > 3)  # Output: True
print(2 == 4) # Output: False
```

2.1.4 Type Conversion Examples

You can convert between types using functions like `int()`, `float()`, `str()`, and `bool()`.

Full runnable code:

```
x = "123"
print(int(x))       # Output: 123 (string to int)
```

```
y = 0
print(bool(y))      # Output: False (0 is Falsey)

z = 3.7
print(int(z))       # Output: 3 (float to int truncates decimal)
```

2.1.5 Naming Conventions and Constants

Variable Names

- Should start with a letter (a-z, A-Z) or underscore (_).
- Can contain letters, digits, and underscores.
- Are case-sensitive (`age` and `Age` are different).
- Use **snake_case** for variable names by convention:

```
user_name = "Alice"
total_score = 100
```

Constants

Python does not have built-in constants, but by convention, variables that should not change are written in **all uppercase letters** with underscores separating words:

```
PI = 3.14159
MAX_USERS = 100
```

This signals to other programmers that these values should be treated as constants.

2.1.6 Summary

Concept	Description	Example
Variable assignment	Assign values without declaring types	`x = 5`
Dynamic typing	Variable types determined at runtime	`x = 10` then `x = "Hi"`
Immutable types	Cannot be changed after creation	`int`, `float`, `str`, `bool`
Mutable types	Can be changed after creation	`list`, `dict`
Core data types	Basic types for numbers, text, and logic	`int`, `float`, `str`, `bool`
Naming conventions	Use snake_case for variables	`user_age = 25`
Simulated constants	Use uppercase for values not meant to change	`MAX_SPEED = 120`

With these basics, you're ready to start working with data in Python. In the next sections, we'll explore operators, expressions, and how to write meaningful computations.

2.2 Operators and Expressions

In Python, **operators** are symbols or keywords that perform operations on values (called **operands**) to produce new values. An **expression** is a combination of values, variables, and operators that Python evaluates to produce a result.

This section will introduce the most common types of operators in Python: arithmetic, comparison, logical, and assignment. We'll also explore operator precedence and how parentheses affect the order of evaluation.

2.2.1 Arithmetic Operators

Arithmetic operators perform mathematical calculations.

Operator	Description	Example	Output
`+`	Addition	`5 + 3`	`8`
`-`	Subtraction	`10 - 4`	`6`
`*`	Multiplication	`2 * 7`	`14`
`/`	Division (float)	`10 / 4`	`2.5`
`//`	Floor division (int)	`10 // 4`	`2`
`%`	Modulus (remainder)	`10 % 4`	`2`
`**`	Exponentiation	`2 ** 3`	`8`

Examples:

Full runnable code:

```
print(5 + 2 * 3)    # Output: 11
print((5 + 2) * 3)  # Output: 21
print(10 / 3)       # Output: 3.3333333333333335
print(10 // 3)      # Output: 3
print(10 % 3)       # Output: 1
print(2 ** 4)       # Output: 16
```

2.2.2 Comparison Operators

Comparison operators compare two values and return a boolean result (`True` or `False`).

Operator	Description	Example	Output
`==`	Equal to	`5 == 5`	`True`
`!=`	Not equal to	`3 != 7`	`True`
`<`	Less than	`4 < 9`	`True`
`>`	Greater than	`8 > 10`	`False`
`<=`	Less than or equal	`7 <= 7`	`True`
`>=`	Greater than or equal	`6 >= 5`	`True`

Examples:

Full runnable code:

```
print(5 == 5)   # Output: True
print(4 != 4)   # Output: False
print(7 > 10)   # Output: False
print(3 <= 3)   # Output: True
```

2.2.3 Logical Operators

Logical operators combine boolean expressions.

Operator	Description	Example	Output
`and`	Logical AND	`True and False`	`False`
`or`	Logical OR	`True or False`	`True`
`not`	Logical NOT	`not True`	`False`

Examples:

Full runnable code:

```
x = 5
print(x > 3 and x < 10)   # Output: True
print(x == 5 or x == 0)   # Output: True
print(not(x > 3))         # Output: False
```

2.2.4 Assignment Operators

Assignment operators assign values to variables and often combine arithmetic operations.

Operator	Description	Example	Effect
`=`	Simple assignment	`x = 5`	Assign 5 to x
`+=`	Add and assign	`x += 3`	`x = x + 3`
`-=`	Subtract and assign	`x -= 2`	`x = x - 2`
`*=`	Multiply and assign	`x *= 4`	`x = x * 4`
`/=`	Divide and assign	`x /= 2`	`x = x / 2`
`%=`	Modulus and assign	`x %= 3`	`x = x % 3`

Example:

Full runnable code:

```
x = 10
x += 5
print(x)  # Output: 15

x *= 2
print(x)  # Output: 30

x %= 7
print(x)  # Output: 2
```

2.2.5 Operator Precedence and Parentheses

Python evaluates expressions based on **operator precedence** — some operators are performed before others.

Precedence order (high to low):

1. Parentheses `()`
2. Exponentiation `**`
3. Unary plus and minus `+x`, `-x`, `~x`
4. Multiplication, division, modulus, floor division `*`, `/`, `%`, `//`
5. Addition and subtraction `+`, `-`
6. Comparison operators `<`, `>`, `<=`, `>=`, `==`, `!=`
7. Logical NOT `not`
8. Logical AND `and`
9. Logical OR `or`
10. Assignment `=`, `+=`, `-=`, etc. (lowest precedence)

Use parentheses to control evaluation order explicitly.

Example:

Full runnable code:

```
result1 = 5 + 3 * 2
result2 = (5 + 3) * 2

print(result1)  # Output: 11  (multiplication first)
print(result2)  # Output: 16  (parentheses force addition first)
```

2.2.6 Combining Operators in Expressions

You can create complex expressions by combining different operators.

Full runnable code:

```
x = 10
y = 5
z = 3

result = (x + y) > z and not (y == 0)
print(result)  # Output: True
```

Explanation:

- `(x + y) > z` evaluates to `(10 + 5) > 3`, which is `15 > 3` → `True`.
- `y == 0` is `5 == 0` → `False`, so `not (y == 0)` is `True`.
- `True and True` evaluates to `True`.

2.2.7 Summary

Operator Type	Examples	Description
Arithmetic	`+`, `-`, `*`, `/`, `//`, `%`, `**`	Basic math operations
Comparison	`==`, `!=`, `<`, `>`, `<=`, `>=`	Compare values, return `True` or `False`
Logical	`and`, `or`, `not`	Combine boolean expressions
Assignment	`=`, `+=`, `-=`, `*=`, `/=`, `%=`	Assign and update variable values

Remember to use parentheses to clarify and control the order in which operations are performed.

2.3 Comments and Naming Conventions

Writing clear and maintainable code is just as important as making your program work. **Comments** and **naming conventions** are key tools that help you and others understand your code better. This section will introduce you to Python's commenting styles and best practices for naming variables, functions, constants, and classes.

2.3.1 Comments in Python

Comments are lines or blocks of text in your code that **Python ignores** when running your program. They are used to explain, document, or clarify your code to human readers.

Single-Line Comments

- Start with a `#` symbol.
- Everything after the `#` on that line is ignored by Python.

Example:

Full runnable code:

```
# This is a single-line comment
print("Hello, world!")  # This prints a message to the screen
```

Multi-Line Comments

Python does not have a special multi-line comment syntax like some other languages, but you can create multi-line comments by:

- Using multiple single-line comments:

  ```
  # This is a comment line 1
  # This is a comment line 2
  # This is a comment line 3
  ```

- Using **multi-line strings** (triple quotes), often for documentation or longer comments:

  `python try """ This is a multi-line comment or docstring. It can span multiple lines. """ print("Example")`

  Note: Multi-line strings are actually string literals, and if not assigned or used, Python ignores them. They are commonly used for function or module documentation (docstrings).

Why Use Comments?

- Explain **why** your code does something (not just what it does).
- Clarify complex logic.
- Make your code easier to read and maintain.

- Provide instructions or warnings.

Example of Well-Commented Code

Full runnable code:

```
# Calculate the area of a rectangle
width = 10    # width of the rectangle in units
height = 5    # height of the rectangle in units

area = width * height  # area formula: width times height

print("Area:", area)   # output the result
```

Example of Bad Commenting Practices

```
x = 5  # set x to 5
y = 10 # set y to 10
z = x + y  # add x and y
print(z)   # print z
```

- These comments restate obvious code, adding no value.
- Instead, focus on **why** or explaining non-trivial parts.

2.3.2 Naming Conventions in Python

Good names make your code easier to understand and maintain. Python follows established naming conventions that are widely accepted in the programming community.

Variables and Functions: `snake_case`

- Use **lowercase letters** with words separated by underscores `_`.
- Descriptive names are better than short or vague names.

```
user_age = 30
total_score = 100

def calculate_area(width, height):
    return width * height
```

Constants: ALL_UPPERCASE

- Constants are variables meant to stay the same during program execution.
- Use uppercase letters with underscores to separate words.

```
MAX_SPEED = 120
PI = 3.14159
```

Classes: `CamelCase`

- Class names use **CamelCase**, where each word starts with a capital letter and no underscores.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
```

2.3.3 Summary of Naming Conventions

Type	Naming Style	Example
Variable	`snake_case`	`user_name`, `total_price`
Function	`snake_case`	`calculate_area()`, `get_data()`
Constant	`ALL_UPPERCASE`	`MAX_LIMIT`, `PI`
Class	`CamelCase`	`Person`, `CarModel`

2.3.4 Final Tips

- Be consistent with your naming conventions throughout your code.
- Choose descriptive and meaningful names.
- Use comments to explain **why** your code does something, especially if it isn't obvious.
- Avoid redundant or obvious comments.
- Use docstrings (triple quotes) to document functions and classes (covered in later chapters).

2.4 Simple Input and Output

Interacting with the user is a key part of many Python programs. Python provides simple but powerful tools to **receive input** from users and **display output** on the screen. In this section, you will learn how to use the `input()` function to get user input, the `print()` function to show messages or results, how to convert input into numbers, and how to format output neatly using **f-strings**.

2.4.1 Receiving Input with `input()`

The `input()` function pauses your program and waits for the user to type something on the keyboard, then returns that input as a **string**.

```
name = input("What is your name? ")
print("Hello, " + name + "!")
```

Example interaction:

```
What is your name? Alice
Hello, Alice!
```

2.4.2 Converting Input to Numbers

Since `input()` returns a string, you often need to convert that input into numeric types like `int` or `float` to perform calculations.

```
age_str = input("Enter your age: ")     # age_str is a string
age = int(age_str)                       # convert to integer

height_str = input("Enter your height in meters: ")
height = float(height_str)               # convert to float

print("You are", age, "years old and", height, "meters tall.")
```

Example:

```
Enter your age: 30
Enter your height in meters: 1.75
You are 30 years old and 1.75 meters tall.
```

2.4.3 Displaying Output with `print()`

The `print()` function outputs data to the console. You can print multiple values separated by commas:

```
print("Age:", age, "Height:", height)
```

Formatting Output with f-strings

Python 3.6+ introduced **f-strings**, which allow you to embed expressions inside string literals using curly braces `{}` for clean and readable formatting.

```
print(f"{name} is {age} years old and {height:.2f} meters tall.")
```

- `{height:.2f}` formats the float to 2 decimal places.

Output:

```
Alice is 30 years old and 1.75 meters tall.
```

2.4.4 Example 1: Greeting Generator

```
name = input("Enter your name: ")
print(f"Hello, {name}! Welcome to Python programming.")
```

2.4.5 Example 2: Simple Calculator (Addition)

This program takes two numbers from the user, converts them to integers, adds them, and displays the result.

```
num1 = int(input("Enter the first number: "))
num2 = int(input("Enter the second number: "))

total = num1 + num2
print(f"The sum of {num1} and {num2} is {total}.")
```

Sample run:

```
Enter the first number: 7
Enter the second number: 5
The sum of 7 and 5 is 12.
```

2.4.6 Summary

- Use `input(prompt)` to get user input as a string.
- Convert input strings to `int` or `float` using `int()` and `float()` for calculations.
- Use `print()` to display output; separate multiple items with commas.
- Use **f-strings** (`f"..."`) to embed variables and expressions neatly in strings.
- Practice combining input and output to build interactive programs.

Chapter 3.

Control Flow

1. Conditional Statements: `if`, `elif`, `else`
2. Loops: `for`, `while`
3. Loop Control Statements: `break`, `continue`, `else` in loops

3 Control Flow

3.1 Conditional Statements: `if`, `elif`, `else`

In programming, making decisions is essential. Python uses **conditional statements**—`if`, `elif`, and `else`—to execute different blocks of code based on conditions. This allows your program to choose different paths depending on the input or state.

3.1.1 Understanding `if`, `elif`, and `else`

The `if` Statement

The `if` statement evaluates a condition (an expression that results in `True` or `False`). If the condition is `True`, the indented block below the `if` runs. Otherwise, Python skips it.

Syntax:

```
if condition:
    # code to run if condition is True
```

Example:

Full runnable code:

```
age = 18
if age >= 18:
    print("You are an adult.")
```

The `elif` Statement

`elif` (short for "else if") provides additional conditions to check if the previous `if` or `elif` conditions were `False`.

Syntax:

```
if condition1:
    # run if condition1 is True
elif condition2:
    # run if condition1 is False and condition2 is True
```

Example:

Full runnable code:

```
score = 75
if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
elif score >= 70:
```

```
    print("Grade: C")
```

The `else` Statement

`else` runs if **all previous conditions are False**.

Syntax:

```
if condition1:
    # run if condition1 is True
else:
    # run if condition1 is False
```

Example:

Full runnable code:

```
age = 15
if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")
```

Complete Syntax with All Parts

```
if condition1:
    # code block 1
elif condition2:
    # code block 2
else:
    # code block 3
```

- You can have **zero or more** `elif` statements.
- The `else` block is optional.

3.1.2 Indentation Is Essential

All code inside the blocks after `if`, `elif`, and `else` **must be indented** (typically 4 spaces).

Incorrect indentation causes errors or unexpected behavior.

3.1.3 Using Logical Conditions

You can combine multiple conditions using **logical operators**:

- `and` — True if *both* conditions are true.
- `or` — True if *at least one* condition is true.

- `not` — Inverts the condition.

Example:

Full runnable code:

```
age = 20
has_id = True

if age >= 18 and has_id:
    print("You can enter the club.")
else:
    print("Access denied.")
```

3.1.4 Nested Conditionals

You can place one conditional inside another to check multiple levels of conditions.

Example:

Full runnable code:

```
score = 85

if score >= 60:
    if score >= 90:
        print("Grade: A")
    else:
        print("Grade: Pass")
else:
    print("Grade: Fail")
```

3.1.5 Example Program: Grade Classifier

Let's build a simple program that classifies a numeric grade into letter grades:

Full runnable code:

```
grade = int(input("Enter your score (0-100): "))

if grade >= 90:
    print("Grade: A")
elif grade >= 80:
    print("Grade: B")
elif grade >= 70:
    print("Grade: C")
elif grade >= 60:
    print("Grade: D")
```

```
else:
    print("Grade: F")
```

3.1.6 Example Program: Simple Login System

This program checks a username and password:

Full runnable code:

```
username = input("Username: ")
password = input("Password: ")

if username == "admin" and password == "secret123":
    print("Login successful!")
elif username == "admin":
    print("Incorrect password.")
else:
    print("Unknown user.")
```

3.1.7 Summary

- Use `if` to execute code only when a condition is true.
- Use `elif` to check additional conditions if the previous ones fail.
- Use `else` to catch all other cases.
- Indentation defines the blocks of code belonging to each condition.
- Logical operators (`and`, `or`, `not`) help combine conditions.
- Conditionals can be nested for more complex decisions.

Mastering conditional statements lets you control your program's flow and create interactive, decision-based programs.

3.2 Loops: `for`, `while`

Loops allow your program to **repeat a block of code multiple times**, which is useful for tasks like processing items in a list, counting, or accumulating results. Python provides two main types of loops:

- The **`for` loop** for iterating over sequences or ranges.
- The **`while` loop** for repeating as long as a condition is true.

3.2.1 The `for` Loop

When to Use

Use a `for` loop when you know **how many times you want to iterate** or want to loop over elements in a sequence (like lists, strings, or ranges).

Syntax

```
for variable in sequence:
    # code block to repeat
```

- `variable` takes the value of each element in `sequence` one by one.
- The indented code block runs once per element.

Example 1: Counting Numbers Using `range()`

Full runnable code:

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

Output:

```
0
1
2
3
4
```

`range(5)` generates numbers from 0 up to, but not including, 5.

Example 2: Iterating Over a List

Full runnable code:

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

for fruit in fruits:
    print(f"I like {fruit}")
```

Output:

```
I like apple
I like banana
I like cherry
```

3.2.2 The `while` Loop

When to Use

Use a `while` loop when you want to repeat code **until a condition changes**, and the number of repetitions is not known in advance.

Syntax

```
while condition:
    # code block to repeat
```

- The condition is checked before each iteration.
- If the condition is `True`, the loop runs; if `False`, it stops.

Example: Counting with `while`

Full runnable code:

```
count = 0
while count < 5:
    print(count)
    count += 1  # increment count to avoid infinite loop
```

Output:

```
0
1
2
3
4
```

3.2.3 How Loops Execute

- **`for` loops** execute once per item in the sequence.
- **`while` loops** execute repeatedly as long as the condition remains `True`.
- Be sure to **update variables** involved in the condition inside a `while` loop to avoid **infinite loops** (loops that never end).

3.2.4 Avoiding Infinite Loops

An infinite loop happens if the condition never becomes `False`.

Example of an infinite loop (do not run!):

```python
while True:
    print("This will print forever!")
```

To stop infinite loops, make sure your loop variable changes so the condition eventually becomes false.

3.2.5 Real-World Task 1: Multiplication Table Using `for`

Print the multiplication table for a given number.

Full runnable code:

```python
num = int(input("Enter a number: "))

print(f"Multiplication table for {num}:")
for i in range(1, 11):
    print(f"{num} x {i} = {num * i}")
```

Sample output for input `5`:

```
Multiplication table for 5:
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
...
5 x 10 = 50
```

3.2.6 Real-World Task 2: Computing Factorials Using `while`

Calculate the factorial of a number (`n! = n * (n-1) * ... * 1`).

Full runnable code:

```python
n = int(input("Enter a positive integer: "))
factorial = 1
counter = n

while counter > 1:
    factorial *= counter
    counter -= 1

print(f"{n}! = {factorial}")
```

Example output for input `5`:

```
5! = 120
```

3.2.7 Summary

Loop Type	Use When	Syntax
`for`	Known number of iterations or iterating over sequences	`for variable in sequence:`
`while`	Unknown number of iterations, repeat until condition false	`while condition:`

- Loops execute the indented code repeatedly.
- Always update loop variables or conditions in `while` loops to avoid infinite loops.
- Use `for` loops for fixed or iterable sequences.
- Use `while` loops when repetition depends on a changing condition.

3.3 Loop Control Statements: `break`, `continue`, `else` in loops

Python provides special **loop control statements** that let you change how loops behave while they run:

- `break` — exits the loop immediately.
- `continue` — skips the rest of the current iteration and moves to the next.
- `else` (with loops) — runs a block of code after the loop completes **only if the loop was not exited by `break`**.

These statements help write clearer and more efficient loops.

3.3.1 The `break` Statement

- When Python encounters `break` inside a loop, it **exits the entire loop immediately**, skipping any remaining iterations.

Example: Searching for a Number

Full runnable code:

```
numbers = [4, 7, 10, 3, 8]
search_for = 3

for num in numbers:
    if num == search_for:
        print(f"Found {search_for}!")
        break  # exit the loop as soon as we find the number
```

```
else:
    print(f"{search_for} not found.")
```

Expected behavior:

- The loop stops and prints `"Found 3!"` once it finds `3`.
- The `else` after the loop will **not run** because of the `break`.

3.3.2 The `continue` Statement

- `continue` **skips the rest of the current loop iteration** and jumps immediately to the next iteration.

Example: Skipping Even Numbers

Full runnable code:

```
for i in range(1, 6):
    if i % 2 == 0:
        continue  # skip even numbers
    print(i)
```

Output:

```
1
3
5
```

Explanation:

- When `i` is even, `continue` skips the `print(i)` line.
- Only odd numbers are printed.

3.3.3 The `else` Clause on Loops

- Python loops can have an **optional `else` block** that executes **only if the loop completes normally** (i.e., no `break` occurred).
- This is useful to detect if the loop finished all iterations without interruption.

Example: Checking for Prime Numbers

Full runnable code:

```
num = 17
```

```
if num < 2:
    print(f"{num} is not prime.")
else:
    for i in range(2, num):
        if num % i == 0:
            print(f"{num} is not prime; divisible by {i}.")
            break
    else:
        print(f"{num} is prime!")
```

Explanation:

- The `for` loop tries to find a divisor of `num`.
- If it finds one, it prints a message and `break`s the loop.
- If no divisor is found (`break` never executes), the `else` block runs and declares the number prime.

3.3.4 Practical Example: Input Validation Using `break`

Full runnable code:

```
while True:
    password = input("Enter password (at least 6 characters): ")
    if len(password) >= 6:
        print("Password accepted.")
        break  # exit loop once valid password is entered
    else:
        print("Password too short. Try again.")
```

Here, the loop will continue prompting until the user enters a valid password, then exit using `break`.

3.3.5 Summary

Statement	Behavior	Example Use Case
`break`	Exits the loop immediately	Stop searching when found item
`continue`	Skips to next loop iteration	Skip processing invalid data
`else`	Runs after loop finishes normally (no `break`)	Confirm no early exit occurred

3.3.6 Visualizing Behavior

Loop Execution	`break` hit?	Does `else` run?	Description
Loop completes	No	Yes	`else` block runs after loop
Loop exits early	Yes	No	`else` block is skipped

Using `break`, `continue`, and loop `else` together can make your code cleaner and more expressive.

Chapter 4.

Functions and Modules

1. Defining and Calling Functions
2. Function Arguments and Return Values
3. Variable Scope and Lifetime
4. Importing Modules and Using Standard Library

4 Functions and Modules

4.1 Defining and Calling Functions

Functions are fundamental building blocks in Python that let you **group reusable code** into a single, named block. This makes your programs easier to read, test, and maintain. Instead of writing the same code multiple times, you can **define a function once** and **call it whenever needed**.

4.1.1 Defining a Function

In Python, you define a function using the `def` keyword, followed by the function name, parentheses `()`, and a colon `:`. The code inside the function is **indented**, indicating the function body.

Syntax

```
def function_name():
    # function body (indented code)
    # do something
```

- **`def`** — keyword to define a function.
- **`function_name`** — a descriptive name following naming conventions (usually `snake_case`).
- **Parentheses `()`** — indicate a function; may include parameters (covered later).
- **Colon `:`** — starts the indented block of code that makes up the function.

4.1.2 Calling a Function

To run the function's code, you **call** the function by writing its name followed by parentheses:

```
function_name()
```

4.1.3 Example 1: A Simple Greeting Function

Full runnable code:

```
def greet():
    print("Hello! Welcome to Python programming.")
```

```
greet()
```

Output:

```
Hello! Welcome to Python programming.
```

- The function `greet` contains one statement: a print message.
- Calling `greet()` runs the function and prints the message.

4.1.4 Example 2: Function to Calculate Area of a Rectangle

Full runnable code:

```
def calculate_area(width, height):
    area = width * height
    print(f"The area is {area} square units.")

calculate_area(5, 3)
```

Output:

```
The area is 15 square units.
```

Explanation:

- The function `calculate_area` takes two inputs: `width` and `height`.
- It calculates the area by multiplying them.
- The result is printed inside the function.
- Calling `calculate_area(5, 3)` runs the function with `width=5` and `height=3`.

4.1.5 Why Use Functions?

- **Reusability:** Write code once and use it many times.
- **Organization:** Break large programs into smaller, manageable pieces.
- **Readability:** Give meaningful names to blocks of code to clarify their purpose.
- **Testing:** Isolate parts of code for easier debugging.

4.1.6 Important Notes

- Function names should be **descriptive** and use `snake_case`, e.g., `calculate_area`, `print_greeting`.
- The function body **must be indented** (usually 4 spaces).

- Functions do not execute until called.
- Functions can accept **parameters** to work with different inputs (explored further in the next section).

4.1.7 Summary

Concept	Description	Example
Define function	Use `def function_name():` followed by indented code	`def greet():`
Call function	Use `function_name()` to execute the function	`greet()`
Function body	Indented code that runs when called	`print("Hello")`
Naming	Use descriptive names in `snake_case`	`calculate_area`

Functions are essential for writing clean and efficient Python programs. Mastering function definition and calls is your first step toward building powerful reusable code blocks.

4.2 Function Arguments and Return Values

Functions become more useful when you can **pass data to them** and get back results. This section explains how to provide inputs to functions using **arguments** and how functions can **return values** to the caller instead of just printing output.

4.2.1 Passing Arguments to Functions

When you call a function, you can provide **arguments**—values that the function uses to perform its task. These are specified inside the parentheses.

Positional Arguments

Arguments passed in order are called **positional arguments**. The first argument is assigned to the first parameter, the second to the second, and so on.

Example:

Full runnable code:

```
def greet(first_name, last_name):
    print(f"Hello, {first_name} {last_name}!")
```

```
greet("Alice", "Smith")
```

Output:

```
Hello, Alice Smith!
```

- `"Alice"` goes to `first_name`.
- `"Smith"` goes to `last_name`.

Keyword Arguments

You can specify arguments by **name** to make code clearer or change their order.

```
greet(last_name="Johnson", first_name="Bob")
```

Output:

```
Hello, Bob Johnson!
```

Default Parameter Values

Functions can provide **default values** for parameters. If an argument is omitted during the call, the default is used.

Full runnable code:

```
def greet(first_name, last_name="Doe"):
    print(f"Hello, {first_name} {last_name}!")

greet("Jane")  # last_name defaults to "Doe"
greet("John", "Smith")
```

Output:

```
Hello, Jane Doe!
Hello, John Smith!
```

4.2.2 Returning Values with `return`

While functions can print results, it's often better to **return a value** so the caller can decide what to do with it.

The `return` Statement

- Ends the function and sends a value back to where the function was called.
- If no `return` is used, the function returns `None` by default.

Example: Function Returning Maximum of Three Numbers

```
def max_of_three(a, b, c):
    if a >= b and a >= c:
        return a
    elif b >= a and b >= c:
        return b
    else:
        return c

result = max_of_three(10, 25, 15)
print(f"The maximum is {result}")
```

Output:

```
The maximum is 25
```

Example: Function Formatting a Full Name

Full runnable code:

```
def format_full_name(first_name, last_name, middle_name=""):
    if middle_name:
        full_name = f"{first_name} {middle_name} {last_name}"
    else:
        full_name = f"{first_name} {last_name}"
    return full_name.title()

name1 = format_full_name("john", "doe")
name2 = format_full_name("jane", "smith", "elizabeth")

print(name1)  # John Doe
print(name2)  # Jane Elizabeth Smith
```

4.2.3 Summary

Concept	Explanation	Example
Positional Arguments	Arguments passed in order	`greet("Alice", "Smith")`
Keyword Arguments	Arguments passed by name	`greet(last_name="Smith", first_name="Alice")`
Default Parameter Values	Provide defaults for parameters	`def greet(name="User"):`
`return` Statement	Send a value back from a function	`return max_value`

4.2.4 Why Return Values?

- Allows the function to produce data that can be stored, manipulated, or printed later.
- Makes functions more flexible and reusable.
- Keeps input/output (I/O) separate from logic.

4.3 Variable Scope and Lifetime

Understanding **variable scope** and **lifetime** is crucial to writing reliable and bug-free Python programs. These concepts determine **where a variable can be accessed** (scope) and **how long it exists** (lifetime) during program execution.

4.3.1 What Is Variable Scope?

Scope refers to the region of a program where a variable is recognized and can be used.

In Python, variables have either:

- **Local scope** — inside a function or block.
- **Global scope** — outside any function, accessible throughout the module.

4.3.2 Local Variables

- Variables defined **inside a function** have **local scope**.
- They exist only **while the function is running**.
- They cannot be accessed from outside the function.

Example: Local Variable

Full runnable code:

```
def greet():
    message = "Hello, world!"  # local variable
    print(message)

greet()
# print(message)  # This would cause an error: NameError: name 'message' is not defined
```

- `message` is local to `greet()` and not accessible outside it.

4.3.3 Global Variables

- Variables defined **outside all functions** have **global scope**.
- They can be accessed inside functions **unless a local variable with the same name exists**.
- Global variables exist for the duration of the program.

Example: Global Variable Accessed Inside Function

```
greeting = "Hello"

def greet():
    print(greeting)  # accesses global variable

greet()  # Output: Hello
```

4.3.4 Variable Shadowing (Local vs Global)

If a local variable has the same name as a global variable, the local variable **shadows** (hides) the global one inside the function.

Full runnable code:

```
count = 10  # global variable

def print_count():
    count = 5  # local variable shadows global 'count'
    print(count)

print_count()  # Output: 5
print(count)   # Output: 10
```

4.3.5 The `global` Keyword

If you need to **modify** a global variable inside a function, use the `global` keyword.

Full runnable code:

```
counter = 0

def increment():
    global counter
    counter += 1

increment()
print(counter)  # Output: 1
```

- Without `global`, `counter += 1` inside `increment()` would raise an error because Python treats `counter` as a new local variable.

4.3.6 Why Avoid Excessive Use of Global Variables?

- Makes code harder to understand and debug.
- Changes to globals can have unintended side effects.
- Functions become less predictable because they depend on outside state.

Best practice: Pass variables as parameters and return results rather than relying on globals.

4.3.7 Lifetime of Variables

- **Local variables**: created when the function starts, destroyed when it ends.
- **Global variables**: created when the program starts, destroyed when it ends.

4.3.8 Example: Demonstrating Scope and Lifetime

Full runnable code:

```
x = 100  # global variable

def func():
    x = 50  # local variable
    print(f"Inside func, x = {x}")

func()
print(f"Outside func, x = {x}")
```

Output:

```
Inside func, x = 50
Outside func, x = 100
```

4.3.9 Summary

Concept	Description	Example
Local variable	Exists only inside function, shadows globals	`def f(): x = 5`
Global variable	Defined outside functions, accessible globally	`x = 10`
`global` keyword	Allows modifying a global variable inside a function	`global x` inside function
Variable lifetime	Local: during function call; Global: entire program	—

Understanding variable scope and lifetime helps you avoid bugs and write cleaner, more maintainable code.

4.4 Importing Modules and Using Standard Library

As your Python programs grow, organizing code into **modules** helps keep things clean and reusable. Python provides a rich **standard library** — a collection of built-in modules you can import to add powerful features without writing everything from scratch.

4.4.1 What Are Modules?

- A **module** is a file containing Python code (functions, classes, variables) that you can **reuse** by importing it into your program.
- The Python **standard library** includes many modules for common tasks like math, random numbers, date/time, file handling, and more.

4.4.2 How to Import Modules

Using `import`

The simplest way to use a module is with the `import` keyword:

Full runnable code:

```
import math

print(math.sqrt(16))  # Output: 4.0
```

- This imports the entire `math` module.
- You access functions with the syntax: `module_name.function_name()`.

Using `from ... import`

To import specific functions or variables from a module:

Full runnable code:

```
from math import sqrt

print(sqrt(25))  # Output: 5.0
```

- This imports only `sqrt` directly into your namespace, so you don't need to prefix it with `math`..

Aliasing with `as`

You can give a module or function an alias to shorten or clarify its name:

Full runnable code:

```
import random as rnd

print(rnd.randint(1, 10))  # Generates a random integer between 1 and 10
```

4.4.3 Common Standard Library Modules and Examples

`math` Module

Useful for mathematical operations:

Full runnable code:

```
import math

print(math.sqrt(9))          # Square root, Output: 3.0
print(math.pi)               # Value of pi, Output: 3.1415926535897793
print(math.factorial(5))     # 5! = 120
```

`random` Module

Generate random numbers and make random choices:

Full runnable code:

```
import random

print(random.randint(1, 100))    # Random integer between 1 and 100
print(random.choice(['apple', 'banana', 'cherry']))  # Randomly select from list
```

`datetime` Module

Work with dates and times:

Full runnable code:

```
from datetime import datetime, timedelta

now = datetime.now()
print("Current date and time:", now)

# Add 5 days to current date
future_date = now + timedelta(days=5)
print("Date 5 days from now:", future_date.date())
```

4.4.4 Summary

Import Style	Usage	Example
Import entire module	Access with `module_name.function()`	`import mathmath.sqrt(4)`
Import specific items	Use functions directly	`from math import sqrtsqrt(4)`
Import with alias	Shorten module name	`import random as rndrnd.randint(1,10)`

4.4.5 Why Use Modules?

- Avoid reinventing the wheel by using pre-built tools.
- Organize your code logically.
- Share and reuse code across projects.
- Access a vast ecosystem of built-in Python functionality.

Chapter 5.

Data Structures

1. Lists and List Comprehensions
2. Tuples and Sets
3. Dictionaries and Dictionary Comprehensions
4. Practical Use Cases of Each Data Structure

5 Data Structures

5.1 Lists and List Comprehensions

Lists are one of the most versatile and commonly used data structures in Python. They allow you to store collections of items — such as numbers, strings, or even other lists — in an ordered and mutable way.

5.1.1 Creating Lists

You create a list by placing items inside square brackets `[]`, separated by commas:

```
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4, 5]
mixed = [1, "hello", True, 3.14]
empty_list = []
```

5.1.2 Accessing List Elements

You access elements by **index**, starting at 0 for the first item:

Full runnable code:

```
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4, 5]
print(fruits[0])  # Output: apple
print(numbers[3]) # Output: 4
```

- Negative indices access elements from the end:

Full runnable code:

```
fruits = ["apple", "banana", "cherry"]
print(fruits[-1])  # Output: cherry
```

5.1.3 Modifying Lists

Lists are **mutable**, meaning you can change their contents:

Changing an Element

Full runnable code:

```
fruits = ["apple", "banana", "cherry"]
fruits[1] = "blueberry"
print(fruits)  # Output: ['apple', 'blueberry', 'cherry']
```

Adding Elements

- Using `append()` to add at the end:

Full runnable code:

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

- Using `insert()` to add at a specific index:

Full runnable code:

```
fruits = ["apple", "banana", "cherry"]
fruits.insert(1, "kiwi")
print(fruits)  # ['apple', 'kiwi', 'blueberry', 'cherry', 'orange']
```

Removing Elements

- Using `remove()` to delete by value:

Full runnable code:

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

- Using `pop()` to delete by index:

Full runnable code:

```
fruits = ["apple", "banana", "cherry"]
last = fruits.pop()  # removes and returns last element
print(last)          # orange
print(fruits)        # ['apple', 'kiwi', 'cherry']
```

5.1.4 List Slicing

You can access **subsections** of lists using slicing:

Full runnable code:

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7]
```

```
print(numbers[2:5])    # Output: [2, 3, 4]  (from index 2 up to but not including 5)
print(numbers[:4])     # Output: [0, 1, 2, 3]  (start to index 4)
print(numbers[5:])     # Output: [5, 6, 7]     (index 5 to end)
print(numbers[-3:])    # Output: [5, 6, 7]     (last three elements)
```

5.1.5 Sorting Lists

- Sort a list **in place** with `sort()`:

Full runnable code:

```
numbers = [5, 3, 8, 1]
numbers.sort()
print(numbers)  # Output: [1, 3, 5, 8]
```

- Get a **sorted copy** with `sorted()`:

Full runnable code:

```
numbers = [5, 3, 8, 1]
sorted_numbers = sorted(numbers)
print(sorted_numbers)  # Output: [1, 3, 5, 8]
print(numbers)         # Original list remains unchanged
```

5.1.6 List Comprehensions

List comprehensions provide a **concise and readable way** to create lists based on existing lists or ranges, often replacing loops.

Basic Syntax

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

Example: Square Numbers 0 to 4

Full runnable code:

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

Conditional List Comprehensions

You can add an `if` condition to filter items:

Full runnable code:

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

Practical Examples

1. **Filtering even numbers from a list:**

Full runnable code:

```
numbers = [1, 2, 3, 4, 5, 6]
evens = [num for num in numbers if num % 2 == 0]
print(evens)  # Output: [2, 4, 6]
```

2. **Squaring elements in a list:**

Full runnable code:

```
nums = [1, 3, 5, 7]
squared = [n**2 for n in nums]
print(squared)  # Output: [1, 9, 25, 49]
```

5.1.7 Summary

Operation	Method / Syntax	Example
Create list	`my_list = [item1, item2, ...]`	`[1, 2, 3]`
Access element	`my_list[index]`	`fruits[0]`
Modify element	`my_list[index] = new_value`	`fruits[1] = "banana"`
Add element	`append()`, `insert()`	`fruits.append("orange")`
Remove element	`remove()`, `pop()`	`fruits.remove("banana")`
Slice list	`my_list[start:end]`	`numbers[2:5]`
Sort list	`sort()`, `sorted()`	`numbers.sort()`
List comprehension	`[expr for item in iterable if condition]`	`[x**2 for x in range(5) if x % 2 == 0]`

Lists and list comprehensions are powerful tools to handle and process collections efficiently and succinctly. Mastering them will greatly enhance your Python coding skills!

5.2 Tuples and Sets

In Python, **tuples** and **sets** are important data structures that serve different purposes. Understanding their unique characteristics helps you choose the right one for your needs.

5.2.1 Tuples: Immutable and Ordered

What is a Tuple?

- A **tuple** is an **ordered**, **immutable** collection of items.
- Once created, the items in a tuple **cannot be changed** (no adding, removing, or modifying elements).
- Tuples are defined using **parentheses `()`** or simply commas.

Creating Tuples

Full runnable code:

```
# Using parentheses
coordinates = (10, 20)
print(coordinates)  # Output: (10, 20)

# Without parentheses (comma-separated)
colors = "red", "green", "blue"
print(colors)  # Output: ('red', 'green', 'blue')

# Single-element tuple (note the comma)
single = (5,)
print(single)  # Output: (5,)
```

Accessing Tuple Elements

Like lists, tuples are **indexed starting at 0**:

Full runnable code:

```
coordinates = (10, 20)
colors = "red", "green", "blue"
print(coordinates[0])  # Output: 10
print(colors[2])       # Output: blue
```

When to Use Tuples

- Use tuples when your data **should not change** throughout the program.
- Commonly used to **group related values**, such as (x, y) coordinates or RGB color values.
- Tuples can be used as keys in dictionaries (because they are immutable).

5.2.2 Sets: Unordered Collections of Unique Elements

What is a Set?

- A **set** is an **unordered** collection of **unique** elements.
- Sets do **not allow duplicates**.
- Defined using **curly braces `{}`** or the `set()` constructor.

Creating Sets

Full runnable code:

```
fruits = {"apple", "banana", "cherry"}
print(fruits)  # Output order may vary because sets are unordered

# Creating an empty set
empty_set = set()

# From a list (removes duplicates)
numbers = [1, 2, 2, 3, 4, 4, 5]
unique_numbers = set(numbers)
print(unique_numbers)  # Output: {1, 2, 3, 4, 5}
```

Adding and Removing Elements

Full runnable code:

```
fruits = {"apple", "banana", "cherry"}
fruits.add("orange")    # Add an element
fruits.remove("banana") # Remove an element
print(fruits)
```

5.2.3 Set Operations

Sets support powerful operations for comparing and combining groups of data:

Operation	Description	Example		
`union` (‘	‘)	All unique elements in both sets	‘{1, 2}	{2, 3}→{1, 2, 3}‘
`intersection` (`&`)	Elements common to both sets	`{1, 2} & {2, 3}` → `{2}`		
`difference` (`-`)	Elements in first set but not second	`{1, 2, 3} - {2}` → `{1, 3}`		
`symmetric_difference` (`^`)	Elements in either set but not both	`{1, 2} ^ {2, 3}` → `{1, 3}`		

Examples:

Full runnable code:

```
set_a = {1, 2, 3, 4}
set_b = {3, 4, 5, 6}

print(set_a | set_b)  # Union: {1, 2, 3, 4, 5, 6}
print(set_a & set_b)  # Intersection: {3, 4}
print(set_a - set_b)  # Difference: {1, 2}
print(set_a ^ set_b)  # Symmetric difference: {1, 2, 5, 6}
```

5.2.4 Practical Examples

Pairing Values with Tuples

Full runnable code:

```
person = ("Alice", 30, "Engineer")
print(f"Name: {person[0]}, Age: {person[1]}, Profession: {person[2]}")
```

Removing Duplicates from a List Using Sets

```
numbers = [1, 2, 2, 3, 4, 4, 5]
unique_numbers = list(set(numbers))
print(unique_numbers)  # Output: [1, 2, 3, 4, 5]
```

Checking Common Elements Between Two Groups

Full runnable code:

```
group1 = {"Alice", "Bob", "Charlie"}
group2 = {"Bob", "Diana", "Charlie"}

common_members = group1 & group2
print(f"Common members: {common_members}")  # Output: {'Bob', 'Charlie'}
```

5.2.5 Summary

Data Structure	Characteristics	Syntax Example	Use Cases
Tuple	Ordered, immutable, can contain duplicates	`point = (3, 5)`	Fixed collections, coordinate pairs

Data Structure	Characteristics	Syntax Example	Use Cases
Set	Unordered, mutable, unique elements	`fruits = {"apple", "banana"}`	Removing duplicates, set math

Tuples and sets serve different needs: tuples when you want an unchangeable sequence of items, sets when you want to store unique elements and perform mathematical set operations.

5.3 Dictionaries and Dictionary Comprehensions

Dictionaries are powerful Python data structures that store **key-value pairs**. They let you organize and access data by unique keys instead of numeric indexes, making them ideal for many practical tasks.

5.3.1 What Is a Dictionary?

- A **dictionary** stores data in **key-value pairs**.
- Keys are **unique** and usually immutable types like strings or numbers.
- Values can be any Python object, including lists or other dictionaries.
- Dictionaries are **mutable**, so you can add, remove, or change items.

Creating a Dictionary

Full runnable code:

```
# Using curly braces with key: value pairs
person = {
    "name": "Alice",
    "age": 30,
    "city": "New York"
}
print(person)
```

Output:

```
{'name': 'Alice', 'age': 30, 'city': 'New York'}
```

5.3.2 Accessing and Modifying Values

Access by Key

Full runnable code:

```
person = {
    "name": "Alice",
    "age": 30,
    "city": "New York"
}

print(person["name"])  # Output: Alice
```

- If the key does not exist, accessing it directly raises a `KeyError`.

Using `.get()` Method

The `.get()` method returns the value for a key, or a default if the key is missing:

Full runnable code:

```
person = {
    "name": "Alice",
    "age": 30,
    "city": "New York"
}

print(person.get("email"))          # Output: None
print(person.get("email", "N/A"))   # Output: N/A
```

Adding or Modifying Entries

Full runnable code:

```
person = {
    "name": "Alice",
    "age": 30,
    "city": "New York"
}

person["email"] = "alice@example.com"  # Add new key-value pair
person["age"] = 31                     # Modify existing value
print(person)
```

5.3.3 Useful Dictionary Methods

Method	Description	Example
`.items()`	Returns a view of (key, value) pairs	`for k, v in person.items(): print(k, v)`
`.keys()`	Returns all keys	`print(person.keys())`
`.values()`	Returns all values	`print(person.values())`
`.update()`	Updates dictionary with another dict or key-value pairs	`person.update({"city": "Boston", "phone": "1234"})`

5.3.4 Dictionary Comprehensions

Dictionary comprehensions let you create dictionaries concisely, similar to list comprehensions.

Syntax

```
{key_expression: value_expression for item in iterable if condition}
```

Example 1: Flipping Keys and Values

Full runnable code:

```
original = {'a': 1, 'b': 2, 'c': 3}
flipped = {value: key for key, value in original.items()}
print(flipped)  # Output: {1: 'a', 2: 'b', 3: 'c'}
```

Example 2: Filtering Entries

Create a dictionary with items where the value is greater than 1:

Full runnable code:

```
original = {'a': 1, 'b': 2, 'c': 3}
filtered = {k: v for k, v in original.items() if v > 1}
print(filtered)  # Output: {'b': 2, 'c': 3}
```

5.3.5 Real-World Examples

Contact Book

Full runnable code:

```
contacts = {
    "Alice": "alice@example.com",
    "Bob": "bob@example.com"
```

```
}

contacts["Charlie"] = "charlie@example.com"  # Add a contact
print(contacts.get("Alice"))                  # Access contact info
```

Word Frequency Counter

Counting how many times each word appears in a list:

Full runnable code:

```
words = ["apple", "banana", "apple", "orange", "banana", "apple"]

frequency = {}
for word in words:
    frequency[word] = frequency.get(word, 0) + 1

print(frequency)  # Output: {'apple': 3, 'banana': 2, 'orange': 1}
```

Or using a dictionary comprehension with the `set()` of words:

Full runnable code:

```
words = ["apple", "banana", "apple", "orange", "banana", "apple"]
frequency = {word: words.count(word) for word in set(words)}
print(frequency)  # Output: {'banana': 2, 'orange': 1, 'apple': 3}
```

5.3.6 Summary

Operation	Description	Example
Create dictionary	Use `{key: value, ...}` or `dict()`	`person = {"name": "Alice"}`
Access value	Use `dict[key]` or `dict.get(key)`	`person["name"]`, `person.get("age")`
Add/modify entry	Assign value to key	`person["email"] = "alice@example.com"`
Iterate items	Use `.items()` to get key-value pairs	`for k, v in person.items():`
Update dictionary	Use `.update()`	`person.update({"city": "Boston"})`
Dictionary comprehension	Create dictionary concisely with expressions and conditions	`{k: v for k, v in original.items() if v > 1}`

Dictionaries and dictionary comprehensions allow you to efficiently manage and manipulate

key-value data in your Python programs.

5.4 Practical Use Cases of Each Data Structure

In this section, we'll summarize the best situations to use **lists, tuples, sets, and dictionaries** — and look at practical mini-project examples for each. Choosing the right data structure improves your program's clarity, efficiency, and maintainability.

5.4.1 Lists: Ordered and Mutable Collections

When to Use

- When you need an **ordered** collection that may change (add, remove, or modify items).
- Ideal for sequences where **duplicates are allowed** and **order matters**.
- Useful when you frequently need to **access elements by position**.

Example: Storing and Processing a To-Do List

Full runnable code:

```
tasks = ["email client", "write report", "call supplier"]
tasks.append("schedule meeting")  # Add new task
print(tasks[0])                   # Access first task
```

Notes

- Lists allow easy iteration, slicing, and sorting.
- Great for stacks, queues, or any collection where order and duplicates matter.

5.4.2 Tuples: Immutable, Ordered Groups

When to Use

- To group a **fixed number of related values** that should not change.
- Useful for **heterogeneous data** (different types) where immutability is important.
- Often used for **coordinates, RGB values, or database records**.

Example: Representing Coordinates

Full runnable code:

```
point = (10, 20)
print(f"X: {point[0]}, Y: {point[1]}")
```

Notes

- Tuples can be used as **dictionary keys** due to immutability.
- Safer than lists when you want to ensure data cannot be modified.

5.4.3 Sets: Unordered Collections of Unique Items

When to Use

- When you need to **store unique elements** and quickly test membership.
- Excellent for **removing duplicates** or performing **set operations** like unions, intersections, and differences.
- Best when **order is not important**.

Example: Tracking Unique Visitors

Full runnable code:

```
visitors = {"alice", "bob", "charlie"}
visitors.add("diana")  # Add new visitor
print("bob" in visitors)  # Check membership
```

Notes

- Fast membership tests (`in` operator).
- Use when uniqueness and set logic are priorities.

5.4.4 Dictionaries: Key-Value Mappings

When to Use

- To associate **keys** with **values** for fast lookups.
- Perfect for **lookup tables**, **caches**, and **counting occurrences**.
- Keys must be immutable types (strings, numbers, tuples).

Example: Counting Word Frequencies

Full runnable code:

```
words = ["apple", "banana", "apple", "orange", "banana", "apple"]
frequency = {}
for word in words:
```

```
    frequency[word] = frequency.get(word, 0) + 1
print(frequency)
# Output: {'apple': 3, 'banana': 2, 'orange': 1}
```

Notes

- Efficient for large datasets with frequent lookups.
- Offers flexibility to map any key to any value.

5.4.5 Summary Table

Data Structure	Characteristics	Best Use Case	Example Use
List	Ordered, mutable, allows duplicates	Ordered collections, sequences	To-do list, queue
Tuple	Ordered, immutable	Fixed collections of related items	Coordinates, RGB colors
Set	Unordered, unique items	Unique elements, fast membership	Unique visitors, removing duplicates
Dictionary	Key-value pairs, mutable	Fast lookups and mappings	Word frequency, contact book

5.4.6 Choosing the Right Data Structure

- Use **lists** when order and duplicates matter.
- Use **tuples** for fixed, immutable groupings.
- Use **sets** when you need uniqueness and fast membership tests.
- Use **dictionaries** to associate keys with values efficiently.

Mini-Project Ideas

- **Word Counter**: Use a dictionary to count the frequency of words in a text.
- **Unique Items Tracker**: Use a set to store unique product IDs visited by users.
- **Task Manager**: Use a list to store tasks with order and priority.
- **Coordinate Storage**: Use tuples to represent fixed points on a grid.

Choosing the right data structure from the start saves you from performance issues and messy code later on. Practice these examples to build a strong foundation in Python data handling!

Chapter 6.

File Handling

1. Reading and Writing Text Files
2. Working with CSV and JSON Files
3. Using Context Managers (`with` Statement)

6 File Handling

6.1 Reading and Writing Text Files

Working with files is essential for many Python programs, whether it's saving data, reading configurations, or processing information. In this section, we'll explore how to **open**, **read**, **write**, and **close** text files using Python's built-in `open()` function.

6.1.1 Opening Files with `open()`

The `open()` function opens a file and returns a file object that you can use to read or write.

File Modes

Mode	Description
`'r'`	Read mode (default). File must exist.
`'w'`	Write mode. Creates file or truncates existing file.
`'a'`	Append mode. Adds to the end of the file.

Syntax

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

6.1.2 Reading Text Files

Reading the Entire File

Full runnable code:

```
try:
    file = open("names.txt", 'r')
    content = file.read()
    print(content)
finally:
    file.close()
```

- Always close the file after reading or writing to free system resources.

Reading Line by Line

Full runnable code:

```
try:
    file = open("names.txt", 'r')
    for line in file:
        print(line.strip())  # strip() removes trailing newline characters
finally:
    file.close()
```

6.1.3 Writing to Text Files

Writing New Content (Overwrites Existing File)

Full runnable code:

```
try:
    file = open("log.txt", 'w')
    file.write("User logged in at 10:00 AM\n")
    file.write("User performed an action.\n")
finally:
    file.close()
```

Appending to an Existing File

Full runnable code:

```
try:
    file = open("log.txt", 'a')
    file.write("User logged out at 11:00 AM\n")
finally:
    file.close()
```

6.1.4 Example: Writing User Input to a Log File

Full runnable code:

```
user_input = input("Enter a message to log: ")

try:
    with open("user_log.txt", 'a') as log_file:
        log_file.write(user_input + "\n")
    print("Message logged successfully.")
except IOError:
    print("An error occurred while writing to the file.")
```

6.1.5 Handling File Not Found Errors

If you try to open a non-existent file in `'r'` mode, Python raises a `FileNotFoundError`. Use a `try-except` block to handle this gracefully:

Full runnable code:

```
try:
    file = open("missing_file.txt", 'r')
    content = file.read()
    file.close()
except FileNotFoundError:
    print("The file does not exist.")
```

6.1.6 Summary

Action	Code Example	Notes
Open a file	`open("file.txt", 'r')`	Modes: `'r'`, `'w'`, `'a'`
Read whole file	`content = file.read()`	Returns entire file as string
Read line by line	`for line in file:`	Useful for large files
Write to file	`file.write("text\n")`	Overwrites or appends depending on mode
Close file	`file.close()`	Frees resources
Handle errors	Use `try-except FileNotFoundError`	Prevents crashes on missing files

By mastering file reading and writing, you can store and retrieve data persistently, making your programs more useful and interactive.

6.2 Working with CSV and JSON Files

In many real-world applications, data is stored in formats like **CSV (Comma-Separated Values)** and **JSON (JavaScript Object Notation)**. Python's standard library provides convenient modules — `csv` and `json` — to easily read, write, and manipulate these file formats.

6.2.1 Working with CSV Files

The CSV format stores tabular data with each row as a line, and columns separated by commas (or other delimiters).

Reading CSV Files

Full runnable code:

```
import csv

with open("contacts.csv", mode='r', newline='') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)  # Each row is a list of strings
```

Example content of `contacts.csv`:

```
Name,Email,Phone
Alice,alice@example.com,123-456-7890
Bob,bob@example.com,987-654-3210
```

Output:

```
['Name', 'Email', 'Phone']
['Alice', 'alice@example.com', '123-456-7890']
['Bob', 'bob@example.com', '987-654-3210']
```

Writing CSV Files

Full runnable code:

```
import csv

contacts = [
    ["Name", "Email", "Phone"],
    ["Charlie", "charlie@example.com", "555-123-4567"],
    ["Diana", "diana@example.com", "555-987-6543"]
]

with open("new_contacts.csv", mode='w', newline='') as file:
    writer = csv.writer(file)
    writer.writerows(contacts)
```

- The `newline=''` argument ensures correct line endings across platforms.
- `writerows()` writes multiple rows at once.

Using `csv.DictReader` and `csv.DictWriter`

For better readability, you can read and write CSV files as dictionaries:

Full runnable code:

```
import csv

# Reading with DictReader
with open("contacts.csv", mode='r', newline='') as file:
    reader = csv.DictReader(file)
    for row in reader:
        print(row["Name"], row["Email"])

# Writing with DictWriter
with open("output.csv", mode='w', newline='') as file:
    fieldnames = ["Name", "Email", "Phone"]
    writer = csv.DictWriter(file, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({"Name": "Eve", "Email": "eve@example.com", "Phone": "555-000-1111"})
```

6.2.2 Working with JSON Files

JSON is a popular data-interchange format that maps naturally to Python dictionaries and lists.

Reading JSON Files

Full runnable code:

```
import json

with open("config.json", 'r') as file:
    data = json.load(file)

print(data)
```

Example content of `config.json`:

```
{
    "app_name": "MyApp",
    "version": "1.0",
    "features": ["login", "data_export", "analytics"]
}
```

Output:

```
{'app_name': 'MyApp', 'version': '1.0', 'features': ['login', 'data_export', 'analytics']}
```

Writing JSON Files

Full runnable code:

```
import json

config = {
    "app_name": "MyApp",
```

```
    "version": "1.0",
    "features": ["login", "data_export", "analytics"]
}

with open("new_config.json", 'w') as file:
    json.dump(config, file, indent=4)  # indent adds readable formatting
```

6.2.3 Practical Example: Exporting Contact Info

Full runnable code:

```
import csv
import json

contacts = [
    {"name": "Alice", "email": "alice@example.com", "phone": "123-456-7890"},
    {"name": "Bob", "email": "bob@example.com", "phone": "987-654-3210"}
]

# Write contacts to JSON file
with open("contacts.json", 'w') as json_file:
    json.dump(contacts, json_file, indent=4)

# Write contacts to CSV file
with open("contacts.csv", 'w', newline='') as csv_file:
    fieldnames = ["name", "email", "phone"]
    writer = csv.DictWriter(csv_file, fieldnames=fieldnames)
    writer.writeheader()
    writer.writerows(contacts)
```

6.2.4 Summary

Task	Module	Function / Class	Purpose
Read CSV file	`csv`	`csv.reader()`	Read rows as lists
Read CSV as dict	`csv`	`csv.DictReader()`	Read rows as dictionaries
Write CSV file	`csv`	`csv.writer()`, `csv.DictWriter()`	Write rows as lists or dicts
Read JSON file	`json`	`json.load()`	Parse JSON into Python objects
Write JSON file	`json`	`json.dump()`	Serialize Python objects as JSON

Using the `csv` and `json` modules, you can efficiently exchange data between your Python programs and other applications or storage formats — a fundamental skill for real-world programming.

6.3 Using Context Managers (`with` Statement)

When working with files, it's important to **ensure files are properly closed** after use, even if errors occur. Python's **context managers**, accessed through the `with` statement, provide a simple and reliable way to manage this.

6.3.1 Why Use Context Managers?

- Automatically **handle opening and closing** of files.
- Prevent **resource leaks** that happen if files are left open.
- Make your code **cleaner and easier to read**.
- Ensure files are closed **even if an error occurs** inside the block.

6.3.2 Traditional File Handling vs. Context Manager

Without `with` (using `open()` and `close()`):

Full runnable code:

```
try:
    file = open("data.txt", 'r')
    for line in file:
        print(line.strip())
finally:
    file.close()
```

- Requires explicit `close()` calls.
- Need `try-finally` to ensure closing if exceptions happen.
- More boilerplate and error-prone.

With `with` Statement (Using Context Manager):

Full runnable code:

```
with open("data.txt", 'r') as file:
    for line in file:
        print(line.strip())
```

- Python automatically calls `file.close()` after the block.
- Cleaner, simpler, and safer code.

6.3.3 Writing to a File Using `with`

Full runnable code:

```
with open("output.txt", 'w') as file:
    for i in range(5):
        file.write(f"Line {i + 1}\n")
```

- The file is automatically closed once the block ends.
- No need for explicit `close()`.

6.3.4 Why Prefer `with`?

- Avoids **forgotten `close()` calls**, which can cause file locks or resource exhaustion.
- Handles exceptions **gracefully** without leaking resources.
- Makes your code easier to maintain and understand.

6.3.5 Summary

Traditional File Handling	Using `with` Statement
Manually open and close files	Use `with` to auto-open and auto-close
Must use `try-finally` for safety	Cleaner syntax, fewer lines of code
Risk of forgetting to close file	No risk of forgetting to close file

6.3.6 Final Note

From now on, always prefer using the `with` statement when working with files in Python. It's a best practice that ensures your program manages system resources safely and effectively.

Chapter 7.

Error Handling and Exceptions

1. Understanding Exceptions and Errors
2. Using `try`, `except`, `else`, and `finally`
3. Creating Custom Exception Classes

7 Error Handling and Exceptions

7.1 Understanding Exceptions and Errors

In programming, errors are inevitable. Python provides a powerful mechanism called **exceptions** to handle these errors gracefully and keep your programs running smoothly.

7.1.1 What Are Errors and Exceptions?

- **Errors** are problems in your code that prevent it from running correctly.
- **Exceptions** are special events that occur during program execution when an error is detected.
- Exceptions can be **caught and handled** to avoid abrupt program termination.

7.1.2 Types of Errors

Syntax Errors

- Occur when Python **cannot parse your code** because it violates the language rules.
- These errors are detected **before the program runs**.
- Examples: missing colon, unmatched parentheses, incorrect indentation.

Full runnable code:

```
# SyntaxError example
if 5 > 3
    print("Five is greater than three")
```

Running this code results in:

```
SyntaxError: invalid syntax
```

Runtime Exceptions

- Occur **while the program is running**.
- Caused by illegal operations like dividing by zero or accessing a non-existent index.
- If not handled, they **stop program execution**.

Full runnable code:

```
# Runtime Exception example
number = int("abc")  # Raises ValueError
```

Output:

```
ValueError: invalid literal for int() with base 10: 'abc'
```

7.1.3 Python Exception Hierarchy

- All exceptions inherit from the base class `BaseException`.
- The most commonly used exceptions inherit from `Exception`.
- This hierarchy lets you catch specific exceptions or handle all exceptions in general.

```
BaseException
 +-- Exception
      +-- ValueError
      +-- TypeError
      +-- ZeroDivisionError
      +-- ... (many others)
```

7.1.4 Common Built-in Exceptions

Exception Name	When It Occurs	Example
`ValueError`	Invalid value or type conversion fails	`int("hello")`
`TypeError`	Unsupported operation on incompatible types	`"2" + 3` (string + integer)
`ZeroDivisionError`	Dividing a number by zero	`10 / 0`

7.1.5 Illustrative Examples

ValueError

Full runnable code:

```python
user_input = "abc"
try:
    number = int(user_input)
except ValueError:
    print("Cannot convert input to a number.")
```

Output:

```
Cannot convert input to a number.
```

TypeError

```
result = "5" + 3  # Causes TypeError
```

Output:

```
TypeError: can only concatenate str (not "int") to str
```

ZeroDivisionError

```
x = 10
y = 0
print(x / y)  # Causes ZeroDivisionError
```

Output:

```
ZeroDivisionError: division by zero
```

7.1.6 What Happens When Exceptions Are Unhandled?

If your program encounters an exception and it is not caught using error handling constructs, the program stops immediately and Python prints a traceback showing the error type and location.

Example:

Full runnable code:

```
print(10 / 0)
print("This line will not run.")
```

Output:

```
ZeroDivisionError: division by zero
```

The second print statement will **not execute**.

7.1.7 Summary

- **Syntax errors** prevent your program from running and must be fixed before execution.
- **Runtime exceptions** occur during execution and can be handled to avoid crashes.
- Python's exception hierarchy allows flexible and specific error handling.
- Common exceptions like `ValueError`, `TypeError`, and `ZeroDivisionError` help identify typical programming mistakes.

Understanding exceptions is the first step toward writing robust Python programs that

gracefully handle unexpected situations.

7.2 Using `try`, `except`, `else`, and `finally`

Python provides a powerful structure to **handle errors and exceptions gracefully** using `try`, `except`, `else`, and `finally` blocks. This allows your program to respond to errors without crashing and to perform cleanup actions.

7.2.1 The Basic Structure

```
try:
    # Code that might raise an exception
except SomeException:
    # Code to handle the exception
else:
    # Code that runs if no exceptions occur
finally:
    # Code that always runs, no matter what
```

7.2.2 Step-by-Step Explanation

`try` Block

Put code that **might raise an exception** inside the `try` block.

`except` Block

Catches and handles specified exceptions. You can have multiple `except` blocks to handle different exception types.

`else` Block

Runs only if the `try` block **did not raise any exceptions**. Useful for code that should run only when everything went well.

`finally` Block

Runs **no matter what** — whether an exception was raised or not. Commonly used for cleanup actions like closing files or releasing resources.

7.2.3 Example 1: Handling Multiple Exceptions

Full runnable code:

```
try:
    x = int(input("Enter a number: "))
    y = 10 / x
except ValueError:
    print("Oops! That was not a valid number.")
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
else:
    print(f"Result is {y}")
```

How this works:

- If the input is not a number, `ValueError` is caught.
- If the user enters `0`, division raises `ZeroDivisionError`.
- If no exceptions occur, the result is printed.

7.2.4 Example 2: Using `finally` for Cleanup

Full runnable code:

```
try:
    file = open("data.txt", "r")
    content = file.read()
    print(content)
except FileNotFoundError:
    print("The file does not exist.")
finally:
    print("Closing the file.")
    try:
        file.close()
    except NameError:
        # file was never opened
        pass
```

Explanation:

- The file is opened and read inside `try`.
- If the file doesn't exist, `FileNotFoundError` is handled.
- The `finally` block **always runs**, ensuring the file is closed if it was opened.
- The nested `try-except` inside `finally` avoids errors if `file` was never created.

7.2.5 Simplified Version Using `with` (Best Practice)

Using context managers with `with` makes file handling easier and safer (covered in Chapter 6):

Full runnable code:

```
try:
    with open("data.txt", "r") as file:
        content = file.read()
        print(content)
except FileNotFoundError:
    print("The file does not exist.")
finally:
    print("Finished attempting to read the file.")
```

7.2.6 Summary

Keyword	Purpose
`try`	Wrap code that might raise exceptions
`except`	Catch and handle specific exceptions
`else`	Run if no exceptions were raised
`finally`	Run code no matter what (cleanup)

7.2.7 Key Tips

- Use multiple `except` blocks to handle different errors explicitly.
- Use `else` for code that should only run when no exceptions happen.
- Always use `finally` to release resources or perform cleanup.
- Prefer `with` statements for file handling to avoid manual `finally` cleanup.

7.3 Creating Custom Exception Classes

Sometimes, built-in Python exceptions are not enough to clearly communicate specific errors in your program. In these cases, you can **create your own custom exception classes** to handle unique situations and improve code readability and maintainability.

7.3.1 Why Create Custom Exceptions?

- To **represent specific error conditions** in your application or domain.
- To **provide clear, meaningful error messages** that help with debugging.
- To **differentiate your errors** from built-in exceptions or other types of exceptions.
- To allow users of your code to **handle your errors explicitly**.

7.3.2 How to Create a Custom Exception

You create a custom exception by subclassing Python's built-in `Exception` class (or another relevant built-in exception).

Basic Syntax

```
class MyCustomError(Exception):
    pass
```

You can also override the initializer to accept custom messages.

7.3.3 Example: Custom Exception for Bank Account Withdrawal

Let's define a custom exception that is raised when a withdrawal exceeds the available balance.

```
class InsufficientFundsError(Exception):
    def __init__(self, balance, amount):
        super().__init__(f"Insufficient funds: Tried to withdraw ${amount}, but only ${balance} availabl
        self.balance = balance
        self.amount = amount
```

7.3.4 Using the Custom Exception

Full runnable code:

```
class InsufficientFundsError(Exception):
    def __init__(self, balance, amount):
        super().__init__(f"Insufficient funds: Tried to withdraw ${amount}, but only ${balance} availabl
        self.balance = balance
        self.amount = amount

class BankAccount:
    def __init__(self, balance=0):
        self.balance = balance
```

```
    def withdraw(self, amount):
        if amount > self.balance:
            raise InsufficientFundsError(self.balance, amount)
        self.balance -= amount
        print(f"Withdrawal successful. New balance: ${self.balance}")

# Example usage:
account = BankAccount(100)

try:
    account.withdraw(150)
except InsufficientFundsError as e:
    print(f"Error: {e}")
```

Output:

```
Error: Insufficient funds: Tried to withdraw $150, but only $100 available.
```

7.3.5 Key Points

- **Subclass `Exception`** to create your custom error.
- Use clear, descriptive **class names** like `InsufficientFundsError`.
- Provide **informative error messages** by overriding the constructor.
- Raise your custom exception using `raise` when the specific error condition occurs.
- Handle the custom exception with a regular `except` block.

7.3.6 Summary

Custom exceptions improve your program's clarity by representing domain-specific errors explicitly. They make debugging easier and allow users of your code to respond appropriately to particular error situations.

Chapter 8.

Object-Oriented Programming (OOP)

1. Classes and Objects
2. Attributes and Methods
3. Inheritance and Polymorphism
4. Encapsulation and Special Methods (`__init__`, `__str__`, etc.)

8 Object-Oriented Programming (OOP)

8.1 Classes and Objects

Object-Oriented Programming (OOP) allows you to model real-world entities as **classes** and **objects** in your code. This approach helps organize complex programs by bundling data and behavior together.

8.1.1 What is a Class?

A **class** is like a blueprint or template that defines the structure and behavior that objects created from the class will have. It describes **what an object will be** and **what it can do**.

8.1.2 What is an Object?

An **object** is an **instance** of a class. When you create an object from a class, you are creating a specific entity based on the class blueprint.

For example:

- Class: `Car` (the blueprint)
- Object: `my_car` (a specific car with its own properties)

8.1.3 Defining a Class: The `Car` Example

Here's how you define a simple `Car` class in Python:

```
class Car:
    pass  # Empty class for now
```

8.1.4 Creating Objects (Instances)

You create objects by **calling the class as if it were a function**:

```
my_car = Car()
your_car = Car()
```

Now, `my_car` and `your_car` are two different instances of the `Car` class.

8.1.5 Adding Attributes and Behavior

Let's improve the `Car` class to include attributes like `make`, `model`, and `year`:

Full runnable code:

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

# Creating an object with attributes
my_car = Car("Toyota", "Corolla", 2020)
print(my_car.make)   # Output: Toyota
print(my_car.model)  # Output: Corolla
print(my_car.year)   # Output: 2020
```

- The `__init__` method is a special method called a **constructor** that runs when an object is created.
- `self` represents the current object and is used to access attributes.

8.1.6 Class vs. Object Analogy

Concept	Real-World Example	Programming Example
Class	Blueprint of a car	`Car` class definition
Object	Your actual car parked outside	`my_car` instance of `Car`

8.1.7 Exercise: Model a Simple Entity

Try creating a class called `Book` with the following attributes:

- `title`
- `author`
- `year_published`

Create at least two objects of `Book` and print their attributes.

8.1.8 Summary

- **Classes** define blueprints for creating **objects**.

- **Objects** are individual instances with their own data.
- The `__init__` method initializes object attributes.
- Using classes helps you model real-world things logically and clearly.

8.2 Attributes and Methods

8.3 Attributes and Methods

In object-oriented programming, **attributes** and **methods** are the core building blocks of classes and objects. Understanding their differences and uses is essential to model real-world entities effectively.

8.3.1 What Are Attributes?

Attributes store **data** about an object. They describe the object's characteristics.

Types of Attributes

1. **Instance Attributes**
    - Defined inside the `__init__` method (constructor).
    - Unique to each object (instance).
    - Accessed with `self.attribute_name`.
2. **Class Attributes**
    - Defined directly inside the class, outside any methods.
    - Shared by all instances of the class.
    - Useful for properties common to all objects.

Example of Instance vs Class Attributes

Full runnable code:

```
class Person:
    species = "Homo sapiens"  # Class attribute

    def __init__(self, name, age):
        self.name = name      # Instance attribute
        self.age = age        # Instance attribute

# Create two objects
person1 = Person("Alice", 30)
person2 = Person("Bob", 25)

print(person1.name)     # Output: Alice
```

```
print(person2.age)      # Output: 25
print(person1.species)  # Output: Homo sapiens
print(person2.species)  # Output: Homo sapiens
```

- `species` is a **class attribute** shared by all `Person` objects.
- `name` and `age` are **instance attributes**, specific to each person.

8.3.2 What Are Methods?

Methods are **functions defined inside a class** that operate on objects. They describe the **behavior** or actions that objects can perform.

- The first parameter is always `self`, which refers to the specific object calling the method.
- Methods can access or modify the object's attributes using `self`.

Defining Instance Methods

Here is a class with instance attributes and methods:

Full runnable code:

```
class Person:
    species = "Homo sapiens"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        print(f"Hello, my name is {self.name}.")

    def update_age(self, new_age):
        self.age = new_age
        print(f"{self.name}'s age has been updated to {self.age}.")

# Using Attributes and Methods

person = Person("Charlie", 40)
person.greet()           # Output: Hello, my name is Charlie.
person.update_age(41)    # Output: Charlie's age has been updated to 41.
print(person.age)        # Output: 41
```

8.3.3 Summary

Term	Description	Defined Where	Access Syntax
Instance Attribute	Data unique to each object	Inside `__init__` via `self`	`self.attribute_name`
Class Attribute	Shared data among all instances	Directly in the class body	`ClassName.attribute` or `self.attribute`
Instance Method	Functions acting on object data	Inside class with `self` parameter	`self.method_name()`

8.3.4 Exercise

Try creating a `Dog` class with:

- Instance attributes: `name` and `breed`
- A class attribute: `species` with the value `"Canis familiaris"`
- Methods:
  - `bark()` that prints a message including the dog's name
  - `change_breed(new_breed)` that updates the dog's breed and prints a confirmation

Create an object of `Dog` and test its attributes and methods.

8.4 Inheritance and Polymorphism

Inheritance and polymorphism are two fundamental principles of Object-Oriented Programming (OOP) that help you write reusable and scalable code.

8.4.1 What is Inheritance?

Inheritance allows a new class (called a **derived** or **child class**) to **inherit attributes and methods** from an existing class (called the **base** or **parent class**). This helps you avoid code duplication by reusing common behavior.

Example: `Animal` Base Class and Derived Classes

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

    def speak(self):
```

```
        print(f"{self.name} makes a sound.")
```

Now, create two subclasses, `Dog` and `Cat`, that inherit from `Animal`:

```
class Dog(Animal):
    def speak(self):
        print(f"{self.name} says Woof!")

class Cat(Animal):
    def speak(self):
        print(f"{self.name} says Meow!")
```

Creating Objects and Using Inherited Methods

```
dog = Dog("Buddy")
cat = Cat("Whiskers")

dog.speak()  # Output: Buddy says Woof!
cat.speak()  # Output: Whiskers says Meow!
```

- Both `Dog` and `Cat` inherit the `__init__` method from `Animal`.
- They **override** the `speak()` method to provide their own behavior.

8.4.2 What is Polymorphism?

Polymorphism means "many forms." In OOP, it refers to the ability to use a **single interface to represent different underlying forms (data types)**.

In our example, you can treat `Dog` and `Cat` objects as `Animal` objects and call `speak()` on them without worrying about their exact type.

Polymorphism in Action

```
animals = [Dog("Buddy"), Cat("Whiskers"), Animal("Generic Animal")]

for animal in animals:
    animal.speak()
```

Output:

```
Buddy says Woof!
Whiskers says Meow!
Generic Animal makes a sound.
```

Here:

- The `speak()` method behaves differently depending on the actual object type.
- This lets you write flexible code that works with any subclass of `Animal`.

8.4.3 Advantages of Inheritance and Polymorphism

- **Reusability:** Share common code in a base class and extend or customize it in subclasses.
- **Scalability:** Easily add new subclasses with specific behavior without changing existing code.
- **Maintainability:** Keep your code organized and easier to debug.
- **Flexibility:** Write functions and loops that operate on base class references but automatically use subclass behavior.

8.4.4 Summary

Concept	Description	Example
Inheritance	Child classes inherit attributes and methods from a parent class	`Dog` and `Cat` inherit from `Animal`
Polymorphism	Different classes implement the same method differently, allowing uniform interface usage	Calling `speak()` on different animals

8.4.5 Exercise

1. Create a base class `Shape` with a method `area()` that prints `"Area not defined."`.
2. Create subclasses `Rectangle` and `Circle` that override the `area()` method to calculate and print their area.
3. Instantiate objects of each class and call `area()` on them in a loop.

8.5 Encapsulation and Special Methods (`__init__`, `__str__`, etc.)

In this section, we'll explore **encapsulation**, a core concept in Object-Oriented Programming that helps protect and organize data within classes. Then, we'll dive into **special methods** (also called magic methods) that let you customize how your objects behave, especially during creation and printing.

8.5.1 What is Encapsulation?

Encapsulation is the practice of **restricting access to certain components** of an object to protect its internal state and prevent unintended interference.

Python uses **naming conventions** to indicate how attributes should be accessed:

Naming Convention	Meaning	Usage Example
`public`	Accessible from anywhere	`self.name`
`_protected`	Intended for internal use (convention only)	`_balance`
`__private`	Name mangling for private use	`__password`

Example: Protected and Private Attributes

Full runnable code:

```
class User:
    def __init__(self, username, password):
        self.username = username        # Public attribute
        self._email = None              # Protected attribute (convention)
        self.__password = password      # Private attribute (name mangling)

    def set_email(self, email):
        self._email = email

    def get_password(self):
        return self.__password  # Accessing private attribute within class

user = User("alice", "secret123")
print(user.username)       # Accessible
print(user._email)         # Accessible, but should be treated as protected
# print(user.__password)   # Error: AttributeError

# Accessing mangled name (not recommended)
print(user._User__password)  # Outputs: secret123
```

- Single underscore (`_`) signals to programmers: "This is internal, don't touch it."
- Double underscore (`__`) triggers **name mangling**: Python internally changes the attribute name to prevent accidental access.

8.5.2 What Are Special (Magic) Methods?

Special methods allow you to **define how your objects behave with built-in Python functions and operators**. They are always surrounded by double underscores (`__methodname__`).

Commonly Used Special Methods

Method	Purpose	Example Use
`__init__`	Constructor: initializes a new object	Set attributes when an object is created
`__str__`	Defines the **informal** string representation of an object (user-friendly)	Used by `print()` or `str()`
`__repr__`	Defines the **official** string representation (debugging)	Used in interactive interpreter or `repr()`

The `__init__` Method

Runs automatically when you create an object. It initializes the object's attributes.

Full runnable code:

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

book = Book("1984", "George Orwell")
print(book.title)   # Output: 1984
```

The `__str__` Method

Defines what is shown when you print an object or convert it to a string.

Full runnable code:

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

    def __str__(self):
        return f"'{self.title}' by {self.author}"

book = Book("1984", "George Orwell")
print(book)  # Output: '1984' by George Orwell
```

Without `__str__`, printing `book` would show something like `<__main__.Book object at 0x7f8b20>`.

The `__repr__` Method

Returns a string representation meant for developers to recreate the object, usually more detailed.

Full runnable code:

```
class Book:
    def __init__(self, title, author):
```

```
        self.title = title
        self.author = author

    def __repr__(self):
        return f"Book(title='{self.title}', author='{self.author}')"

book = Book("1984", "George Orwell")
print(repr(book))
# Output: Book(title='1984', author='George Orwell')
```

Using Both `__str__` and `__repr__`

You can define both to serve different purposes:

Full runnable code:

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

    def __str__(self):
        return f"'{self.title}' by {self.author}"

    def __repr__(self):
        return f"Book(title='{self.title}', author='{self.author}')"

book = Book("1984", "George Orwell")
print(book)       # Uses __str__
print(repr(book)) # Uses __repr__
```

8.5.3 Summary

- **Encapsulation** protects object data by using naming conventions (`_protected`, `__private`).
- Special methods like `__init__` initialize objects.
- `__str__` provides a user-friendly string version of objects.
- `__repr__` gives a developer-friendly representation useful for debugging.
- Defining these methods improves how your objects integrate with Python's syntax and built-ins.

8.5.4 Exercise

Create a class `Student` with:

- Attributes: `name`, `grade`

- Use `_grade` as a protected attribute
- Implement `__init__`, `__str__`, and `__repr__` methods
- Test printing a `Student` object to see the difference between `__str__` and `__repr__`

Chapter 9.

Working with Libraries and Packages

1. Installing and Managing Packages with `pip`
2. Using Popular Libraries (e.g., `math`, `datetime`, `random`)
3. Creating and Distributing Your Own Modules

9 Working with Libraries and Packages

9.1 Installing and Managing Packages with `pip`

Python's strength comes from its vast ecosystem of **third-party packages** that you can easily add to your projects using the package manager called **`pip`**. This section will guide you through installing, upgrading, and removing packages using `pip`, and introduce you to managing project dependencies with `requirements.txt` and virtual environments.

9.1.1 What is `pip`?

`pip` is Python's **package installer**. It allows you to download and manage libraries from the Python Package Index (PyPI) — a huge repository of open-source Python packages.

9.1.2 Installing Packages

To install a package, use:

```
pip install package-name
```

Example:

```
pip install requests
```

This command downloads and installs the popular `requests` library, which makes working with HTTP requests simple.

9.1.3 Listing Installed Packages

To see all packages installed in your current environment, run:

```
pip list
```

This will show the package name and its version.

9.1.4 Upgrading Packages

To upgrade an existing package to the latest version, use:

```
pip install --upgrade package-name
```

Example:

```
pip install --upgrade requests
```

9.1.5 Uninstalling Packages

To remove a package you no longer need, run:

```
pip uninstall package-name
```

Example:

```
pip uninstall requests
```

You will be prompted to confirm the uninstallation.

9.1.6 Managing Project Dependencies with `requirements.txt`

When working on projects, you often need to share or recreate your environment with specific package versions. A **`requirements.txt`** file lists all required packages.

Creating a `requirements.txt`

You can generate this file automatically with:

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

This captures the current environment's packages and versions.

Installing from `requirements.txt`

To install all packages listed in this file on another machine or environment, run:

```
pip install -r requirements.txt
```

This ensures consistent environments across different setups.

9.1.7 Using Virtual Environments (`venv`)

Installing packages globally can lead to version conflicts between projects. To isolate dependencies, use **virtual environments**:

Creating a Virtual Environment

```
python -m venv myenv
```

This creates a folder `myenv` containing an isolated Python environment.

Activating the Virtual Environment

- On **Windows**:
  ```
  myenv\Scripts\activate
  ```
- On **macOS/Linux**:
  ```
  source myenv/bin/activate
  ```

You'll see your terminal prompt change, indicating you are working inside the virtual environment.

Installing Packages Inside the Virtual Environment

Once activated, any `pip install` commands will install packages only in this isolated environment.

Deactivating the Virtual Environment

Simply run:

```
deactivate
```

to exit the virtual environment.

9.1.8 Summary

Task	Command Example
Install a package	`pip install package-name`
List installed packages	`pip list`
Upgrade a package	`pip install --upgrade package-name`
Uninstall a package	`pip uninstall package-name`
Save dependencies	`pip freeze > requirements.txt`
Install from requirements	`pip install -r requirements.txt`
Create virtual environment	`python -m venv myenv`
Activate virtual environment	`source myenv/bin/activate` (macOS/Linux) or `myenv\Scripts\activate` (Windows)
Deactivate virtual environment	`deactivate`

9.2 Using Popular Libraries (e.g., `math`, `datetime`, `random`)

Python comes with many **built-in standard libraries** that provide ready-made tools to make programming easier and more powerful. Here, we'll explore three commonly used libraries: **`math`**, **`datetime`**, and **`random`**, with simple examples you can try yourself.

9.2.1 The `math` Library

The `math` module offers mathematical functions and constants like square roots, powers, trigonometry, and more.

Key Functions and Usage

Full runnable code:

```
import math

# Calculate the square root of 16
print(math.sqrt(16))  # Output: 4.0

# Calculate the cosine of 0 radians
print(math.cos(0))    # Output: 1.0

# Find the value of pi
print(math.pi)        # Output: 3.141592653589793

# Round down to the nearest integer
print(math.floor(3.7))  # Output: 3
```

9.2.2 The `datetime` Library

The `datetime` module helps you work with dates and times — ideal for tasks like logging, scheduling, or calculating durations.

Key Functions and Usage

Full runnable code:

```
import datetime

# Get the current date and time
now = datetime.datetime.now()
print("Now:", now)  # Example output: 2025-06-26 21:15:43.123456

# Extract just the date
today = datetime.date.today()
print("Today:", today)  # Example output: 2025-06-26
```

```
# Create a specific date
birthday = datetime.date(1990, 12, 15)
print("Birthday:", birthday)

# Calculate the difference between two dates
days_until_birthday = (birthday - today).days
print("Days until birthday:", days_until_birthday)
```

9.2.3 The `random` Library

The `random` module is great for generating random numbers, selections, and simulating chance.

Key Functions and Usage

Full runnable code:

```
import random

# Generate a random integer between 1 and 10 (inclusive)
rand_num = random.randint(1, 10)
print("Random number:", rand_num)

# Choose a random item from a list
colors = ['red', 'green', 'blue']
random_color = random.choice(colors)
print("Random color:", random_color)

# Shuffle a list in place
random.shuffle(colors)
print("Shuffled colors:", colors)
```

9.2.4 Practical Tasks to Try

- **Using `math`**: Calculate the length of the hypotenuse of a right triangle using `math.sqrt()` and the Pythagorean theorem.
- **Using `datetime`**: Write a program to calculate how many days are left until your next birthday.
- **Using `random`**: Create a simple random password generator by picking random letters and digits.

9.2.5 Where to Learn More

These libraries contain many more functions and options. Explore the official Python documentation to deepen your understanding:

- `math` module
- `datetime` module
- `random` module

Ready to explore more? Next, we'll learn how to **create and distribute your own modules** to share reusable code across your projects!

9.3 Creating and Distributing Your Own Modules

One of Python's great strengths is how easily you can **organize code into reusable modules** and even share those modules with others by creating packages. This section guides you through creating a simple module, importing it, and the basics of packaging it for distribution.

9.3.1 Creating a Simple Module

A **module** is just a Python file (`.py`) containing functions, classes, or variables that you want to reuse.

Step 1: Create `utilities.py`

Create a file named `utilities.py` and add some useful functions:

```
# utilities.py

def greet(name):
    return f"Hello, {name}!"

def square(number):
    return number * number

if __name__ == "__main__":
    # This block runs only when you execute utilities.py directly
    print(greet("World"))
    print(square(5))
```

9.3.2 Importing Your Module in Another Script

Create a new file called `main.py` in the same directory:

```
# main.py

import utilities

print(utilities.greet("Alice"))   # Output: Hello, Alice!
print(utilities.square(7))        # Output: 49
```

Run `main.py` and see your module in action.

9.3.3 Understanding `if __name__ "__main__"`

- This special condition checks if the file is **run directly** versus **imported** as a module.
- Code inside this block only runs when executing the file itself, not when imported.
- It's useful for testing or demo code within your module.

Example:

```
if __name__ == "__main__":
    # Run some tests or examples
    print("Running tests...")
```

9.3.4 Packaging Your Module for Distribution

If you need to share your module with others or install it on different machines, you package it as a **Python package**.

Basic Steps:

1. **Organize your code** Put your module(s) inside a directory named after your package, for example:

   ```
   mypackage/
       __init__.py
       utilities.py
   ```

   The `__init__.py` file can be empty but marks the directory as a package.

2. **Create a `setup.py` file** This file tells Python how to install your package.

   ```
   # setup.py
   from setuptools import setup, find_packages

   setup(
       name="mypackage",
       version="0.1",
   ```

```
    packages=find_packages(),
    author="Your Name",
    description="A simple utilities package",
    python_requires='>=3.6',
)
```

3. **Build and install your package locally** In the terminal, run:
```
pip install .
```
inside the directory containing `setup.py`. This installs your package into your current environment.

4. **Distribute your package** (optional) To share your package publicly, you can upload it to PyPI using tools like `twine`. This is more advanced and can be explored later.

9.3.5 Additional Resources

- Python Packaging User Guide
- setuptools documentation
- PyPI — The Python Package Index

9.3.6 Summary

- **Modules** help organize and reuse code.
- Use `import` to access functions or classes from other modules.
- `if __name__ == "__main__"` lets you run test code only when the module is executed directly.
- Packaging your code lets you distribute and reuse it easily.
- Basic packaging uses `setup.py` with `setuptools`.
- Explore packaging tools further as you advance.

Chapter 10.

Advanced Functions

1. Lambda Functions
2. Decorators
3. Generators and Iterators
4. Closures and Higher-Order Functions

10 Advanced Functions

10.1 Lambda Functions

In Python, **lambda functions** are small, anonymous functions defined with the `lambda` keyword. They provide a concise way to write simple functions without needing to formally define a function using `def`.

10.1.1 What Are Lambda Functions?

- Lambda functions are **anonymous**, meaning they don't have a name.
- They are designed for **short, simple operations**.
- Unlike regular functions defined with `def`, lambdas consist of a **single expression only**.
- The result of that expression is implicitly returned.

10.1.2 Lambda Syntax

```
lambda arguments: expression
```

- `arguments` are the inputs.
- `expression` is a single operation or calculation using the arguments.
- The expression's value is automatically returned.

Example: Regular Function vs. Lambda Function

Regular function to add two numbers:

Full runnable code:

```
def add(x, y):
    return x + y

print(add(3, 5))  # Output: 8
```

Equivalent lambda function:

Full runnable code:

```
add = lambda x, y: x + y
print(add(3, 5))  # Output: 8
```

10.1.3 Limitations of Lambda Functions

- **Single expression only:** No multiple statements or complex logic.
- No explicit `return` statement—result comes from the expression itself.
- Limited debugging info because they have no names.

10.1.4 Practical Examples Using Lambdas

Lambda functions become powerful when used with functions like `map()`, `filter()`, and `sorted()` that take other functions as arguments.

Using `map()` with a Lambda

`map()` applies a function to every item in an iterable.

Full runnable code:

```
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x ** 2, numbers))
print(squared)  # Output: [1, 4, 9, 16, 25]
```

Using `filter()` with a Lambda

`filter()` selects items from an iterable for which the function returns `True`.

Full runnable code:

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

Using `sorted()` with a Lambda as Key

You can customize sorting with a lambda function as the `key` parameter.

Full runnable code:

```
words = ['apple', 'banana', 'cherry', 'date']
sorted_by_length = sorted(words, key=lambda w: len(w))
print(sorted_by_length)  # Output: ['date', 'apple', 'banana', 'cherry']
```

10.1.5 When to Use Lambda Functions

- Use lambdas for **simple, short functions** used temporarily.
- They are convenient when you need a quick function but don't want to write a full `def`.

- Avoid lambdas when the logic is complex—use regular functions for readability and maintainability.

10.1.6 Summary

Feature	Lambda Functions	Regular Functions
Syntax	`lambda args: expression`	`def function_name(args):`
Name	Anonymous (usually assigned to variable)	Named
Body	Single expression only	Multiple statements allowed
Return	Expression result (implicit)	Explicit `return` statement
Use case	Simple, inline functions	Complex or reusable functions

10.1.7 Exercise

1. Use a lambda with `map()` to convert a list of temperatures from Celsius to Fahrenheit.
2. Use `filter()` with a lambda to find all strings longer than 5 characters in a list.
3. Sort a list of tuples `(name, age)` by age using `sorted()` and a lambda as the key.

10.2 Decorators

Decorators are a powerful feature in Python that allow you to **modify or enhance the behavior of functions** or methods without changing their actual code. They're commonly used for logging, access control, memoization, timing, and more.

10.2.1 Functions as First-Class Citizens

In Python, functions are **first-class objects** — they can be assigned to variables, passed as arguments, and returned from other functions.

Full runnable code:

```
def greet(name):
    return f"Hello, {name}!"

say_hello = greet
print(say_hello("Alice"))  # Output: Hello, Alice!
```

10.2.2 Nested Functions

You can define a function inside another function:

Full runnable code:

```
def outer():
    def inner():
        print("This is inner")
    inner()

outer()
```

This concept forms the foundation for **decorators**.

10.2.3 What Is a Decorator?

A **decorator** is a function that takes another function as input and returns a modified or enhanced version of it.

Basic Decorator Structure

Full runnable code:

```
def my_decorator(func):
    def wrapper():
        print("Before the function runs")
        func()
        print("After the function runs")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

Output:

```
Before the function runs
Hello!
After the function runs
```

The `@my_decorator` syntax is **shorthand** for:

```
say_hello = my_decorator(say_hello)
```

10.2.4 Writing a Simple Logging Decorator

Let's log whenever a function is called:

Full runnable code:

```
def log_calls(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with {args} {kwargs}")
        return func(*args, **kwargs)
    return wrapper

@log_calls
def add(x, y):
    return x + y

print(add(3, 4))  # Output: Calling add with (3, 4) {}
```

10.2.5 Preserving Function Metadata with `functools.wraps`

Decorators can obscure the original function's metadata (like name and docstring). Use `functools.wraps` to preserve it.

```
import functools

def log_calls(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__}")
        return func(*args, **kwargs)
    return wrapper
```

10.2.6 Decorators with Arguments

To make a decorator configurable, you can use another level of function wrapping:

Full runnable code:

```
def repeat(times):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            for _ in range(times):
                func(*args, **kwargs)
        return wrapper
    return decorator

@repeat(3)
```

```python
def greet():
    print("Hi!")

greet()
```

Output:

```
Hi!
Hi!
Hi!
```

10.2.7 When to Use Decorators

- Logging function calls
- Measuring execution time
- Enforcing access control
- Caching or memoizing results
- Validating arguments

10.2.8 Summary

Concept	Description
Decorator	Function that wraps another function
`@decorator` syntax	Short form for applying a decorator
`functools.wraps`	Preserves function metadata (name, docstring)
With arguments	Return a decorator from another function

10.2.9 Practice Tasks

1. Write a decorator that prints `"Start"` and `"End"` around any function.
2. Create a decorator called `@timer` to measure how long a function takes to run.
3. Use `@repeat(n)` to repeat a function `n` times, just like in the example above.

10.3 Generators and Iterators

In Python, **iterators** and **generators** are tools that let you **loop over sequences of data**—especially when that data is large or potentially infinite. They allow you to write code that is **memory-efficient**, **lazy-evaluated**, and often more readable.

10.3.1 Iterators

An **iterator** is any Python object that implements the following two methods:

- `__iter__()` — returns the iterator object itself.
- `__next__()` — returns the next item in the sequence and raises `StopIteration` when there are no more items.

Creating a Custom Iterator

Full runnable code:

```
class CountToThree:
    def __init__(self):
        self.n = 1

    def __iter__(self):
        return self

    def __next__(self):
        if self.n <= 3:
            result = self.n
            self.n += 1
            return result
        else:
            raise StopIteration

for num in CountToThree():
    print(num)
```

Output:

```
1
2
3
```

This is how Python's `for` loop works behind the scenes!

10.3.2 Generators

Generators are a **simpler way to create iterators** using a special kind of function that yields values one at a time using the `yield` keyword.

Generator Function Example

Full runnable code:

```
def count_to_three():
    yield 1
    yield 2
    yield 3

for num in count_to_three():
    print(num)
```

Generators **automatically implement** the iterator protocol—you don't need to write `__iter__()` or `__next__()` manually.

10.3.3 The `yield` Keyword

- `yield` pauses the function and **saves its state**.
- On the next call to `next()`, the function resumes **right after the last `yield`**.
- This makes generators great for **lazy evaluation**—producing values one at a time only when needed.

10.3.4 Example: Generator for Even Numbers

Full runnable code:

```
def even_numbers(limit):
    n = 0
    while n <= limit:
        if n % 2 == 0:
            yield n
        n += 1

for num in even_numbers(10):
    print(num)
```

Output:

```
0
2
4
```

```
6
8
10
```

10.3.5 Memory Efficiency with Generators

Let's say you're reading a large file. Reading all lines into a list (`.readlines()`) can consume lots of memory. A generator lets you **read one line at a time**.

Example: Reading Lines with a Generator

Full runnable code:

```
def read_large_file(filename):
    with open(filename) as f:
        for line in f:
            yield line.strip()

for line in read_large_file("bigfile.txt"):
    print(line)
```

This approach is **efficient** even for files with millions of lines, because only **one line is in memory at a time**.

10.3.6 Generator Expressions

Just like list comprehensions, Python also supports **generator expressions**:

Full runnable code:

```
squares = (x * x for x in range(5))
for num in squares:
    print(num)
```

This looks like a list comprehension but uses **()instead of[]‘**, and values are generated **on the fly**.

10.3.7 Summary: Iterators vs. Generators

Feature	Iterators	Generators
Creation	Define `__iter__()` and `__next__()`	Use `yield` inside a function
Syntax	Class-based	Function-based
Memory efficiency	May store all items	Lazy-evaluated, one item at a time
Use case	Complex state machines	Sequences, data streams, large files

10.3.8 Practice Challenges

1. Write a generator that yields Fibonacci numbers up to a given limit.
2. Create a generator expression that produces cubes of numbers from 1 to 10.
3. Modify a file reader to skip blank lines using a generator.

Generators are especially useful when dealing with large or infinite data streams. Next, we'll explore **closures and higher-order functions**, which allow you to build even more dynamic and powerful tools.

10.4 Closures and Higher-Order Functions

In Python, functions are powerful—they can be passed around, returned from other functions, and even store private data. Two key features that enable this flexibility are **higher-order functions** and **closures**.

10.4.1 What Are Higher-Order Functions?

A **higher-order function** is a function that does **at least one of the following**:

- Takes another function as an argument
- Returns a function as a result

This enables flexible, abstract, and reusable code.

Example: Passing a Function as an Argument

Full runnable code:

```
def greet(name):
    return f"Hello, {name}!"
```

```
def loud(func):
    def wrapper(name):
        return func(name).upper()
    return wrapper

loud_greet = loud(greet)
print(loud_greet("Alice"))  # Output: HELLO, ALICE!
```

Here, `loud()` is a higher-order function that enhances another function.

10.4.2 What Are Closures?

A **closure** is a function that:

1. Is **defined inside another function**, and
2. **Remembers the values from its enclosing lexical scope** even after the outer function has finished executing.

Closures allow you to **encapsulate behavior and data** without using classes.

Closure Example: Counter Function

Full runnable code:

```
def make_counter():
    count = 0
    def counter():
        nonlocal count
        count += 1
        return count
    return counter

c1 = make_counter()
print(c1())  # Output: 1
print(c1())  # Output: 2

c2 = make_counter()
print(c2())  # Output: 1 (independent counter)
```

- `count` is a variable in the **enclosing scope**.
- The `counter()` function **remembers** and modifies `count`, even though `make_counter()` has already finished executing.

Closure Example: Custom Multiplier

Full runnable code:

```
def make_multiplier(factor):
    def multiply(x):
        return x * factor
```

```
    return multiply

double = make_multiplier(2)
triple = make_multiplier(3)

print(double(5))  # Output: 10
print(triple(5))  # Output: 15
```

Each returned function "remembers" the value of `factor` that was passed into `make_multiplier()`.

10.4.3 Why Use Closures?

- To **encapsulate behavior** (like methods) without classes
- To retain **state** between function calls
- To write **cleaner, modular, and reusable code**

Closures are often used in:

- Decorators
- Factory functions
- Callback functions
- Functional-style programming

10.4.4 Summary

Concept	Description
Higher-order function	Takes or returns another function
Closure	Remembers variables from its enclosing scope
`nonlocal` keyword	Allows modification of a variable from outer scope

10.4.5 Practice Challenges

1. Write a function `make_greeter(name)` that returns a function which greets with that name.
2. Create a closure that maintains a running total (like a bank balance).
3. Implement a `power_of(n)` closure that returns a function to compute `x ** n`.

Closures are a foundational concept that let you build powerful abstractions without using classes. With higher-order functions and closures, you can start thinking of functions as

flexible tools—not just for computation, but for *customizing behavior* and *managing state*.

Chapter 11.

Comprehensions and Expressions

1. Deep Dive into List, Set, and Dictionary Comprehensions
2. Generator Expressions

11 Comprehensions and Expressions

11.1 Deep Dive into List, Set, and Dictionary Comprehensions

Python comprehensions are a concise and expressive way to create collections like **lists**, **sets**, and **dictionaries**. They provide a clean syntax for filtering, transforming, or extracting data—often replacing longer for-loops with single-line expressions.

11.1.1 List Comprehensions

Basic Syntax:

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

Example: Squares of Numbers

Full runnable code:

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

With an `if` Condition

You can filter items using an `if` clause:

Full runnable code:

```
even_squares = [x * x for x in range(10) if x % 2 == 0]
print(even_squares)  # Output: [0, 4, 16, 36, 64]
```

11.1.2 Set Comprehensions

Set comprehensions use curly braces `{}` instead of square brackets and automatically remove duplicates.

Full runnable code:

```
unique_lengths = {len(word) for word in ["hi", "hello", "world", "hi"]}
print(unique_lengths)  # Output: {2, 5}
```

They are great for eliminating duplicates while transforming data.

11.1.3 Dictionary Comprehensions

Basic Syntax:

```
{key_expr: value_expr for item in iterable}
```

Example: Mapping Strings to Lengths

Full runnable code:

```
words = ["apple", "banana", "cherry"]
length_map = {word: len(word) for word in words}
print(length_map)  # Output: {'apple': 5, 'banana': 6, 'cherry': 6}
```

With Filtering

Full runnable code:

```
words = ["apple", "banana", "cherry"]
# Only include words longer than 5 characters
filtered_map = {word: len(word) for word in words if len(word) > 5}
print(filtered_map)  # Output: {'banana': 6, 'cherry': 6}
```

11.1.4 Nested Comprehensions

You can nest comprehensions to handle complex data structures, like lists of lists (e.g., matrices).

Example: Flatten a Matrix

Full runnable code:

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

Read from left to right: for each row, then for each number in that row.

11.1.5 Example: Filtering Dictionary Values

Full runnable code:

```
inventory = {"apple": 10, "banana": 0, "cherry": 12}
available = {item: count for item, count in inventory.items() if count > 0}
print(available)  # Output: {'apple': 10, 'cherry': 12}
```

11.1.6 Comprehensions vs. Traditional Loops

Feature	Comprehension	Traditional Loop
Readability	Compact and expressive	Clear for complex logic
Performance	Slightly faster in many cases	May be slower for large or nested loops
Flexibility	Best for simple transformations	More control over logic

11.1.7 When to Use Comprehensions

YES Use when:

- The transformation is simple and readable
- You're filtering or transforming elements
- You want to reduce code size

Avoid when:

- The logic is too complex to fit comfortably in one line
- It sacrifices clarity for cleverness

11.1.8 Practice Exercises

1. Create a list comprehension to extract vowels from a string.
2. Use a dictionary comprehension to map numbers 1–5 to their squares.
3. Write a set comprehension to find unique lowercase characters from a string.
4. Flatten a 2D list using a nested list comprehension.
5. Filter dictionary items where the value is greater than a threshold.

11.1.9 Summary

Python comprehensions are elegant tools for creating and manipulating collections efficiently. Mastering them boosts both your **productivity** and **code readability**. In the next section, we'll take this concept even further with **generator expressions**—for large or infinite sequences.

11.2 Generator Expressions

Generator expressions are a concise way to create **generators**—special iterators that **produce items lazily**, one at a time, only as needed. They look very similar to list comprehensions, but instead of building the entire list in memory, they yield items on demand, making them **much more memory-efficient**.

11.2.1 Syntax: Generator vs. List Comprehension

Full runnable code:

```
# List comprehension
squares_list = [x * x for x in range(1000)]
print(squares_list)

# Generator expression
squares_gen = (x * x for x in range(1000))
print(squares_gen)
```

The **only difference** is that generator expressions use **parentheses `()`** instead of square brackets `[]`.

11.2.2 When to Use Generator Expressions

Use a generator expression when:

- You're working with **large or infinite datasets**.
- You **don't need to store** all items in memory.
- You're passing values to a **function that consumes an iterable** like `sum()`, `max()`, `any()`, or `all()`.

11.2.3 Basic Example: Sum of Squares

Full runnable code:

```
# Using list comprehension (builds full list in memory)
print(sum([x * x for x in range(1000000)]))

# Using generator expression (more memory-efficient)
print(sum(x * x for x in range(1000000)))
```

The generator version avoids allocating memory for a million items at once.

11.2.4 Practical Use: Any Even Numbers?

Full runnable code:

```
nums = range(1, 100)

# Checks if any number is even
has_even = any(n % 2 == 0 for n in nums)
print(has_even)  # Output: True
```

The generator expression stops **as soon as** it finds an even number. Efficient and fast!

11.2.5 Working with Large Data Sets

Imagine you need to read and count lines from a large file:

Full runnable code:

```
def line_count(filename):
    with open(filename) as f:
        return sum(1 for _ in f)

print(line_count("huge_log_file.txt"))
```

Here, the generator expression `1 for _ in f` reads one line at a time—perfect for huge files.

11.2.6 Infinite Sequences with Generators

You can combine generator expressions with infinite iterators, such as from the `itertools` module:

Full runnable code:

```
import itertools

# Get the first 5 squares from an infinite count
squares = (x * x for x in itertools.count(1))
for i, square in enumerate(squares):
    if i == 5:
        break
    print(square)
```

Output:

```
1
4
```

```
9
16
25
```

Even though `itertools.count()` never ends, the generator only computes what you ask for.

11.2.7 Comparing Memory Usage

Let's compare list vs. generator in terms of memory:

Full runnable code:

```
import sys

list_comp = [x for x in range(1000000)]
gen_expr = (x for x in range(1000000))

print(sys.getsizeof(list_comp))  # Larger memory size
print(sys.getsizeof(gen_expr))   # Much smaller size
```

> The generator typically uses only a few dozen bytes, while the list might use tens of megabytes.

11.2.8 Summary: List vs. Generator Expressions

Feature	List Comprehension	Generator Expression
Syntax	`[x for x in iterable]`	`(x for x in iterable)`
Memory Usage	Loads entire list in memory	Yields one item at a time
Speed	Faster for small data	Better for large or infinite data
Output Type	List	Generator object

11.2.9 Practice Tasks

1. Use a generator expression to count how many numbers between 1 and 10000 are divisible by 7.
2. Write a function that checks if any word in a list starts with a capital letter using a generator.
3. Replace a list comprehension in an existing program with a generator expression and compare performance.

Generator expressions offer an elegant and efficient way to process data streams in Python. Mastering them lets you write **cleaner**, **faster**, and **more scalable** code—especially for data-heavy applications.

Chapter 12.

Concurrency and Parallelism

1. Threading Basics
2. Multiprocessing
3. Asynchronous Programming with `async` and `await`

12 Concurrency and Parallelism

12.1 Threading Basics

Concurrency allows your program to handle multiple tasks seemingly at the same time. **Threading** is one of Python's tools for concurrency, ideal for **I/O-bound tasks** such as reading from disk, making network requests, or waiting for user input.

12.1.1 What Is Threading?

Threading involves running multiple threads (lightweight processes) within a single program. Each thread shares the same memory space but executes independently.

- YES **Best for**: I/O-bound tasks (e.g., downloading files, reading data from sockets)
- NO **Not ideal for**: CPU-bound tasks (use `multiprocessing` instead)

12.1.2 Pythons `threading` Module

Python provides the built-in `threading` module to manage threads.

12.1.3 Creating and Running Threads

Example 1: Basic Thread Creation

Full runnable code:

```
import threading

def greet():
    print("Hello from a thread!")

# Create a thread
t = threading.Thread(target=greet)

# Start the thread
t.start()

# Wait for the thread to finish
t.join()

print("Main thread finished.")
```

Output:

```
Hello from a thread!
Main thread finished.
```

12.1.4 Starting Multiple Threads

Full runnable code:

```
import threading
import time

def countdown(name):
    for i in range(3, 0, -1):
        print(f"{name}: {i}")
        time.sleep(1)

t1 = threading.Thread(target=countdown, args=("Thread-1",))
t2 = threading.Thread(target=countdown, args=("Thread-2",))

t1.start()
t2.start()

t1.join()
t2.join()

print("All threads finished.")
```

Each thread runs independently, printing its countdown in parallel.

12.1.5 Race Conditions

When two threads access **shared data** simultaneously without coordination, it may lead to a **race condition**, causing unpredictable results.

Example of a Race Condition

Full runnable code:

```
counter = 0

def increment():
    global counter
    for _ in range(100000):
        counter += 1

threads = []
for _ in range(2):
```

```
    t = threading.Thread(target=increment)
    threads.append(t)
    t.start()

for t in threads:
    t.join()

print(f"Final counter: {counter}")
```

Output may vary and likely will not be `200000` due to concurrent writes.

12.1.6 Thread Safety with `Lock`

To prevent race conditions, use a `Lock` to control access to shared resources.

Full runnable code:

```
import threading

counter = 0
lock = threading.Lock()

def safe_increment():
    global counter
    for _ in range(100000):
        with lock:
            counter += 1

threads = []
for _ in range(2):
    t = threading.Thread(target=safe_increment)
    threads.append(t)
    t.start()

for t in threads:
    t.join()

print(f"Final counter with lock: {counter}")
```

Now the counter will always be `200000`.

12.1.7 Summary

Concept	Explanation
Thread	A lightweight execution unit that runs in a program
`Thread(target=...)`	Creates a new thread running a function

Concept	Explanation
`.start()`	Begins thread execution
`.join()`	Waits for thread to complete
`Lock()`	Prevents multiple threads from accessing shared data simultaneously

12.1.8 Best Practices

- Use threads for I/O-bound operations, not CPU-heavy computation.
- Always protect shared resources with locks to avoid race conditions.
- Keep thread logic simple to avoid deadlocks and bugs.

12.2 Multiprocessing

When writing Python programs that need to perform **CPU-intensive** tasks—such as heavy computations, data processing, or image manipulation—**threading** may not be efficient due to the **Global Interpreter Lock (GIL)**.

12.2.1 Understanding the GIL

The **Global Interpreter Lock** is a mechanism in the CPython interpreter that ensures only **one thread executes Python bytecode at a time**, even on multi-core systems. This makes **multithreading ineffective for CPU-bound tasks**, because threads can't run in true parallel.

Solution: `multiprocessing`

Python's `multiprocessing` module allows you to **bypass the GIL** by using **separate processes**. Each process runs in its own Python interpreter and memory space, enabling **true parallel execution** across multiple CPU cores.

12.2.2 Basic Example: Creating a Process

Full runnable code:

```
from multiprocessing import Process
```

```
def say_hello():
    print("Hello from a process!")

if __name__ == '__main__':
    p = Process(target=say_hello)
    p.start()
    p.join()
    print("Main process finished.")
```

Output:

```
Hello from a process!
Main process finished.
```

12.2.3 Passing Arguments to a Process

Full runnable code:

```
from multiprocessing import Process

def greet(name):
    print(f"Hello, {name}!")

if __name__ == '__main__':
    p = Process(target=greet, args=("Alice",))
    p.start()
    p.join()
```

12.2.4 Example: Parallel Computation of Factorials

Full runnable code:

```
from multiprocessing import Process
import math

def compute_factorial(n):
    print(f"{n}! = {math.factorial(n)}")

if __name__ == '__main__':
    numbers = [100000, 50000, 20000]
    processes = []

    for n in numbers:
        p = Process(target=compute_factorial, args=(n,))
        processes.append(p)
        p.start()
```

```
    for p in processes:
        p.join()
```

Each number's factorial is calculated in a separate process, utilizing multiple CPU cores.

12.2.5 Sharing Data Between Processes

Since each process has its own memory, **sharing data** must be done explicitly.

Using `Queue`

Full runnable code:

```
from multiprocessing import Process, Queue

def square(n, q):
    q.put(n * n)

if __name__ == '__main__':
    q = Queue()
    p = Process(target=square, args=(4, q))
    p.start()
    p.join()
    result = q.get()
    print(f"Result from process: {result}")
```

12.2.6 Example: Image Processing (Simulated)

Full runnable code:

```
from multiprocessing import Process
import time

def process_image(image_name):
    print(f"Processing {image_name}...")
    time.sleep(1)
    print(f"{image_name} done.")

if __name__ == '__main__':
    images = ['img1.jpg', 'img2.jpg', 'img3.jpg']
    processes = [Process(target=process_image, args=(img,)) for img in images]

    for p in processes:
        p.start()
    for p in processes:
        p.join()

    print("All images processed.")
```

Each image is "processed" in parallel, reducing total execution time.

12.2.7 Summary: Threading vs. Multiprocessing

Feature	Threading	Multiprocessing
Ideal For	I/O-bound tasks	CPU-bound tasks
GIL Affected	Yes	No
Runs in Parallel	Not truly (due to GIL)	Yes (separate processes)
Memory	Shared between threads	Separate memory per process
Communication	Shared variables, `queue.Queue`	`multiprocessing.Queue` or `Pipe`

12.2.8 Best Practices

- Always guard process creation with `if __name__ == '__main__':` to avoid issues on Windows.
- Use `multiprocessing.Pool` for managing many parallel tasks easily (advanced topic).
- Be mindful of memory usage—each process has its own copy of the data.

12.3 Asynchronous Programming with `async` and `await`

Modern applications often need to perform many tasks at once, such as fetching data from the internet, reading from databases, or waiting for user input. **Asynchronous programming** allows Python programs to handle multiple **I/O-bound operations** concurrently—**without using threads or processes**.

12.3.1 What is Asynchronous Programming?

Asynchronous programming allows a program to **pause** a task while waiting (e.g., for data to load) and do something else in the meantime. This is especially useful when:

- Making multiple network/API requests
- Performing slow file or database I/O
- Managing thousands of simultaneous connections (e.g., a chat server)

12.3.2 Key Terms

Term	Meaning
`async def`	Defines an **asynchronous function**, known as a *coroutine*
`await`	Pauses execution until the awaited task completes
`asyncio.run()`	Starts and manages the event loop to execute `async` code
Event Loop	The engine that runs asynchronous code and schedules I/O-bound tasks

12.3.3 Basic Syntax

Full runnable code:

```
import asyncio

async def say_hello():
    print("Hello...")
    await asyncio.sleep(1)
    print("...World!")

asyncio.run(say_hello())
```

Output:

```
Hello...
...World!
```

- `asyncio.sleep(1)` simulates a delay without blocking the whole program.
- `await` tells Python to pause this function, let other tasks run, and resume after the delay.

12.3.4 Running Multiple Tasks Concurrently

Full runnable code:

```
import asyncio

async def greet(name):
    print(f"Starting: {name}")
    await asyncio.sleep(1)
    print(f"Hello, {name}!")

async def main():
    await asyncio.gather(
        greet("Alice"),
```

```
        greet("Bob"),
        greet("Charlie")
    )

asyncio.run(main())
```

Even though each task has a delay, they run **concurrently**, finishing in ~1 second instead of ~3.

12.3.5 Simulating I/O-Bound Work (e.g., Network Requests)

Full runnable code:

```
import asyncio

async def download_file(file_id):
    print(f"Downloading file {file_id}...")
    await asyncio.sleep(2)
    print(f"Download complete: file {file_id}")

async def main():
    tasks = [download_file(i) for i in range(3)]
    await asyncio.gather(*tasks)

asyncio.run(main())
```

12.3.6 Example: Asynchronous HTTP Requests with `aiohttp`

```
pip install aiohttp
```

Full runnable code:

```
import asyncio
import aiohttp

async def fetch(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            print(f"{url}: {response.status}")

async def main():
    urls = [
        "https://example.com",
        "https://httpbin.org/get",
        "https://python.org"
    ]
    tasks = [fetch(url) for url in urls]
```

```
    await asyncio.gather(*tasks)

asyncio.run(main())
```

12.3.7 Threading vs. Async: A Quick Comparison

Feature	Threading	Async Programming
Best For	I/O-bound or mixed tasks	I/O-bound (especially many simultaneous)
Overhead	Moderate (thread switching, memory)	Low (lightweight coroutines)
Uses Multiple Cores?	No (due to GIL)	No (still runs on one thread)
Code Structure	Looks like regular functions	Uses `async` / `await`
Concurrency Style	Preemptive (thread switching)	Cooperative (tasks yield control via `await`)

12.3.8 Summary

- Use `async def` to define asynchronous functions.
- Use `await` to pause and resume tasks at awaitable points.
- Use `asyncio.run()` to start the event loop.
- Combine multiple coroutines using `asyncio.gather()`.

12.3.9 Challenge Task

Write an async Python program that:

- Simulates downloading 5 files using `async def`
- Uses `asyncio.sleep()` to fake variable download durations
- Prints the start and end of each task

Asynchronous programming is a powerful tool for writing scalable, responsive applications—particularly network-driven ones like servers, chat apps, and crawlers. By mastering `async` and `await`, you can write highly efficient Python programs that juggle many tasks smoothly.

Chapter 13.

Testing and Debugging

1. Writing Unit Tests with `unittest`
2. Using Debuggers (`pdb`)
3. Best Practices for Writing Maintainable Code

13 Testing and Debugging

13.1 Writing Unit Tests with `unittest`

Testing is a crucial part of software development. It ensures your code works as expected and continues to work even after you make changes. One of the most common forms of testing in Python is **unit testing**, where you test small, individual pieces of code (usually functions or methods) in isolation.

Python comes with a powerful built-in module called `unittest` for writing and running unit tests.

13.1.1 Why Unit Testing?

- **Reliability**: Catch bugs early before they grow into major problems.
- **Refactoring Safety**: Feel confident changing code without breaking functionality.
- **Documentation**: Well-written tests show how your code is expected to behave.
- **Automation**: Tests can run automatically during development or in CI/CD pipelines.

13.1.2 Basic Structure of a `unittest` Test Case

Here's how a simple test case looks using the `unittest` framework:

Full runnable code:

```
import unittest

def add(a, b):
    return a + b

class TestAddFunction(unittest.TestCase):
    def test_add_positive_numbers(self):
        self.assertEqual(add(2, 3), 5)

    def test_add_negative_numbers(self):
        self.assertEqual(add(-1, -1), -2)

    def test_add_zero(self):
        self.assertEqual(add(0, 5), 5)

if __name__ == '__main__':
    unittest.main()
```

Explanation:

- `TestAddFunction`: The test case class inherits from `unittest.TestCase`.

- `test_...`: Method names that start with `test_` are automatically run by the framework.
- `self.assertEqual(...)`: One of many assertion methods used to check expected outcomes.
- `unittest.main()`: This runs all tests when the script is executed.

13.1.3 Common Assertion Methods

Method	Purpose
`assertEqual(a, b)`	Checks if `a == b`
`assertNotEqual(a, b)`	Checks if `a != b`
`assertTrue(x)` / `assertFalse(x)`	Checks the truthiness of `x`
`assertIsNone(x)`	Checks if `x is None`
`assertRaises(Exception, func, args...)`	Asserts that the function raises an error

13.1.4 Example: Testing for Exceptions

```
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b

class TestDivideFunction(unittest.TestCase):
    def test_divide_normal(self):
        self.assertEqual(divide(10, 2), 5)

    def test_divide_by_zero(self):
        with self.assertRaises(ValueError):
            divide(10, 0)
```

13.1.5 Organizing Tests in Separate Files

It's good practice to place tests in a separate file or directory. For example:

```
project/
+-- calculator.py
+-- tests/
    +-- test_calculator.py
```

test_calculator.py:

```
import unittest
from calculator import add, divide

class TestCalculator(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(2, 2), 4)

    def test_divide(self):
        self.assertEqual(divide(10, 2), 5)
```

13.1.6 Running Tests from the Command Line

To run your tests:

```
python -m unittest test_calculator.py
```

Or discover all tests in a directory:

```
python -m unittest discover
```

This will search for all files named `test*.py` and execute the test cases inside.

13.1.7 Testing Edge Cases

When writing tests, always think about:

- Zero, empty, or null inputs
- Extremely large/small numbers
- Unexpected data types
- Boundary conditions (e.g., first/last items in a list)

Example:

```
class TestEdgeCases(unittest.TestCase):
    def test_empty_string(self):
        self.assertEqual(len(""), 0)

    def test_large_input(self):
        self.assertEqual(add(1_000_000, 2_000_000), 3_000_000)
```

13.1.8 Summary

- Unit tests help ensure your code is correct, reliable, and maintainable.
- Python's `unittest` module offers a full-featured testing framework.

- Use assertions to validate behavior, including expected exceptions.
- Organize your tests in separate files and run them with `unittest`.

13.1.9 Challenge

Write a module `math_utils.py` with two functions: `square(x)` and `sqrt(x)`. Then write a test file `test_math_utils.py` that includes:

- A test for correct square values
- A test that checks `sqrt(-1)` raises a `ValueError`

In the next section, you'll learn how to use Python's built-in debugger `pdb` to diagnose issues when things go wrong during testing or development.

13.2 Using Debuggers (`pdb`)

Debugging is a critical skill for every programmer. While `print()` statements can be helpful, Python's built-in debugger—`pdb`—offers far more power and flexibility when you need to inspect what your code is doing step-by-step.

This section introduces `pdb` with hands-on examples and walks you through essential commands and workflows to track down and fix bugs efficiently.

13.2.1 What is `pdb`?

`pdb` stands for **Python Debugger**. It is a **command-line tool** that allows you to:

- Pause execution at any line of code (set breakpoints)
- Inspect variable values
- Step through code line-by-line
- Examine the call stack
- Continue, restart, or quit execution at any time

13.2.2 Why Use `pdb`?

- Helps trace **unexpected behavior**
- Avoids cluttering your code with `print()` statements
- Allows **interactive inspection** of variables and program flow

- Integrated into Python, so **no installation required**

13.2.3 Basic Example with a Bug

Consider the following function, which tries to remove even numbers from a list:

```
def remove_even(numbers):
    for i in range(len(numbers)):
        if numbers[i] % 2 == 0:
            numbers.pop(i)
    return numbers

print(remove_even([1, 2, 3, 4, 5, 6]))
```

Output:

```
[1, 3, 5]
```

This looks correct at a glance, but actually **skips numbers** due to the list changing while being iterated over.

13.2.4 Step 1: Insert a Breakpoint with `pdb`

You can insert a breakpoint using `import pdb; pdb.set_trace()`:

```
def remove_even(numbers):
    import pdb; pdb.set_trace()
    for i in range(len(numbers)):
        if numbers[i] % 2 == 0:
            numbers.pop(i)
    return numbers

print(remove_even([1, 2, 3, 4, 5, 6]))
```

When you run this program, youll enter interactive mode at the breakpoint:

```
> example.py(3)remove_even()
-> for i in range(len(numbers)):
(Pdb)
```

13.2.5 Step 2: Use Common `pdb` Commands

Here are some useful commands you can type at the `(Pdb)` prompt:

Command	Description
`n`	Next line (step over)
`s`	Step into function calls
`c`	Continue execution until the next breakpoint
`l`	List source code
`p var`	Print the value of `var`
`q`	Quit the debugger
`b 10`	Set a breakpoint at line 10

Try stepping through and inspecting:

```
(Pdb) p numbers
[1, 2, 3, 4, 5, 6]
(Pdb) n
(Pdb) p i
0
(Pdb) p numbers[i]
1
(Pdb) n
...
```

You'll see how `pop()` causes the loop to skip elements due to shifting indices.

13.2.6 Fixing the Bug

A safe way to remove elements while iterating is to use a **list comprehension** or iterate over a **copy** of the list:

```
def remove_even(numbers):
    return [num for num in numbers if num % 2 != 0]
```

Or:

```
def remove_even(numbers):
    for num in numbers[:]:  # iterate over a copy
        if num % 2 == 0:
            numbers.remove(num)
    return numbers
```

13.2.7 Running a Script in `pdb` from the Command Line

You can run any Python script in the debugger from the start using:

```
python -m pdb your_script.py
```

This opens the script at the first line and allows you to step through as it runs.

13.2.8 IDE-Integrated Debuggers

While `pdb` is powerful, modern IDEs offer user-friendly **graphical debuggers**:

- **VS Code**: Has built-in debugging with breakpoints, variable watchers, call stack navigation.
- **PyCharm**: Provides visual tools to step into, step over, and inspect runtime values.
- **Thonny**: Beginner-friendly Python IDE with excellent step-by-step execution views.

Using an IDE debugger can make understanding program flow more intuitive.

13.2.9 Summary

- Use `pdb` to pause and step through Python code interactively.
- Learn core commands like `n`, `s`, `p`, and `c` to navigate and inspect.
- Use breakpoints with `pdb.set_trace()` or run a script with `python -m pdb`.
- Prefer IDE-based debugging for visual tools and easier navigation.

13.3 Best Practices for Writing Maintainable Code

As your Python projects grow in size and complexity, *maintainability* becomes crucial. Maintainable code is easier to read, debug, test, and extend—not just by you, but by others as well.

This section outlines key principles and best practices that help you write clean, readable, and long-lasting Python code.

13.3.1 Use Meaningful Variable and Function Names

Choose descriptive names that convey the purpose of a variable or function.

YES **Good:**

```
def calculate_total_price(price, tax_rate):
    total = price + (price * tax_rate)
    return total
```

NO **Bad:**

```
def calc(p, t):
    return p + (p * t)
```

Why it matters: Clear names reduce the need for comments and make logic self-explanatory.

13.3.2 Keep Functions Small and Focused

A good function does **one thing only**. If a function becomes long or tries to do too much, break it into smaller helper functions.

YES **Good:**

```
def read_file(filepath):
    with open(filepath, 'r') as f:
        return f.read()

def count_words(text):
    return len(text.split())
```

NO **Bad:**

```
def read_and_count_words(filepath):
    with open(filepath, 'r') as f:
        text = f.read()
    words = text.split()
    return len(words)
```

Why it matters: Smaller functions are easier to test, reuse, and debug.

13.3.3 Write Modular Code

Organize your code into modules and functions so that related functionality is grouped together. Avoid putting all logic in one file or block.

YES **Example:**

- `math_utils.py` for math-related functions
- `file_utils.py` for file handling

This also encourages **code reuse** and **testability**.

13.3.4 Use Docstrings to Document Functions

Every function should describe **what it does**, **what inputs it expects**, and **what it returns**.

YES **Example:**

```
def greet(name):
    """Return a personalized greeting message."""
    return f"Hello, {name}!"
```

Use triple-quoted strings right below the function or class definition. Docstrings help you and others understand the purpose of the code without reading its internal logic.

13.3.5 Follow PEP 8 (Python Style Guide)

PEP 8 is the official style guide for Python. Some key rules:

- Use **4 spaces** for indentation (never tabs)
- Limit lines to **79 characters**
- Leave **2 blank lines** between functions
- Use `snake_case` for functions and variables
- Use `CamelCase` for class names
- Import modules at the top of the file

You can use tools like `flake8` or `black` to automatically check or format your code.

13.3.6 Prefer Explicit Code Over Clever Tricks

Readable code is better than clever one-liners that are hard to understand.

YES **Readable:**

```
if user_input.lower() == 'yes':
    proceed()
```

NO **Hard to Read:**

```
proceed() if user_input.lower() == 'yes' else None
```

Why it matters: Readability is one of Python's core design philosophies—*"Simple is better than complex."*

13.3.7 Avoid Hard-Coding and Magic Numbers

Avoid using unexplained constants in your code.

NO **Bad:**

```
discount = price * 0.85  # what is 0.85?
```

YES **Good:**

```
DISCOUNT_RATE = 0.15
discount = price * (1 - DISCOUNT_RATE)
```

13.3.8 Write Self-Validating Logic

Try to write code that *makes errors obvious* when something goes wrong.

YES **Good:**

```
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b
```

Instead of silently failing or returning incorrect values, fail early with helpful messages.

13.3.9 Comment *Why*, Not *What*

Only add comments when necessary—but when you do, focus on **why** something is done, not just what is done.

YES **Good:**

```
# Skip the first row because it contains headers
for row in data[1:]:
    process(row)
```

NO **Bad:**

```
# Loop over rows
for row in data[1:]:
    process(row)
```

13.3.10 Test Your Code Early and Often

Use unit tests (as covered earlier in this chapter) to validate that your functions behave as expected. Writing testable code often forces you to write cleaner, modular functions.

13.3.11 Final Thoughts

Writing maintainable code is more than a matter of style—it's about building software that's easy to evolve, debug, and share. Here's a quick checklist:

- YES Descriptive names
- YES Short, focused functions
- YES Modular structure
- YES Docstrings and comments
- YES PEP8 formatting
- YES Clear, readable logic
- YES Error handling
- YES Avoid magic numbers

- YES Write tests

By following these habits, you'll reduce bugs, improve clarity, and become a much more productive and professional Python developer.

Chapter 14.

File and Data Processing Projects

1. Web Scraping Basics (`requests`, `BeautifulSoup`)
2. Working with APIs and JSON Data
3. Automating Tasks and Scripting

14 File and Data Processing Projects

14.1 Web Scraping Basics (`requests`, `BeautifulSoup`)

Web scraping is the process of programmatically extracting data from websites. In Python, this is often done using two powerful libraries:

- `requests` – to download web pages.
- `BeautifulSoup` – to parse and navigate the HTML content.

This section introduces you to basic web scraping techniques and shows you how to extract useful information such as headlines, links, and product details.

14.1.1 Installing Required Libraries

To follow along, install the required packages using `pip`:

```
pip install requests beautifulsoup4
```

14.1.2 Step 1: Fetch a Web Page with `requests`

The `requests` library makes it easy to download a webpage's content:

```
import requests

url = "https://example.com"
response = requests.get(url)

print(response.text)  # Shows the raw HTML
```

Check the response status code:

```
if response.status_code == 200:
    print("Page fetched successfully!")
else:
    print("Failed to retrieve the page.")
```

14.1.3 Step 2: Parse HTML with `BeautifulSoup`

After fetching the page, we parse it with `BeautifulSoup`:

```
from bs4 import BeautifulSoup

soup = BeautifulSoup(response.text, "html.parser")
print(soup.prettify())  # View the structured HTML
```

14.1.4 Step 3: Extracting Data

Example: Extracting All Links

```
for link in soup.find_all('a'):
    print(link.get('href'))
```

Example: Extracting Headings

```
headings = soup.find_all('h2')
for h in headings:
    print(h.text.strip())
```

Example: Extracting Items by Class

Suppose you want to scrape product names with class `"product-title"`:

```
products = soup.find_all(class_='product-title')
for product in products:
    print(product.text.strip())
```

14.1.5 Step 4: Navigating the DOM

You can chain methods to drill down into elements:

```
article = soup.find('div', class_='article')
title = article.find('h1').text
print(title)
```

You can also use CSS selectors:

```
for item in soup.select('.news-item h3'):
    print(item.text)
```

14.1.6 Sample Project: Scraping Top News Headlines

Let's build a quick script to extract headlines from a real website like https://news.ycombinator.com:

```
import requests
from bs4 import BeautifulSoup

url = "https://news.ycombinator.com"
response = requests.get(url)
soup = BeautifulSoup(response.text, "html.parser")

headlines = soup.select('.titleline > a')
for i, headline in enumerate(headlines[:10], 1):
    print(f"{i}. {headline.text}")
```

14.1.7 Step 5: Ethics and Best Practices

Web scraping must be done **responsibly**. Keep in mind:

- YES **Check `robots.txt`:** Many websites specify rules for automated access (e.g., `https://example.com/robots.txt`).
- YES **Use Headers:** Mimic a browser using headers like `User-Agent`.
- YES **Throttle Requests:** Avoid overloading servers. Use `time.sleep()` between requests.
- NO **Never scrape login-protected or copyrighted content** without permission.

```
import time

for page in range(1, 5):
    response = requests.get(f"https://example.com/page={page}")
    time.sleep(1)  # Pause 1 second between requests
```

14.1.8 Summary

In this section, you learned how to:

- Use `requests` to download web pages
- Use `BeautifulSoup` to parse and navigate HTML
- Extract data like links, headings, and product info
- Build a simple scraping script
- Follow ethical guidelines and avoid legal trouble

14.2 Working with APIs and JSON Data

Modern applications often rely on data from web-based APIs (Application Programming Interfaces). APIs typically return data in **JSON (JavaScript Object Notation)** format, which is lightweight and easy to use in Python.

In this section, you'll learn how to:

- Use `requests.get()` to make API calls
- Handle query parameters and headers
- Parse and extract data from JSON responses
- Handle API errors gracefully
- Save API results to a file

14.2.1 Step 1: Making a Simple API Request

Let's start by making a request to a sample API. We'll use a free fake API for demonstration: `https://jsonplaceholder.typicode.com`.

```
import requests

url = "https://jsonplaceholder.typicode.com/posts"
response = requests.get(url)

print(response.status_code)  # Should be 200 for success
print(response.text)         # Raw JSON string
```

14.2.2 Step 2: Parsing JSON Data

The `response.json()` method converts the JSON string into Python data types (lists and dictionaries):

```
data = response.json()

# Print the first post's title
print(data[0]['title'])
```

14.2.3 Step 3: Accessing Nested JSON Fields

Many APIs return nested data. For example:

```
{
  "user": {
    "id": 1,
    "profile": {
      "name": "Alice",
      "email": "alice@example.com"
    }
  }
}
```

To access the email:

```
email = data['user']['profile']['email']
```

14.2.4 Step 4: Adding Query Parameters

Some APIs require query parameters (e.g., filtering or pagination). Pass them as a dictionary using the `params` keyword:

```
params = {'userId': 1}
response = requests.get("https://jsonplaceholder.typicode.com/posts", params=params)
posts = response.json()

for post in posts:
    print(f"{post['id']}: {post['title']}")
```

14.2.5 Step 5: Handling Headers and Error Responses

Some APIs require custom headers, like API keys. You can send headers like this:

```
headers = {
    'Authorization': 'Bearer YOUR_API_KEY_HERE'
}
response = requests.get('https://api.example.com/data', headers=headers)
```

Check the response status code to handle errors:

```
if response.status_code == 200:
    data = response.json()
else:
    print(f"Error: {response.status_code}")
```

14.2.6 Step 6: Example Saving Data to a File

Let's fetch data from the API and save specific fields to a file.

```
import json

response = requests.get("https://jsonplaceholder.typicode.com/users")
users = response.json()

# Extract names and emails
user_contacts = [{"name": u["name"], "email": u["email"]} for u in users]

# Save to a JSON file
with open("contacts.json", "w") as f:
    json.dump(user_contacts, f, indent=2)

print("Saved contacts to contacts.json")
```

14.2.7 Optional: Working with Real APIs

If you'd like to work with real-world APIs, here are some popular examples:

- OpenWeatherMap – Weather data

- REST Countries – Country and region info
- NewsAPI – News headlines

Most require an API key (a free account is usually enough).

14.2.8 Best Practices

- YES Read the API documentation carefully.
- YES Always handle errors (`try/except` or check `response.status_code`).
- YES Respect rate limits (throttle your requests if needed).
- YES Use pagination for large datasets.

14.3 Automating Tasks and Scripting

Python shines in automating repetitive or time-consuming tasks. Whether it's renaming hundreds of files, sending routine emails, scraping data, or organizing content—Python can streamline it efficiently with minimal code.

In this section, you'll learn how to:

- Automate file operations using `os` and `shutil`
- Use scheduling and conditions to control task execution
- Handle errors and write cross-platform scripts
- Create two practical automation scripts

14.3.1 Example 1: Renaming Files in a Folder

Let's automate renaming a batch of files using the `os` module.

Task: Rename all `.txt` files in a folder to have a `processed_` prefix.

Full runnable code:

```
import os

folder_path = "./my_text_files"

try:
    for filename in os.listdir(folder_path):
        if filename.endswith(".txt"):
            old_path = os.path.join(folder_path, filename)
            new_filename = "processed_" + filename
```

```
            new_path = os.path.join(folder_path, new_filename)
            os.rename(old_path, new_path)
            print(f"Renamed: {filename} → {new_filename}")
except FileNotFoundError:
    print("The specified folder was not found.")
except PermissionError:
    print("Permission denied. Try running as administrator.")
```

Notes:

- Works on Windows, macOS, and Linux
- Always use `os.path.join()` for cross-platform paths
- Add logging or backups in production scripts

14.3.2 Example 2: Scheduled Web Scraping and CSV Export

Let's write a script that scrapes data every morning and saves it.

Task: Fetch top headlines and save them to a CSV.

```
import requests
from bs4 import BeautifulSoup
import csv
import datetime
import time

def fetch_headlines():
    url = "https://news.ycombinator.com"
    response = requests.get(url)
    soup = BeautifulSoup(response.text, 'html.parser')
    headlines = soup.select('.titleline > a')

    today = datetime.date.today().isoformat()
    filename = f"headlines_{today}.csv"

    with open(filename, "w", newline='', encoding='utf-8') as file:
        writer = csv.writer(file)
        writer.writerow(["Title", "URL"])
        for h in headlines:
            writer.writerow([h.get_text(), h['href']])

    print(f"Saved {len(headlines)} headlines to {filename}")

# Run the task once daily
while True:
    now = datetime.datetime.now()
    if now.hour == 7:  # Run at 7 AM
        fetch_headlines()
        time.sleep(86400)  # Sleep for 24 hours
    else:
        time.sleep(60)  # Check again in 1 minute
```

You can stop the script using `Ctrl + C`. For actual deployment, consider using a task scheduler:

- **Windows:** Task Scheduler
- **macOS/Linux:** `cron`

14.3.3 Real-World Automation Ideas

- **Back up important files** using `shutil.copy()`
- **Clean up downloads or temp folders** based on file age
- **Send daily reports via email** using `smtplib`
- **Scrape and update a database** with product prices or weather data

14.3.4 Cross-Platform and Error-Handling Tips

Challenge	Best Practice
File paths	Use `os.path.join()` or `pathlib.Path()`
Scheduling tasks	Use OS-native tools (`cron`, Task Scheduler)
File permission issues	Use `try/except` blocks and avoid hardcoding paths
Encoding problems	Use `encoding='utf-8'` when reading/writing files

14.3.5 Summary

Python makes it easy to automate mundane tasks that would otherwise take hours. In this section, you:

- Renamed multiple files automatically
- Scheduled a script to scrape headlines and save them
- Learned how to write scripts that are safe, flexible, and reusable

Chapter 15.

Python in Data Science and Machine Learning (Intro)

1. Using `numpy` and `pandas` for Data Manipulation
2. Plotting with `matplotlib` and `seaborn`
3. Introduction to `scikit-learn`

15 Python in Data Science and Machine Learning (Intro)

15.1 Using `numpy` and `pandas` for Data Manipulation

When working with data in Python, two libraries stand out as essential tools:

- **`numpy`**: provides fast array operations and numerical computing tools.
- **`pandas`**: built on top of `numpy`, it adds powerful, flexible data structures for handling labeled data like tables (spreadsheets or SQL-like datasets).

This section introduces both libraries with practical examples to help you prepare, clean, and manipulate data efficiently.

Getting Started

Install the libraries (if not already installed) using:

```
pip install numpy pandas
```

15.1.1 Part 1: `numpy` Efficient Numerical Arrays

Creating Arrays

```
import numpy as np

# Create a 1D array
a = np.array([1, 2, 3, 4])

# Create a 2D array
b = np.array([[1, 2], [3, 4]])

print(a)
print(b)
```

Array Slicing and Indexing

```
# Get the second element
print(a[1])  # Output: 2

# Slice first three elements
print(a[:3])  # Output: [1 2 3]

# Access rows/columns in 2D
print(b[0, 1])  # Output: 2
```

Vectorized Operations

```
# Add 10 to every element
print(a + 10)

# Element-wise multiplication
print(a * 2)

# Useful functions
print(np.mean(a))      # Average
print(np.std(a))       # Standard deviation
print(np.sum(a))       # Total sum
```

15.1.2 Part 2: pandas DataFrames and Series

Creating a DataFrame

Let's create a simple dataset of student grades.

```
import pandas as pd

data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'Diana'],
    'Math': [85, 92, 78, 90],
    'Science': [88, 79, 93, 85]
}

df = pd.DataFrame(data)
print(df)
```

Output:

```
      Name  Math  Science
0    Alice    85       88
1      Bob    92       79
2  Charlie    78       93
3    Diana    90       85
```

Accessing and Filtering Data

```
# Access column
print(df['Math'])

# Get a single row
print(df.loc[2])  # Row for Charlie

# Filter rows
print(df[df['Math'] > 85])
```

Basic Aggregation

```
# Average score per subject
print(df.mean(numeric_only=True))
```

```
# Maximum math score
print(df['Math'].max())
```

Data Cleaning and Transformation

Let's introduce missing data and handle it.

```
df.loc[1, 'Science'] = None  # Set Bob's science score as missing

# Fill missing values
df['Science'] = df['Science'].fillna(df['Science'].mean())

# Add a new column for average score
df['Average'] = df[['Math', 'Science']].mean(axis=1)

print(df)
```

15.1.3 Example: Product Sales Dataset

```
sales_data = {
    'Product': ['A', 'B', 'C', 'A', 'B'],
    'Region': ['North', 'South', 'East', 'South', 'North'],
    'Sales': [100, 150, 200, 130, 170]
}

sales_df = pd.DataFrame(sales_data)

# Total sales per product
print(sales_df.groupby('Product')['Sales'].sum())

# Average sales per region
print(sales_df.groupby('Region')['Sales'].mean())
```

Summary

- **numpy** is ideal for fast, low-level numerical operations on arrays and matrices.
- **pandas** provides high-level data structures like Series and DataFrames for tabular data.
- Together, they are foundational for data science and analytics workflows in Python.

15.2 Plotting with `matplotlib` and `seaborn`

Visualization is a key part of understanding data. Python offers two powerful libraries for data visualization:

- **`matplotlib`** – The foundational plotting library, flexible and highly customizable.
- **`seaborn`** – Built on top of `matplotlib`, it provides simpler syntax and attractive default styles, especially for statistical plots.

In this section, we'll explore how to create basic charts using both libraries, working from the same dataset and noting the differences in syntax and output.

Setup

Install the required libraries if needed:

```
pip install matplotlib seaborn pandas
```

Import them in your script:

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

Let's create a simple `pandas` DataFrame of student test scores:

```
data = {
    'Student': ['Alice', 'Bob', 'Charlie', 'Diana', 'Ethan'],
    'Math': [88, 92, 79, 85, 95],
    'Science': [90, 85, 86, 88, 93],
    'Hours_Studied': [5, 6, 3, 4, 7]
}

df = pd.DataFrame(data)
```

15.2.1 Line Plot

Using `matplotlib`:

```
plt.plot(df['Student'], df['Math'], label='Math')
plt.plot(df['Student'], df['Science'], label='Science')
plt.title('Test Scores by Student')
plt.xlabel('Student')
plt.ylabel('Score')
plt.legend()
plt.grid(True)
plt.show()
```

Using `seaborn`:

```
sns.lineplot(data=df.set_index('Student')[['Math', 'Science']])
plt.title('Test Scores by Student')
plt.ylabel('Score')
plt.show()
```

> **Note**: `seaborn` integrates tightly with `pandas` DataFrames and automatically handles legends and styles.

15.2.2 Bar Chart

Using `matplotlib`:

```
plt.bar(df['Student'], df['Hours_Studied'], color='skyblue')
plt.title('Hours Studied per Student')
plt.xlabel('Student')
plt.ylabel('Hours')
plt.show()
```

Using `seaborn`:

```
sns.barplot(x='Student', y='Hours_Studied', data=df, palette='pastel')
plt.title('Hours Studied per Student')
plt.show()
```

15.2.3 Scatter Plot

Scatter plots are useful for exploring relationships between variables.

Using `matplotlib`:

```
plt.scatter(df['Hours_Studied'], df['Math'], color='green')
plt.title('Math Score vs. Hours Studied')
plt.xlabel('Hours Studied')
plt.ylabel('Math Score')
plt.show()
```

Using `seaborn`:

```
sns.scatterplot(x='Hours_Studied', y='Math', data=df)
plt.title('Math Score vs. Hours Studied')
plt.show()
```

15.2.4 Histogram

Histograms show frequency distribution.

Using `matplotlib`:

```
plt.hist(df['Math'], bins=5, color='purple', edgecolor='black')
plt.title('Distribution of Math Scores')
plt.xlabel('Score')
plt.ylabel('Frequency')
plt.show()
```

Using `seaborn`:

```
sns.histplot(df['Math'], bins=5, kde=True, color='purple')
plt.title('Distribution of Math Scores')
plt.show()
```

The `kde=True` in `seaborn` adds a Kernel Density Estimation line, giving a smoothed view of the distribution.

15.2.5 Customization Tips

Both libraries allow customization of:

- **Titles**: `plt.title()`
- **Axis labels**: `plt.xlabel()`, `plt.ylabel()`
- **Legends**: `plt.legend()`
- **Colors**: via `color=`, `palette=`
- **Grid lines**: `plt.grid(True)`

15.2.6 Syntax & Style Comparison

Feature	`matplotlib`	`seaborn`
Syntax	Low-level, more manual	High-level, integrated with `pandas`
Aesthetics	Basic by default	Clean, statistical-focused by default
Best for	Full control and custom visualizations	Quick, elegant visuals for data exploration
Integration	Less automatic with DataFrames	Seamless with DataFrames

Summary

- Use **`matplotlib`** when you need complete control over plot layout and appearance.
- Use **`seaborn`** for faster, cleaner visuals with built-in statistical insights.
- Practice plotting with your own datasets to solidify your understanding.

Suggested Exercises

1. Create a line plot comparing two subjects over time.
2. Plot a histogram of random scores and overlay a KDE.
3. Generate a scatter plot showing correlation between hours studied and science score.

15.3 Introduction to `scikit-learn`

`scikit-learn` is one of the most widely used libraries for machine learning in Python. It provides simple, efficient tools for data mining and analysis, built on top of `numpy`, `scipy`, and `matplotlib`.

In this section, you'll learn the basic workflow for building and evaluating machine learning models using `scikit-learn`. We'll use a classic dataset — the **Iris dataset** — to demonstrate classification.

15.3.1 Machine Learning Workflow with `scikit-learn`

The typical steps for a machine learning project using `scikit-learn` are:

1. **Load the dataset**
2. **Split the data** into training and testing sets
3. **Choose and train a model**
4. **Make predictions**
5. **Evaluate the model**

Let's walk through each step with code.

15.3.2 Step 1: Load the Dataset

The Iris dataset contains measurements for three species of iris flowers (setosa, versicolor, virginica).

```
from sklearn.datasets import load_iris

iris = load_iris()
X = iris.data  # Features: sepal/petal length and width
y = iris.target  # Labels: 0=setosa, 1=versicolor, 2=virginica

print(iris.feature_names)
# Output: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```

15.3.3 Step 2: Split the Data

Split the data into training and test sets using `train_test_split`.

```
from sklearn.model_selection import train_test_split

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

```
)
```

15.3.4 Step 3: Train a Model

Use a simple classifier such as `DecisionTreeClassifier`.

```
from sklearn.tree import DecisionTreeClassifier

model = DecisionTreeClassifier()
model.fit(X_train, y_train)
```

15.3.5 Step 4: Make Predictions

Use the trained model to predict the test data.

```
y_pred = model.predict(X_test)
print(y_pred)
```

15.3.6 Step 5: Evaluate the Model

Use `accuracy_score` to evaluate prediction accuracy.

```
from sklearn.metrics import accuracy_score

accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
```

15.3.7 Summary: Full Example

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Load data
iris = load_iris()
X = iris.data
y = iris.target

# Split data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
```

```
)

# Train model
model = DecisionTreeClassifier()
model.fit(X_train, y_train)

# Predict
y_pred = model.predict(X_test)

# Evaluate
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy:.2f}")
```

15.3.8 Additional Notes

Overfitting

Overfitting occurs when a model performs well on training data but poorly on unseen data. To reduce overfitting:

- Use **simpler models**
- Apply **cross-validation**
- Limit **tree depth** or use **regularization**

Model Selection

`scikit-learn` provides many models beyond decision trees, such as:

- `LogisticRegression` for binary classification
- `RandomForestClassifier` for ensemble learning
- `LinearRegression` for numeric prediction

15.3.9 Suggested Practice

- Use the **digits dataset** (`load_digits`) and classify handwritten digits.
- Modify the `DecisionTreeClassifier` to limit depth using `max_depth=3`.
- Try using `RandomForestClassifier` and compare accuracy.

15.3.10 Conclusion

You've just built and evaluated your first machine learning model using `scikit-learn`. Although this example is simple, it mirrors real-world workflows and sets the foundation for

more advanced techniques.

Chapter 16.

Project: Command-Line To-Do List Application

1. CRUD Operations with File Persistence
2. User Input and Data Validation

16 Project: Command-Line To-Do List Application

16.1 CRUD Operations with File Persistence

A to-do list is a classic beginner project that teaches core programming concepts such as:

- File I/O
- Data persistence
- CRUD operations
- Modular code design

In this section, we'll create a simple command-line application that lets users:

- **Create** a task
- **Read** (list) all tasks
- **Update** a task's status (mark as done)
- **Delete** a task

We'll use a **JSON file** to store tasks so they persist between program runs.

16.1.1 Step 1: Storing Task Data

Each task will be represented as a Python dictionary:

```
{
    "id": 1,
    "title": "Buy groceries",
    "done": False
}
```

Tasks will be stored in a list and saved to a file named `tasks.json`.

16.1.2 Step 2: Utility Functions

Let's start by creating helper functions to load and save tasks.

```
import json
import os

FILE_NAME = 'tasks.json'

def load_tasks():
    if not os.path.exists(FILE_NAME):
        return []
    with open(FILE_NAME, 'r') as file:
        return json.load(file)

def save_tasks(tasks):
```

```
    with open(FILE_NAME, 'w') as file:
        json.dump(tasks, file, indent=2)
```

16.1.3 Step 3: Create Add a Task

```
def add_task(title):
    tasks = load_tasks()
    task_id = tasks[-1]['id'] + 1 if tasks else 1
    tasks.append({
        "id": task_id,
        "title": title,
        "done": False
    })
    save_tasks(tasks)
    print(f"Task added: [{task_id}] {title}")
```

Example Usage:

```
add_task("Finish homework")
```

16.1.4 Step 4: Read List All Tasks

```
def list_tasks():
    tasks = load_tasks()
    if not tasks:
        print("No tasks found.")
        return
    for task in tasks:
        status = " " if task["done"] else " "
        print(f"[{task['id']}] {status} {task['title']}")
```

Example Output:

```
[1]   Finish homework
[2]   Buy groceries
```

16.1.5 Step 5: Update Mark Task as Done

```
def mark_done(task_id):
    tasks = load_tasks()
    for task in tasks:
        if task["id"] == task_id:
            task["done"] = True
            save_tasks(tasks)
```

```
            print(f"Task {task_id} marked as done.")
            return
    print(f"No task found with ID {task_id}.")
```

Example Usage:

```
mark_done(1)
```

16.1.6 Step 6: Delete Remove a Task

```
def delete_task(task_id):
    tasks = load_tasks()
    filtered = [task for task in tasks if task["id"] != task_id]
    if len(filtered) == len(tasks):
        print(f"No task found with ID {task_id}.")
    else:
        save_tasks(filtered)
        print(f"Task {task_id} deleted.")
```

Example Usage:

```
delete_task(2)
```

16.1.7 Complete Command Example

You can wire these functions into a simple CLI menu:

```
def main():
    while True:
        print("\nTo-Do List Menu")
        print("1. Add Task")
        print("2. List Tasks")
        print("3. Mark Task as Done")
        print("4. Delete Task")
        print("5. Exit")
        choice = input("Choose an option: ")

        if choice == "1":
            title = input("Enter task title: ")
            add_task(title)
        elif choice == "2":
            list_tasks()
        elif choice == "3":
            try:
                task_id = int(input("Enter task ID to mark as done: "))
                mark_done(task_id)
            except ValueError:
                print("Invalid input.")
        elif choice == "4":
            try:
```

```
                task_id = int(input("Enter task ID to delete: "))
                delete_task(task_id)
            except ValueError:
                print("Invalid input.")
        elif choice == "5":
            print("Goodbye!")
            break
        else:
            print("Invalid option. Try again.")

if __name__ == "__main__":
    main()
```

16.1.8 Summary

In this section, we built a complete file-backed to-do list application that supports:

- **Create**: Adding a task
- **Read**: Listing tasks
- **Update**: Marking as done
- **Delete**: Removing tasks

All tasks are saved in `tasks.json`, so your progress persists across sessions.

16.2 User Input and Data Validation

Now that your command-line to-do application supports CRUD operations with persistent storage, the next step is ensuring it behaves well when users interact with it. This means **capturing user input clearly** and **validating it properly**.

Poorly handled input can crash your app or confuse the user. Good input validation makes your application more reliable, professional, and enjoyable to use.

16.2.1 Basic Input Handling with `input()`

Python's `input()` function lets you capture user commands and data:

```
command = input("Enter command (add/list/done/delete/exit): ").strip().lower()
```

We use `.strip()` to remove extra spaces and `.lower()` to allow case-insensitive input.

16.2.2 Example: A Command Loop with Help Messages

```
def show_help():
    print("\nAvailable commands:")
    print("  add      - Add a new task")
    print("  list     - List all tasks")
    print("  done     - Mark a task as done")
    print("  delete   - Delete a task")
    print("  help     - Show this help message")
    print("  exit     - Exit the application")

def main():
    print("Welcome to the To-Do List App!")
    show_help()

    while True:
        command = input("\nCommand: ").strip().lower()

        if command == "add":
            title = input("Task title: ").strip()
            if title:
                add_task(title)
            else:
                print("Error: Task title cannot be empty.")

        elif command == "list":
            list_tasks()

        elif command == "done":
            try:
                task_id = int(input("Enter task ID to mark as done: "))
                mark_done(task_id)
            except ValueError:
                print("Error: Task ID must be a number.")

        elif command == "delete":
            try:
                task_id = int(input("Enter task ID to delete: "))
                confirm = input(f"Are you sure you want to delete task {task_id}? (y/n): ").strip().lowe
                if confirm == 'y':
                    delete_task(task_id)
                else:
                    print("Deletion cancelled.")
            except ValueError:
                print("Error: Task ID must be a number.")

        elif command == "help":
            show_help()

        elif command == "exit":
            print("Goodbye!")
            break

        else:
            print(f"Unknown command: '{command}' (type 'help' for options)")
```

16.2.3 Validating Common Inputs

Validation Task	Example/Error Message
Empty title	"Error: Task title cannot be empty."
Non-numeric task ID	"Error: Task ID must be a number."
Unrecognized command	"Unknown command: 'foo' (type 'help'...)"
Confirming risky operations	Use `input("Are you sure? (y/n)")`
Preventing duplicate tasks (optional)	"Task already exists."

16.2.4 Optional Enhancement: Colored Terminal Output

To improve readability, you can use the `colorama` library:

```
pip install colorama
```

Example usage:

```
from colorama import Fore, Style, init
init(autoreset=True)

print(Fore.GREEN + "Task added successfully.")
print(Fore.RED + "Error: Invalid input.")
print(Fore.YELLOW + "Warning: No tasks found.")
```

16.2.5 Summary

This section focused on:

- Capturing user input using `input()`
- Validating user input to prevent errors
- Providing helpful prompts and messages
- (Optionally) Enhancing the interface with color and confirmation prompts

By implementing these best practices, your to-do list app becomes not only functional—but also **reliable, user-friendly, and professional**.

Chapter 17.

Project: Web Scraper for News Headlines

1. Fetching Web Pages
2. Parsing and Extracting Data

17 Project: Web Scraper for News Headlines

17.1 Fetching Web Pages

Before extracting news headlines, the first step is to **fetch the HTML content** of a webpage. The `requests` library makes this straightforward and reliable.

17.1.1 Installing `requests`

If you haven't installed it yet, run:

```
pip install requests
```

17.1.2 Making a GET Request

To get the content of a webpage, you send an HTTP GET request:

```
import requests

url = "https://news.ycombinator.com/"  # Example: Hacker News homepage

try:
    response = requests.get(url)
    print(f"Status Code: {response.status_code}")

    # Check if the request was successful
    if response.status_code == 200:
        print("Page fetched successfully!")
        print("HTML snippet:")
        print(response.text[:500])  # Print first 500 characters of raw HTML
    else:
        print(f"Failed to retrieve page. Status code: {response.status_code}")

except requests.exceptions.RequestException as e:
    print(f"An error occurred: {e}")
```

17.1.3 Explanation

- `requests.get(url)` sends the GET request.
- `response.status_code` tells you the HTTP response status (200 means OK).
- `response.text` contains the full HTML content as a string.
- The `try/except` block catches common network-related errors (like timeouts or connection errors).

17.1.4 Handling Headers and User-Agent

Some websites may block requests that look like bots or scripts. To reduce the chance of being blocked, you can add headers to mimic a real browser:

```
headers = {
    "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) "
                  "AppleWebKit/537.36 (KHTML, like Gecko) "
                  "Chrome/114.0.0.0 Safari/537.36"
}

response = requests.get(url, headers=headers)
```

Including a User-Agent string often helps avoid getting blocked or served limited content.

17.1.5 Summary

- Use `requests.get()` to fetch web pages.
- Always check the status code to confirm success.
- Handle exceptions with `try/except` for robustness.
- Optionally, include headers with a User-Agent to simulate a browser.

With the HTML content in hand, you're now ready to parse it and extract news headlines — which we will cover in the next section.

17.2 Parsing and Extracting Data

After fetching the raw HTML of a news webpage, the next step is to **parse the HTML and extract the headlines, links, timestamps, or other relevant information**. The `BeautifulSoup` library makes navigating and searching the HTML tree simple and intuitive.

17.2.1 Installing `BeautifulSoup`

If you haven't installed it yet, run:

```
pip install beautifulsoup4
```

17.2.2 Using Developer Tools to Inspect the Webpage

Before writing code, use your browser's developer tools (right-click on a headline → Inspect) to find:

- The **tag name** (e.g., `h2`, `a`, `span`)
- Relevant **CSS classes or IDs**
- The attributes you want (like `href` for links)

This helps you target the right HTML elements in your code.

17.2.3 Example: Extracting Headlines and Links

Let's say you've fetched HTML from a site where headlines are inside `<a>` tags with class `"storylink"` (like Hacker News).

```
from bs4 import BeautifulSoup

# Sample HTML content from previous section
html_content = """
<html>
  <body>
    <a class="storylink" href="https://example.com/article1">Headline 1</a>
    <a class="storylink" href="https://example.com/article2">Headline 2</a>
    <a class="storylink" href="https://example.com/article3">Headline 3</a>
  </body>
</html>
"""

# Parse the HTML
soup = BeautifulSoup(html_content, 'html.parser')

# Find all <a> tags with class 'storylink'
headlines = soup.find_all('a', class_='storylink')

# Extract headline text and link
news_items = []
for item in headlines:
    title = item.get_text(strip=True)   # Clean text, strip whitespace
    link = item.get('href')              # Extract href attribute
    news_items.append({'title': title, 'link': link})

# Display results
for news in news_items:
    print(f"Title: {news['title']}")
    print(f"Link: {news['link']}\n")
```

Output:

```
Title: Headline 1
Link: https://example.com/article1
```

```
Title: Headline 2
Link: https://example.com/article2

Title: Headline 3
Link: https://example.com/article3
```

17.2.4 Handling Additional Data: Timestamps or Summaries

You can also extract other data by targeting additional tags or classes. For example, if each news item has a timestamp in a `<span>` with class `"age"`, you would:

```
timestamps = soup.find_all('span', class_='age')

for idx, ts in enumerate(timestamps):
    news_items[idx]['timestamp'] = ts.get_text(strip=True)
```

17.2.5 Working with `.find()` vs `.find_all()`

- `.find()` returns the **first** matching element.
- `.find_all()` returns **all** matching elements as a list.

Use `.find()` if you only need one element, or `.find_all()` to get multiple.

17.2.6 Saving Extracted Data to CSV

Once you have your list of dictionaries (`news_items`), you can save it for further use:

```
import csv

with open('headlines.csv', 'w', newline='', encoding='utf-8') as file:
    writer = csv.DictWriter(file, fieldnames=['title', 'link'])
    writer.writeheader()
    writer.writerows(news_items)

print("Saved headlines to headlines.csv")
```

17.2.7 Writing Adaptable Code

Websites often change their HTML structure. To make your scraper more resilient:

- Avoid hardcoding too many CSS classes.

- Use combinations of tags, classes, or attributes.
- Handle missing elements gracefully (check if an element exists before accessing).
- Consider logging unexpected structures for debugging.

Example with checks:

```
for item in headlines:
    title = item.get_text(strip=True) if item else 'No Title'
    link = item.get('href') if item and item.has_attr('href') else 'No Link'
    news_items.append({'title': title, 'link': link})
```

17.2.8 Summary

- Use `BeautifulSoup` to parse and navigate HTML.
- Inspect the page with browser developer tools to find relevant tags and classes.
- Use `.find()` and `.find_all()` to select elements.
- Access text with `.get_text()` and attributes with `.get()`.
- Store extracted data in lists or dictionaries.
- Save your results to CSV or other formats for further analysis.
- Write flexible, defensive code to handle site changes.