

# Essentials: Grammar, Semantics, and Interaction

## Applied Python Programming with AI and Raspberry Pi Interfaces

Instructor: Dr. Vikas Thammanna Gowda

Semester: ABCD 20YX

**Module Overview:** *This chapter introduces the fundamental building blocks of programming in Python through the lens of computing concepts and everyday logic. Students will begin by understanding how computers work and interact with humans, laying the groundwork for writing meaningful programs. The focus then shifts to Python syntax (grammar), semantics (meaning), and interaction with users and systems. Key topics include declaring and manipulating variables, understanding data types, performing arithmetic and logical operations using operators, and writing programs that interact with users through input/output (I/O). The chapter also introduces the concept of errors and exception handling, helping students build confidence in debugging and writing resilient code. This module forms the core foundation on which all subsequent programming logic and design will be built.*

### Learning Objectives:

1. *Describe the basic components and functionality of a computer system.*
2. *Understand the purpose of a programming language and its role in human-computer interaction.*
3. *Define and use variables to store and manipulate data in Python.*
4. *Identify and work with Python's basic data types (integers, floats, strings, booleans).*
5. *Apply arithmetic, comparison, and logical operators in expressions.*
6. *Use input and output functions to interact with users.*
7. *Recognize common types of errors and apply basic exception handling to make programs more robust.*
8. *Develop simple Python programs that demonstrate syntax correctness, logical flow, and user interaction.*

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	What Is a Computer?	4
1.1.1	Basic Hardware vs. Software	4
1.1.2	How CPU, Memory, Storage, and I/O Work Together	4
1.1.3	Types of Computers	4
1.2	What Is Programming?	6
1.3	Programming Languages: Low-Level vs. High-Level	7
1.3.1	Low-Level Languages (Machine Code and Assembly)	7
1.3.2	High-Level Languages (Python, Java, C++)	8
1.3.3	Trade-offs: Control vs. Productivity	8
1.4	Program Translation	8
1.4.1	Assemblers (for Assembly Language)	8
1.4.2	Compilers (e.g., for C/C++) vs. Interpreters (e.g., for Python)	9
1.4.3	Bytecode and Just-In-Time Compilation (JIT)	9
1.5	The Software Development Life Cycle	10
1.5.1	Requirements and Design	10
1.5.2	Implementation (Coding)	10
1.5.3	Testing and Debugging	10
1.5.4	Deployment and Maintenance	11
1.6	Basic Program Design Principles	11
1.6.1	Modularity: Functions and Modules	11
1.6.2	Readability: Naming, Comments, Style Guidelines	12
1.6.3	Incremental Development and Version Control	12
1.7	Why Python?	13
1.7.1	Simplicity, Readability, and Versatility	13
1.7.2	Python's Place in the Low-Level vs. High-Level Spectrum	14
<b>2</b>	<b>Variables and Data Types</b>	<b>15</b>
2.1	Variable	15
2.1.1	Identifier	15
2.1.2	Rules for Identifiers	15
2.2	Data types	16
2.2.1	Integer ( <code>int</code> )	16
2.2.2	Floating-Point Number ( <code>float</code> )	16
2.2.3	String ( <code>str</code> )	16
2.2.4	Boolean ( <code>bool</code> )	17
2.3	Input and Output (I/O)	17

2.3.1	The <code>print()</code> Function	17
2.3.2	The <code>input()</code> Function	18
2.4	Type Casting	19
2.4.1	Why Do We Need Type Casting?	19
2.4.2	Common Casting Functions in Python	20
<b>3</b>	<b>Operators</b>	<b>21</b>
3.1	Introduction	21
3.2	Arithmetic Operators	21
3.2.1	Overview	22
3.2.2	Types	22
3.2.3	Order of Operations	25
3.2.4	Common Pitfalls	26
3.3	Relational (Comparison) Operators	26
3.3.1	Overview	27
3.3.2	Common Pitfalls	28
3.4	Assignment Operator	29
3.4.1	Simple Assignment Operator ( <code>=</code> )	29
3.4.2	Augmented Assignment Operator	29
3.4.3	Why Use Augmented Assignment?	31
3.5	Logical Operators	31
3.5.1	Overview	31
3.5.2	Logical AND ( <code>and</code> )	31
3.5.3	Logical OR ( <code>or</code> )	32
3.5.4	Logical NOT ( <code>not</code> )	33
3.6	Special Operators	34
3.6.1	Membership Operators	34
3.6.2	Identity Operators	34
<b>4</b>	<b>Error and Exception Handling</b>	<b>35</b>
4.1	Introduction	35
4.2	Syntax Errors	35
4.2.1	Common causes	35
4.2.2	Illustration	36
4.3	Runtime Errors	37
4.3.1	Common causes	37
4.3.2	Illustration	37
4.4	Logical Errors	38
4.4.1	Characteristics	38
4.4.2	Illustration	38
4.5	Exception Handling in Python	39
4.5.1	What is an Exception?	39
4.5.2	Why Handle Exceptions?	39
4.5.3	Basic Exception Handling Syntax	40
4.5.4	Catching Multiple Exceptions	41
4.5.5	The <code>else</code> Clause	41
4.5.6	The <code>finally</code> Clause	41

# Chapter 1

## Introduction

### 1.1 What Is a Computer?

#### 1.1.1 Basic Hardware vs. Software

A **computer** is essentially a machine that can be *programmed* to carry out sequences of instructions and calculations automatically. Every computer system is built from a combination of **hardware** and **software**. *Hardware* refers to the physical components of a computer that you can touch – such as the CPU, memory chips, hard drive, keyboard, or monitor. In contrast, *software* refers to the programs and instructions that run on the hardware and tell the computer what to do. For example, an application like a web browser or a game is software, whereas the processor and memory it uses are hardware.

#### 1.1.2 How CPU, Memory, Storage, and I/O Work Together

A computer's hardware components work in tandem to execute software instructions. The **Central Processing Unit (CPU)** – often called the “brain” of the computer – fetches and executes instructions from memory. The **memory** (RAM) holds data and program instructions that are actively being used, providing the CPU quick access to this information. The CPU and memory are connected via the motherboard and communicate over buses (electronic pathways). When you run a program, the program's instructions are loaded from long-term **storage** (like a hard disk or SSD) into memory, and the CPU reads those instructions from memory one by one to perform them. **Input/Output (I/O)** devices allow the computer to interact with the outside world: *input devices* (keyboard, mouse, etc.) let users or other systems send data into the computer, while *output devices* (monitor, printer, speakers) display or transmit the results of the computer's computations to users. All these parts are coordinated by the system's architecture to work together – for instance, when you press a key (input), the CPU processes that input, possibly stores or retrieves data from memory or storage, then outputs a result to your screen (output).

#### 1.1.3 Types of Computers

Computers are often categorized by the scale of tasks they're designed to perform and the environments in which they operate. At one end of the spectrum are Personal Computers, built for individual users handling everyday tasks such as web browsing, document editing, and media playback. In the middle lie Workstations, which pack extra processing muscle and specialized hardware for demanding technical or creative work. At the top are Mainframes, colossal systems

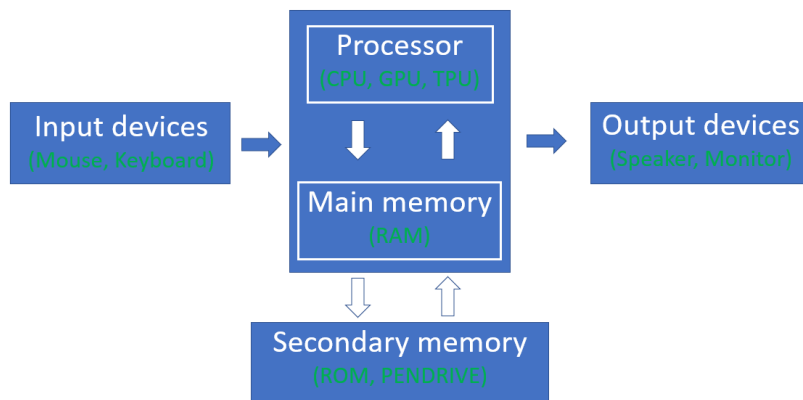


Figure 1.1: Block Diagram of a Computer

engineered to support thousands of users and process vast volumes of critical data with near-perfect reliability.

### Classification Criteria

When deciding whether a machine is a PC, a workstation, or a mainframe, engineers look at several factors:

- **Processing Capability:** Personal Computers use consumer-grade CPUs and possibly a modest GPU. Workstations step up to server-class processors (often with many cores or specialized accelerators) and professional GPUs. Mainframes employ proprietary multi-chip architectures that deliver massive parallelism.
- **Memory and Storage:** A typical PC has a few gigabytes of RAM and one or two terabytes of disk or SSD. Workstations raise that to tens or hundreds of gigabytes of error-correcting (ECC) memory and high-speed storage arrays. Mainframes boast terabytes of memory and petabytes of redundant, enterprise-grade storage.
- **Reliability and Uptime:** Desktop PCs are built for an average of a few years of consumer-level use. Workstations add features like ECC memory, RAID disk arrays, and robust cooling to reduce downtime. Mainframes target “five nines” availability (99.999).
- **User Capacity:** One person works at a PC at a time. A workstation typically serves a single specialized user but can sometimes share resources. A mainframe simultaneously supports hundreds or thousands of users running business-critical applications.
- **Cost and Support:** PCs are the most affordable category, ranging from a few hundred to a couple thousand dollars. Workstations cost several thousand to tens of thousands, reflecting their enhanced hardware and warranties. Mainframes carry six-figure price tags plus ongoing maintenance contracts.

### Personal Computers

Personal Computers are optimized for individual productivity. A desktop PC—the classic tower or small-form-factor box—lets users upgrade components like RAM, storage, or graphics cards. A

laptop integrates keyboard, display, battery, and trackpad into one portable package; its design balances performance against battery life and weight. The all-in-one PC (for example, Apple's iMac) houses all system components behind the display, offering a sleek footprint at the expense of internal expandability. Finally, tablets or convertibles run mobile or desktop operating systems on touchscreen hardware, trading raw power for extreme portability and instant-on convenience.

## Workstations

Workstations sit between PCs and mainframes in both power and price. An engineering workstation typically includes multi-core server CPUs (such as Intel Xeon or AMD Threadripper), large banks of ECC-protected RAM, and high-performance storage—ideal for tasks like finite-element analysis or 3D CAD modeling. A graphics or video workstation pairs professional-grade GPUs (e.g., NVIDIA Quadro or AMD Radeon Pro) with NVMe SSD arrays, enabling real-time rendering, animation, and video editing workflows. In the era of data-driven science, data science/AI workstations bring together powerful CPUs, multiple GPUs connected by high-bandwidth interconnects, and ample memory so you can train machine-learning models right at your desk.

## Mainframes

At the enterprise level, mainframes power the backbones of banking, insurance, government, and large-scale commerce. Transaction-processing mainframes handle enormous volumes of small, quick operations—such as ATM withdrawals or online retail purchases—without missing a beat. Batch-processing mainframes run scheduled jobs like payroll calculations or billing cycles, where hundreds of millions of records are processed in a single, windowed batch run. For extreme continuity, high-availability mainframes use redundant hardware clusters and advanced failover software so that even if one component fails, the system as a whole keeps running with virtually zero downtime.

## 1.2 What Is Programming?

Programming is the process of giving a computer precise instructions to perform tasks. Because computers cannot think for themselves, a programmer must write out step-by-step commands in a form the machine can follow exactly. In essence, programming involves designing a solution to a problem and then implementing that solution in a **program** – a set of instructions executed by the computer. We often start by formulating an **algorithm**: a sequence of steps or a procedure to solve a specific problem or accomplish a task. Algorithm consists of three parts:

- **Input(s):** The data or resources the algorithm requires.
- **Output(s):** The result(s) the algorithm produces.
- **Steps:** A precise, numbered list of actions that transform the inputs into the outputs.

By defining an algorithm in this way, we eliminate ambiguity and ensure that both humans and computers can follow the same procedure. Once an algorithm is defined, programming means translating that algorithm into a program using a programming language so that the computer can execute it. Good programming also requires being very exact – computers will do exactly what they are told, no more and no less. This means a programmer needs to think through every detail of the task. The process from conceptual *algorithm* to actual *program* involves careful planning (sometimes using pseudocode or flowcharts to sketch out the logic) and then coding, testing, and refining the instructions until the computer behaves as expected.

**Algorithm: MakeTea****Input(s):**

- Cold or room-temperature water, Tea leaves (or tea bag), Kettle or pot, Cup, (Optional) Sugar and/or milk

**Output(s):**

- A hot cup of tea, ready to drink

**Steps:**

1. Pour water into the kettle or pot.
2. Heat the water until it reaches a rolling boil.
3. Place the tea leaves or tea bag into the pot (or directly into the cup).
4. Pour the boiling water over the tea.
5. Let the tea steep for the desired time (typically 3–5 minutes).
6. Remove the tea leaves or tea bag from the liquid.
7. (Optional) Add sugar and/or milk, then stir until dissolved.
8. Serve the tea in a cup.

### 1.3 Programming Languages: Low-Level vs. High-Level

Not all programming languages are the same; they exist on a spectrum from low-level to high-level. The difference lies in how close the language is to the computer's hardware instructions versus how close it is to human-friendly abstractions.

#### 1.3.1 Low-Level Languages (Machine Code and Assembly)

**Low-level languages** are languages that are very close to the hardware's native language (machine code). *Machine code* is the most fundamental level of code – binary digits (0s and 1s) that directly control the CPU's operations. This is extremely difficult for humans to read or write. *Assembly language* is a small step above machine code: it uses short textual mnemonics (like **ADD** for addition, **MOV** for moving data) instead of raw binary, but each assembly instruction still corresponds very closely to a single machine instruction. Low-level code gives the programmer very fine-grained control over the hardware (for example, managing memory addresses and CPU registers directly). However, it's not intuitive for humans – even simple tasks require many detailed instructions. For instance, printing the word “HELLO” in pure binary machine code would require a sequence of binary values representing each operation and character, something like 01001000 01000101 01001100 01001100 01001111 (which is not at all obvious to a human). In assembly, it would be somewhat more readable but still quite complex, involving multiple instructions to output each character. Low-level languages are generally used when maximum performance or direct hardware control is needed (such as writing device drivers or embedded systems software).

### 1.3.2 High-Level Languages (Python, Java, C++)

**High-level languages**, on the other hand, are designed to be closer to human language and abstract away much of the hardware detail. Languages like Python, Java, and C++ allow a programmer to accomplish tasks with more succinct and readable code. For example, to display the word “HELLO” on the screen in a high-level language, you might just write a single statement like `print("HELLO")` in Python, which is far easier to understand and write than the equivalent low-level code. High-level languages manage many aspects automatically – things like memory management, complex CPU instructions, etc., are handled behind the scenes. This makes developers more productive and programs more portable, as the same high-level code can often run on different types of machines with minimal or no modification. The trade-off is that high-level languages may be less efficient in terms of performance and may not allow as much fine control over the hardware. In practice, however, high-level languages are by far the most commonly used for application development because they greatly speed up development and reduce the chance of errors.

### 1.3.3 Trade-offs: Control vs. Productivity

The choice between low-level and high-level languages involves a trade-off between control and productivity. Low-level languages offer **greater control** over hardware and potential for highly optimized performance, but writing code in them is labor-intensive and error-prone. High-level languages prioritize **developer productivity** and ease of use, at the cost of some control and efficiency. In other words, it typically takes much less time and code to develop a program in a high-level language, and that code will be easier to read and maintain, whereas a low-level language might yield a faster or more optimized program but require significantly more effort to write. Modern software development often finds a balance: critical low-level components (where performance or hardware access is crucial) might be written in C or assembly, while higher-level application logic is written in languages like Python or Java. Understanding this spectrum helps programmers choose the right tool for a given task.

## 1.4 Program Translation

When we write a program in a high-level language, the computer cannot directly understand it. The code must be translated into machine instructions that the hardware can execute. There are several ways this translation happens, and it’s important to know the distinctions:

### 1.4.1 Assemblers (for Assembly Language)

An **assembler** is the simplest translator, used for low-level assembly language. Assembly language uses human-readable mnemonics but is specific to a CPU’s instruction set. The assembler program takes each assembly instruction and converts it into the corresponding machine code (binary instruction) for that CPU. Essentially, it has a one-to-one (or very close) mapping from assembly to machine code. The result of assembly is an executable machine code program. For example, if an assembly code has a line to add two numbers, the assembler will translate that into the exact binary opcode that tells the CPU to perform addition. Assemblers produce very efficient code and give access to hardware, but the programmer must manage every detail.



### 1.4.2 Compilers (e.g., for C/C++) vs. Interpreters (e.g., for Python)

A **compiler** is a program that translates code written in a high-level language into machine code (or an intermediate low-level form) *before* the program is run. It takes the entire source code, processes it, and produces an executable file or binary. For example, C and C++ are traditionally compiled languages: you write the code, run a compiler, and it outputs an executable program. This machine code can then be executed directly by the computer's CPU. The compilation process typically involves multiple stages (parsing the code, optimizing, etc.), but the key point is that after compilation, you have a standalone machine-code program. Compiled programs generally run very fast because the translation to machine instructions is done upfront.

An **interpreter**, by contrast, executes a high-level program by reading it one piece at a time and performing the operations on the fly, without a separate compile step. In an interpreted language (like standard Python, Ruby, or JavaScript), you feed the source code to an interpreter, and it handles each instruction step-by-step: read a line or statement, translate it (for example, into some internal form), and execute it immediately, then move on to the next. This means you don't get a separate binary file; the translation and execution happen together. Interpreted execution can be more flexible (you can often run code interactively, and it's easy to test changes quickly), but it tends to be slower than running compiled code because translation happens as the program runs, and some optimizations may not be as aggressive.

To illustrate, consider a simple program that calculates a result. In a compiled language like C, you would compile the program once (catching many errors at compile time), and then you can run the resulting executable as many times as needed efficiently. In an interpreted language like Python, you would run the Python interpreter on your script each time; the interpreter checks each line and executes it, which adds overhead during execution. Many scripting and beginner-friendly languages use interpreters to allow rapid development and debugging (since you can run code immediately without a compilation step).

### 1.4.3 Bytecode and Just-In-Time Compilation (JIT)

Modern language implementations often blend these approaches for efficiency. Some languages compile source code into an intermediate form known as **bytecode**. Bytecode is a lower-level, platform-independent code that is not as detailed as machine code but is easier to translate than high-level code. For example, Java is compiled into bytecode (.class files) which run on the Java Virtual Machine (JVM). Similarly, Python (when using the standard CPython interpreter) compiles source files into bytecode (with .pyc files) which its virtual machine then interprets. Bytecode improves performance over interpreting raw source because the high-level parsing is done ahead of time, and it also allows portability (the same bytecode can run on any machine that has the virtual machine for that language).

To further improve performance, many runtime systems use **Just-In-Time (JIT) compilation**. JIT compilers take the bytecode (or other intermediate representation) and compile parts of it into machine code *at runtime*, i.e., while the program is running. The idea is to get the best of both worlds: the flexibility of interpretation and the speed of compilation. For instance, the Java JVM will interpret the bytecode at first, but as it identifies “hot” code paths that run frequently, it will JIT-compile those into native machine code for faster execution. This means that after a short warm-up period, a Java program can run nearly as fast as a fully compiled program, because its most critical parts have been turned into optimized machine code by the JIT. Python itself has alternative implementations (like PyPy) that use JIT compilation to accelerate execution. In summary, program translation might involve pure ahead-of-time compilation, pure interpretation,

or hybrid approaches (bytecode interpretation and JIT), each with advantages in terms of speed, portability, and development ease.

## 1.5 The Software Development Life Cycle

Writing a program (or developing any software) is not just about writing code. It involves a series of stages collectively known as the **Software Development Life Cycle (SDLC)**. The SDLC provides a structured approach to building software, ensuring quality and manageability throughout the process. We will outline the key phases of SDLC and what happens in each:

### 1.5.1 Requirements and Design

The first step is determining **what the software needs to do** and **how it should be designed to do it**. In the *requirements* phase, developers (often along with stakeholders or clients) gather and document the specifications: what problem is the software solving? What are the features and constraints? This often results in a Software Requirements Specification (SRS) document or a clear list of requirements. Once the team understands the goals, the next step is *design*. In the design phase, developers plan the structure of the software and how it will meet the requirements. This can include creating high-level architecture diagrams, flowcharts, and using pseudocode to sketch out algorithms before actual coding. For example, if you were designing a simple program like a calculator app, the design might include a flowchart of how user input flows through the system and a description of modules or functions (addition module, display module, etc.). The guiding principle in this stage is to figure out the blueprint of the software – breaking the problem into smaller components, deciding how those components will interact, and ensuring that all requirements will be addressed by the design. Good design often uses techniques like modularization (which we discuss later) to keep the system organized. By the end of this stage, the team should have a clear plan (perhaps documented as specifications, pseudocode, or diagrams) for how to build the software.

### 1.5.2 Implementation (Coding)

In the implementation phase, the developers actually **write the code** according to the design. This is the programming part: using a programming language to create the functions, classes, and modules that were planned in the design stage. The goal is to translate the design into a working software product. Developers will follow best practices for coding and use the chosen technology stack to build the application. During implementation, it's common to do incremental coding and frequent testing of small parts to ensure each part works correctly as it's built. The result of this phase is a working software system (or at least individual components of it) built according to the design. If we use the calculator app example, this is where developers write the actual code for the add, subtract, multiply, divide functionalities, and the user interface code, etc., following the plan made earlier. It's important that the code is written with care for *readability* and *maintainability* so that future developers (or the same developers later) can understand and modify it. The output of this phase is typically one or more program files (source code) that implement the required features.

### 1.5.3 Testing and Debugging

Once some code is written, it must be **tested** to ensure it works as intended. In the testing phase, developers (and possibly dedicated QA testers) run the program with various inputs and scenarios to verify that each requirement is met and to catch any **bugs** (errors or unexpected behavior). Testing can be broken down into multiple levels:

- *Unit testing*: checking individual components or functions for correctness.
- *Integration testing*: checking that different parts of the system work together properly.
- *System testing*: checking the entire integrated application against the requirements.
- *User acceptance testing*: sometimes, having actual end-users or clients test to see if the system meets their needs.

When tests reveal a problem, the process of **debugging** begins – this means locating the source of the error in the code and fixing it. Debugging can be challenging, as it requires understanding what the code is actually doing versus what it was intended to do. Tools like debuggers or simply strategic printouts/logs are used to inspect the program’s state and find where it goes wrong. This stage is critical: it’s much cheaper and easier to fix problems before software is deployed to real users. A well-tested and debugged program will be more reliable and maintainable. Testing and debugging continue iteratively until the developers are confident that the software is correct and stable.

#### 1.5.4 Deployment and Maintenance

After testing is successful and the software is deemed ready, it is **deployed** – delivered to the end users or moved into a production environment. Deployment might mean publishing a downloadable application, releasing an update through an app store, or installing a system on a server for users to access. Sometimes deployment is done in stages (for example, a beta release to a small group before a full launch).

Once users begin using the software, the project enters the **maintenance** phase. Maintenance involves updating the software to fix any new bugs that appear in real-world use, improving features, or adapting the software to new requirements or environments over time. No software is ever truly “finished” – operating systems update, user needs evolve, and new security vulnerabilities might be discovered, so developers often need to issue patches or new versions. Maintenance also covers activities like optimizing performance after seeing how the system behaves in production or refactoring code for better clarity and extensibility. Essentially, maintenance ensures the software continues to meet user needs and run smoothly after its initial release. This phase can last as long as the software is in use.

It’s worth noting that there are various models such as Waterfall, Agile, etc., that organize these stages differently – some are strictly sequential, while others repeat these stages in cycles – but nearly all software development will include these fundamental activities of planning, coding, testing, and maintenance in some form.

### 1.6 Basic Program Design Principles

To write good software (not just code that “works,” but code that is robust, readable, and maintainable), programmers follow several important design principles. Here we discuss a few fundamental ones: **modularity**, **readability**, and practices like **incremental development** and **version control**.

#### 1.6.1 Modularity: Functions and Modules

**Modularity** is the design principle of breaking a program into separate parts (modules) such that each part handles a specific piece of the overall functionality. In practice, this means using functions,

classes, and modules (files or libraries) to encapsulate different functionality. For example, if you are writing a game, you might have one module or section of code for the player input, another for the game physics, another for rendering graphics, etc. Each module can be developed and understood independently, as long as the modules have well-defined interfaces (ways they interact with each other). This separation of concerns makes programs easier to understand and maintain because you can focus on one part at a time and because changes in one module (as long as the interface remains the same) won't heavily impact other parts. Functions are a basic unit of modularity: each function performs a well-defined task, and by calling functions, you can avoid repeating code and make the program structure clearer. In Python, for example, you can group related functions into a module (a .py file) and reuse that module in different programs. Overall, modular design leads to code that is more organized, reusable, and testable.

### 1.6.2 Readability: Naming, Comments, Style Guidelines

Code is read much more often than it is written. **Readability** refers to how easily a human (including the original programmer, weeks or months later) can understand the code. This is vital, because code that is hard to read tends to breed bugs and is difficult to extend or fix. In fact, programmers spend a majority of their time reading and understanding existing code rather than writing new code. Unreadable or sloppy code can lead to mistakes, inefficiencies, and even duplicate code when future developers can't tell that a feature already exists and end up reimplementing it.

Key practices to enhance readability include:

- **Meaningful naming:** Use clear, descriptive names for variables, functions, and other identifiers. For instance, a variable named `total_price` is more informative than `x`. Good naming makes the code self-documenting.
- **Comments and documentation:** Provide comments to explain non-obvious parts of the code or the rationale behind certain decisions. For example, a brief comment above a complex block of code can help others (or your future self) understand its purpose. However, comments should supplement clear code – they are not a substitute for it.
- **Consistent style and formatting:** Following a style guide (like PEP 8 for Python) ensures consistency in things like indentation, bracket placement, spacing, etc. Consistent indentation and formatting help visually structure code blocks and make it easier to parse code at a glance.
- **Modular decomposition:** As mentioned, breaking code into functions and modules not only helps design but also readability. Each function should ideally do one thing, and do it well (the Single Responsibility Principle), which makes it easy to understand.

In essence, writing readable code means writing code for *other humans* to understand, not just for the computer to execute. Following naming conventions, writing clear comments, and adhering to coding style guidelines are all practices that contribute to code quality and maintainability.

### 1.6.3 Incremental Development and Version Control

Developing software is an iterative process. **Incremental development** is a practice where you build and test your program in small pieces, gradually adding functionality, rather than trying to write the entire program in one go. This approach involves writing a small amount of code, then running it to test and verify it works, then continuing to add more. By proceeding in increments, you catch errors early and avoid being overwhelmed by large amounts of new code that haven't been

tested. For example, if you're creating a simple application, you might first implement the core function with a simple output and test it. Once that works, you add the next feature, test again, and so forth. This approach not only makes debugging easier (since you know any new problems are likely in the code you just added), but also gives you a working program (albeit with limited features) at each step. It aligns with the idea of “building up” a program iteratively and is especially useful for beginners to gain confidence and for large projects where continuous feedback is valuable.

Another essential practice in modern software development is using a **version control system** (VCS) such as Git. Version control systems help manage changes to the source code over time and facilitate collaboration between multiple developers. With version control, every change you make to the code is tracked. You can commit changes with a message describing what was done, and the system keeps a history of all these commits. This means you can revert to a previous version if a new change introduces a bug, or compare different versions of code to find out when a bug was introduced. For team projects, VCS allows multiple people to work on different parts of the code simultaneously without overriding each other's work – changes can be merged together systematically. It also provides a level of backup: the code repository exists on developers' machines and often on a remote server, so the risk of losing code is minimized. In short, version control is a safety net and an organizational tool; it enforces discipline in saving work and documenting changes, and is indispensable in professional software development. Even for an individual working solo, using version control is considered a good practice to keep their project history and progress well-managed.

## 1.7 Why Python?

Python is often recommended as an excellent language for beginners and experts alike, and it's worth understanding **why** – especially in the context of learning programming fundamentals. Python is a high-level programming language, and it exemplifies many of the concepts we've discussed in terms of ease-of-use and abstraction.

### 1.7.1 Simplicity, Readability, and Versatility

Python is known for its **simple and readable syntax**. This means that Python code often looks closer to pseudocode or even plain English, making it easier to learn and understand. For example, to get the length of a list in Python, you might write `len(my_list)`, and to print something, you just write `print("something")`. There are minimal unnecessary symbols or complicated structures for a beginner to worry about. Python enforces indentation for blocks of code (like loops or function definitions), which naturally leads to clean and consistently formatted code. These features lower the barrier to entry for new programmers and allow you to focus on learning programming concepts rather than complex syntax. In fact, Python's design philosophy (as described in “The Zen of Python”) emphasizes readability and simplicity.

Despite its simplicity, Python is a very **powerful and versatile** language. It comes with a large standard library and has a vast ecosystem of third-party libraries (via the Python Package Index) for almost any task: web development, data analysis, artificial intelligence, automation, scientific computing, and more. This means that once you learn the Python basics, you can apply it to many different domains without having to learn a new language for each domain. For instance, a single Python program could grab data from the web, crunch numbers using a math library, and then output a graph – all using libraries readily available to Python. Because of this versatility, Python is used by companies and researchers in many fields, and Python skills are highly in demand.

Another reason Python is a great learning language is its **community and support**. There is extensive documentation, and a large community of Python users means that if you encounter a problem, it's likely someone else has as well – and you can find help on forums like Stack Overflow or through countless tutorials. Python's community also contributes to its many libraries and frameworks, constantly improving the tools available to programmers.

### 1.7.2 Python's Place in the Low-Level vs. High-Level Spectrum

As discussed earlier, Python is a **high-level language**. It abstracts away most of the low-level details of the computer. For example, Python manages memory for you (via garbage collection), it doesn't require you to manually handle addresses or allocate memory as you might in C. It allows you to write code without thinking about the specifics of the hardware. This makes Python highly productive – you can often implement a feature in Python in a fraction of the number of lines it would take in a lower-level language. The trade-off is performance: Python code is generally slower to run than equivalent code in a low-level language like C, because Python code is executed by an interpreter and does a lot of work behind the scenes. However, for many applications, especially at the learning stage, this is not a problem. Computers are fast, and Python is “fast enough” for tasks like scripting, web development, or prototyping. Moreover, when performance is critical, Python allows integration with low-level languages (for example, heavy computations can be offloaded to libraries written in C/C++ which Python can call).

In terms of spectrum, Python sits at the high-level end, emphasizing ease of use over fine-grained control. It means that as a learner, you can focus on *concepts* like loops, functions, and data structures without getting bogged down by complex syntax or machine-level details. Python also exemplifies many good programming practices – its emphasis on readability and the enforced indentation teach you to write clean code from the start.

To summarize, we learn Python not only because it's beginner-friendly (simple and readable) but also because it's a language with real-world strength: it's used in industry and academia, and it allows a gentle introduction to programming concepts that scale up to serious applications. Python serves as a gentle stepping stone into the wider world of programming: once you understand the concepts using Python, you can transfer those concepts to other languages (the differences will mostly be in syntax and level of manual control). Meanwhile, you can also accomplish a lot directly with Python, making it a practical choice for projects big and small.

## Chapter 2

# Variables and Data Types

### 2.1 Variable

A *variable* in Python is a name that refers to a value stored in memory. Think of it as a labeled container that holds data. Python is dynamically typed, so you don't declare the type explicitly—variables can be rebound to objects of different types at runtime.

#### Variable Creation and Assignment

```
value_x = 10      # value_x refers to an integer object with value 10
name = "Alice"    # name refers to a string object "Alice"
pi = 3.14159      # pi refers to a float object
```

#### 2.1.1 Identifier

An *identifier* is the name you choose for variables, functions, classes, modules, etc. It must follow Python's naming conventions and syntax rules.

#### 2.1.2 Rules for Identifiers

Python enforces several rules for valid identifiers:

1. **Start with a letter or underscore:**

- Valid: `_temp`, `varName`, `tax_1`.
- Invalid: `1_tax` (starts with digit).

2. **Contain only letters, digits, or underscores. No spaces or special characters:**

- Valid: `user1`, `_data_set2`.
- Invalid: `first name`, `user-name`, `count$`.

3. **Case-sensitive:** `Data`, `DATA`, and `data` are distinct.

4. **Cannot be a Python keyword** (Inbuilt identifiers): Reserved words such as `if`, `for`, `while`, `import`. Invalid: `class = 5`.

5. **Unlimited length** (but keep names reasonable).

## 2.2 Data types

A data type is simply a way of telling Python (and you!) what kind of information you're working with, so that the computer knows:

1. How much space to set aside in memory?
2. What operations make sense? (e.g. you can add two numbers but you can't add two pieces of text instead you can concatenate them)

### Analogy: sorting school supplies

Imagine you've got bins labeled "Pen", "Erasers", "Rulers", and "Notebooks". Before you drop an item into a bin, you look at what it is:

- If it's a pen, it goes in the Pens bin.
- If you tried to put a notebook into the Erasers bin, someone would stop you!

In programming, data types are like those labeled bins. They keep your information organized so that Python knows:

- You can add two pens to get more pens.
- You can't mix a ruler with an eraser in the same operation.

### 2.2.1 Integer (int)

**Definition:** Whole numbers without any fractional part.

**Analogy:** Like counting the number of apples in a basket—you can't have 3.5 apples (unless you cut one!).

#### Assigning integers to variables

```
num_students = 25
year = 2025
```

### 2.2.2 Floating-Point Number (float)

**Definition:** Numbers that have a decimal point (can represent fractions).

**Analogy:** Like measuring 3.75 liters of juice—you often need decimals in real life.

#### Assigning floats to variables

```
price_per_hour = 12.50
temperature = 98.6
```

### 2.2.3 String (str)

**Definition:** A sequence of characters (letters, numbers, symbols) **enclosed in quotes**.

**Analogy:** Like writing words or sentences on a chalkboard—anything you can type.



**Assigning strings to variables**

```
first_name = "Alex"
greeting = 'Hello, world!'
```

**2.2.4 Boolean (bool)**

**Definition:** A value that is either `True` or `False`.

**Analogy:** A light switch—it's either on (`True`) or off (`False`).

**Assigning boolean to variables**

```
is_raining = False
passed_test = True
```

**Summary**

Data Type	Example Value	Use Case	Size (bytes)
<code>int</code>	42	Counting items, indexing	28
<code>float</code>	3.14	Measurements, prices	24
<code>str</code>	"Hello"	Text, messages	52
<code>bool</code>	<code>True</code>	Yes/no flags, conditional checks	28

Table 2.1: Basic data types

**2.3 Input and Output (I/O)**

When you write a Python program, it doesn't do much if it just sits there silently! Most useful programs need to communicate with the person using them:

- Input is how your program receives information from the user (for example, asking for their name or a number).
- Output is how your program shares information back (for example, printing a greeting or the result of a calculation).

Together, input and output (often called **I/O**) let you build interactive programs—ones that can ask questions, take in answers, do something with those answers, and then tell the user what happened.

**2.3.1 The `print()` Function**

**Purpose:** Display text (or other values) on the screen.

**Syntax:**

```
print(value1, value2, ..., sep = ' ', end = '\n')
```

where,

**value1, value2, ...** are the things you want to display.

**sep** is the string placed between values (default is a space).

**end** is what's printed after all values (default is a newline).

**Note:** **print** is a built-in function. **sep** and **end** are keywords.

#### Example 1:

```
print(10, 20, 30, sep = ' ', end = '\n')
print(10, 20, 30)
print(10, 20, 30, sep = ',')
print(10, 20, 30, sep = '') # no space between the values
print("Hello, world!")
print("Sum of 2 and 3 is", 2 + 3)
```

#### Output:

```
10 20 30
10 20 30
10,20,30
102030
Hello, world!
Sum of 2 and 3 is 5
```

#### Example 2:

```
print("Zoro!", end = '\n')
print(10, 20, 30, end = ' ') # line ends with a space
print(11, 22, 33)
```

#### Output:

```
Zoro
10 20 30 11 22 33
```

### 2.3.2 The input() Function

**Purpose:** Pause the program and ask the user to type something; then return what they typed as a **string**.

#### Syntax:

```
user_text = input(prompt)
```

where,

**prompt** is the message shown to the user enclosed in quotes.

**user\_text** stores whatever the user types (until they press Enter) as a string.

**Example 1:**

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

**Output:**

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

**Step-by-step:**

1. Python displays `What is your name?` and waits.

**Output:**

```
What is your name?
```

2. You type, for example, `Alice` and press Enter.

**Output:**

```
What is your name? Alice
```

3. The variable `name` now holds the string `"Alice"`.
4. `print("Hello, " + name + "!")` prints `Hello, Alice!`.  
Note: `+` is used to concatenate strings.

## 2.4 Type Casting

In programming, type casting (also called type conversion) is the process of changing a value from one data type to another. For example, you might want to turn the text `"42"` into the number `42` so you can do arithmetic with it. Type casting helps make your programs more flexible and prevents errors when you mix different kinds of data.

### 2.4.1 Why Do We Need Type Casting?

Imagine you ask the user for their age, even if they type `16`, Python stores it as a string (`"16"`).

**Example 1:**

```
age = input("How old are you? ")
print(type(age))
```

**Output:**

```
How old are you? 16
<class 'str'>
```

You can't perform arithmetic operations unless you first convert into the integer/float. That's where type casting comes in.

**Example 2:**

```
text = input("How old are you? ")
age = int(text)
print(type(age))
print("Next year you will be", age + 1)
```

**Output:**

```
How old are you? 16
<class 'int'>
Next year you will be 17
```

### 2.4.2 Common Casting Functions in Python

Function	Converts To	Example Usage
int()	Integer	int("42") → 42; int(42.6) → 42
float()	Float	float("3.14") → 3.14; float(5) → 5.0
str()	String	str(99) → "99"
bool()	Boolean	bool(0) → False; bool("hi") → True

Table 2.2: Casting Functions

# Chapter 3

## Operators

### 3.1 Introduction

In Python (as in most programming languages), an operator is a special symbol or keyword that tells the interpreter to perform a specific computation or action on one or more values (called operands). You can think of operators as the “verbs” in the language—they take inputs, do something with them, and produce an output.

**Key points about operators:**

- Operands are the values or variables you apply the operator to.
- Unary operators work on a single operand (e.g. `-x` negates `x`).
- Binary operators work on two operands (e.g. `x + y` adds `x` and `y`).
- Operators always return a result, which you can assign to a variable or use directly in an expression.

**Why operators matter:**

- They let you compute numeric results (like adding `3 + 5`).
- They let you compare values (like checking if `age >= 18`).
- They let you combine logical conditions (like `is_member and has_paid`).
- They let you manipulate data structures (e.g. checking membership with `'a' in my_list`).

### 3.2 Arithmetic Operators

In programming, arithmetic operators let you perform mathematical calculations on numbers. Just as you use `+`, `-`, `×`, and `÷` on paper, Python provides a set of symbols that tell the computer how to combine or transform numeric values. Understanding these operators is essential for writing programs that handle everything from simple sums to complex formulas.

Operator	Name	Description
+	Addition	Adds two numbers
-	Subtraction	Subtracts the right number from the left
*	Multiplication	Multiplies two numbers
/	Division	Divides left by right, results in a floating-point number
//	Floor Division	Divides left by right, then rounds down to the nearest integer
%	Modulus (Remainder)	Returns the remainder after division
**	Exponentiation	Raises left number to the power of the right number
()	Parentheses (Grouping)	Changes the order of operations

Table 3.1: Arithmetic Operators

### 3.2.1 Overview

### 3.2.2 Types

#### 1. Addition (+)

**What it does:** Combines two numbers into their sum.

Example:

```
value_1 = 7
value_2 = 5
result = value_1 + value_2
print("The result of", value_1, "+", value_2, "is", result)
```

Output:

```
The result of 7 + 5 is 12
```

#### 2. Subtraction (-)

**What it does:** Takes one number away from another.

Example:

```
value_1 = 7
value_2 = 5
result = value_1 - value_2
print("The result of", value_1, "-", value_2, "is", result)
```

Output:

```
The result of 7 - 5 is 2
```

### 3. Multiplication (\*)

**What it does:** Finds the product of two numbers.

Example:

```
value_1 = 7
value_2 = 5
result = value_1 * value_2
print("The result of", value_1, "*", value_2, "is", result)
```

Output:

```
The result of 7 * 5 is 35
```

### 4. Division (/)

**What it does:** Divides the first number by the second, always returning a float.

Example:

```
value_1 = 8
value_2 = 2
result = value_1 / value_2
print("The result of", value_1, "-", value_2, "is", result)
```

Output:

```
The result of 8 / 2 is 4.0
```

**Note:** Even though 8 divided by 2 is 4, Python shows it as 4.0 because division yields a floating-point number.

### 5. Floor Division (//)

**What it does:** Divides and then “floors” (rounds down) the result to the nearest integer.

Example:

```
value_1 = 9
value_2 = 2
result = value_1 // value_2
print("The result of", value_1, "//", value_2, "is", result)
```

Output:

```
The result of 9 // 2 is 4
```

**Note:** 9 divided by 2 is 4.5, but floor division drops the .5 to give 4.

## 6. Modulus (%)

**What it does:** Gives the remainder after division.

Example:

```
value_1 = 9
value_2 = 2
result = value_1 % value_2
print("The result of", value_1, "%", value_2, "is", result)
```

Output:

```
The result of 9 % 2 is 1
```

**Note:** 9 divided by 2 leaves a remainder of 1.

## 7. Exponentiation (\*\*)

**What it does:** Raises one number to the power of another.

Example:

```
base = 3
exponent = 4
result = base ** exponent
print("The result of", base, "to the power of", exponent, "is",
      result)
```

Output:

```
The result of 3 to the power of 4 is 81
```

## 8. Grouping with Parentheses (())

**What it does:** Forces operations inside parentheses to be computed first.

Example:

```
result1 = 2 + 3 * 4
result2 = (2 + 3) * 4
print("The result of 2 + 3 * 4 is ", result1)
print("The result of (2 + 3) * 4 is ", result2)
```

Output:

```
The result of 2 + 3 * 4 is 14
The result of (2 + 3) * 4 is 20
```



### 3.2.3 Order of Operations

Python follows the same precedence rules as standard mathematics:

1. Parentheses
2. Exponentiation
3. Multiplication, Division, Floor Division, Modulus (left to right)
4. Addition and Subtraction (left to right)

Use parentheses to make your intentions clear and to override this default order.

Example:

```
value_1 = 5
value_2 = 2
value_3 = 3

result1 = value_1 + value_2 * value_3 ** 2 // 4 % 3

result2 = ((value_1 + value_2) * (value_3 ** 2)) // (4 % 3)

print("result1:", result1)
print("result2:", result2)
```

Output:

```
result1: 6
result2: 63
```

**Breakdown for result1:**

1.  $value\_3 ** 2 \rightarrow 3^2 = 9$
2.  $value\_2 \times 9 \rightarrow 2 \times 9 = 18$
3.  $18 // 4 \rightarrow 4$  (floor division since  $18/4 = 4.5$ )
4.  $4 \% 3 \rightarrow 1$
5.  $value\_1 + 1 \rightarrow 5 + 1 = 6$

**Breakdown for result2:**

1. Parentheses force  $(value\_1 + value\_2) = 7$  and  $value\_3 ** 2 = 9$
2. Multiply:  $7 \times 9 = 63$
3. Compute  $4 \% 3 = 1$
4. Floor divide:  $63 // 1 = 63$  (floor division by 1 leaves it unchanged)

### 3.2.4 Common Pitfalls

**1. Division vs. floor division:** Remember `/` always gives a float, while `//` gives an integer or float depending on the type of operands.

Example:

```
print("9/4 =", 9/4)
print("9//4 =", 9//4)
print("9.0/4 =", 9.0/4)
```

Output:

```
9/4 =2.25
9//4 =2
9.0/4 =2.0
```

**2. Negative numbers with floor division:** Floor division rounds down, not toward zero.

Example:

```
value_1 = -3
value_2 = 2
result = value_1 // value_2
print("The result of", value_1, "//", value_2, "is", result)
```

Output:

```
The result of -3 // 2 is -2
```

## 3.3 Relational (Comparison) Operators

In programming, relational operators (also called comparison operators) let you compare two values. Instead of doing math, these operators ask questions like “Is one number bigger than the other?” or “Are these two values equal?” Each comparison returns a **Boolean** result: **True** if the statement is correct, or **False** if it’s not. Relational operators are the **foundation of decision-making** in your programs, because they let you control what code runs next.

Operator	Name	What it Checks	Example
==	Equal to	Are the two values the same?	5 == 5 → True
!=	Not equal to	Are the two values different?	5 != 3 → True
>	Greater than	Is the left value larger than the right?	7 > 10 → False
<	Less than	Is the left value smaller than the right?	3 < 8 → True
>=	Greater than or equal to	Is the left value at least as big as the right?	4 >= 4 → True
<=	Less than or equal to	Is the left value at most as big as the right?	2 <= 1 → False

Table 3.2: Relational Operators

### 3.3.1 Overview

#### Example 1:

```
# Equal to (==)
actual_password = "SecurePass123"
entered_password = "securepass123"
is_password_correct = entered_password == actual_password

# Not equal to (!=)
required_subject = "Mathematics"
chosen_subject = "Physics"
is_different_subject = chosen_subject != required_subject

# Greater than (>)
student_score = 85
passing_score = 70
has_exceeded_passing = student_score > passing_score

# Print all results
print("Password correct?", is_password_correct)
print("Subject differs?", is_different_subject)
print("Score exceeds passing?", has_exceeded_passing)
```

#### Output:

```
Password correct? False
Subject differs? True
Score exceeds passing? True
```

**Example 2:**

```
# Less than (<)
current_temperature = 18.0 # in degree C
comfortable_temperature = 20.0
is_too_cold = current_temperature < comfortable_temperature

# Greater than or equal to (>=)
min_height_cm = 150
applicant_height_cm = 150
meets_height_requirement = applicant_height_cm >= min_height_cm

# Less than or equal to (<=)
max_team_members = 5
group_size = 4
within_team_limit = group_size <= max_team_members

# Chained comparison
age = 16
is_in_teen_range = 13 <= age <= 19

# Print all results
print("Too cold?", is_too_cold)
print("Meets height requirement?", meets_height_requirement)
print("Within team limit?", within_team_limit)
print("Is in teen age range?", is_in_teen_range)
```

**Output:**

```
Too cold? True
Meets height requirement? True
Within team limit? True
Is in teen age range? True
```

**3.3.2 Common Pitfalls**

1. **Accidental assignment vs. comparison:** In Python, = is assignment, == is comparison.
2. **Comparing different types and Floating-point precision issues:**

**Example:**

```
print(5 == "5") # integer 5 is not the same as string "5"
print(0.1 + 0.2 == 0.3) # Often False due to how floats are stored
```

**Output:**

```
False
False
```

**Note:** If you need to compare floats, consider checking if they're "close enough" using a small tolerance. (We will revisit when handling selection statements.)

### 3.4 Assignment Operator

In Python, assignment operators are used to store or update values in variables. The simplest assignment operator is `=`, which takes the value on the right and places it into the variable on the left. Beyond that, Python provides augmented assignment operators like `+=` and `*=` that combine an operation with assignment, letting you update a variable more concisely.

#### 3.4.1 Simple Assignment Operator (`=`)

It assigns the right-hand value to the variable on the left hand, i.e., first the value on the RHS of `=` is computed and then assigned to the variable on the LHS of `=`.

Example:

```
length = 5                # line 1
width = 4                  # line 2
area_rectangle = length * width # line 3
print("The area is", area_rectangle) # line 4
```

Output:

```
The area is 20
```

Breakdown for `area_rectangle`:

1. In **line 1**, **5** is assigned to the variable **length**.
2. In **line 2**, **4** is assigned to the variable **length**.
3. In **line 3**, the value of **length \* width** (**5 \* 4**) is first computed, i.e., **20**. Then, **20** is assigned to the variable **area\_rectangle**.

#### 3.4.2 Augmented Assignment Operator

Augmented assignment operators perform an operation and store the result back into the same variable.

Syntax:

```
variable <op>= expression
```

The above syntax is equivalent to:

```
variable = variable <op> expression
```

Augmented Operator	Equivalent To	What It Does
<code>+=</code>	<code>x = x + y</code>	<b>Add</b> y to x, then assign back to x
<code>-=</code>	<code>x = x - y</code>	<b>Subtract</b> y from x, then assign back to x
<code>*=</code>	<code>x = x * y</code>	<b>Multiply</b> x by y, then assign back to x
<code>/=</code>	<code>x = x / y</code>	<b>Divide</b> x by y, then assign back to x
<code>//=</code>	<code>x = x // y</code>	<b>Floor-divide</b> x by y, then assign back to x
<code>%=</code>	<code>x = x % y</code>	<b>Modulo</b> (remainder) of $x \div y$ , then assign
<code>**=</code>	<code>x = x ** y</code>	<b>Exponentiate</b> x to power y, then assign

Table 3.3: Augmented assignment operators

Example (Equivalent code and Output are added as comments):

```

points = 10
print("Initial points:", points)      # Initial points: 10

# 1. += (add and assign)
points += 5 # points = points + 5
print("After +=5:", points)          # After +=5: 15

# 2. -= (subtract and assign)
points -= 3 # points = points - 3
print("After -=3:", points)          # After -=3: 12

# 3. *= (multiply and assign)
multiplier = 2
points *= multiplier
print("After *=2:", points)          # After *=2: 24

# 4. /= (divide and assign)
points /= 4
print("After /=4:", points)          # After /=4: 6.0

# 5. //= (floor-divide and assign)
points //= 4
print("After //=4:", points)         # After //=4: 1.0

# 6. %= (modulus and assign)
points = 10
points %= 3
print("After %=3:", points)          # After %=3: 1

# 7. **= (exponentiate and assign)
value = 3
value **= 3
print("After **=3:", value)          # After **=3: 27

```

### 3.4.3 Why Use Augmented Assignment?

- Conciseness: `x += 1` is shorter than `x = x + 1`.
- Clarity: It clearly shows “update this variable by doing ...”.
- Performance: Under the hood, it can be slightly more efficient, especially with mutable types.

## 3.5 Logical Operators

In Python, logical operators let you combine or invert Boolean expressions—statements that are either **True** or **False**. They’re fundamental for making decisions in your code (e.g., “if this and that are both true...”). Python provides three logical operators:

- `and`
- `or`
- `not`

Each one follows simple rules (shown below in truth tables) and can be used in conditions, loops, and anywhere you need to test multiple criteria.

### 3.5.1 Overview

Operator	Meaning	Result when applied
<code>and</code>	<b>Logical AND</b>	True only if <b>both</b> operands are True
<code>or</code>	<b>Logical OR</b>	True if <b>at least one</b> operand is True (or both)
<code>not</code>	<b>Logical NOT</b>	True becomes False, and False becomes True

Table 3.4: Logical operators

### 3.5.2 Logical AND (`and`)

**What it does:** Returns True only if **both** operands are True.

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

Table 3.5: Truth table for the logical `and` operator

**Note:**

- **A** and **B** can be the result of a logical operators.
- **and** stops evaluation as soon as it sees a **False** (because the whole expression can't be **True** anymore).

**Example:**

```
has_id_card = True
has_ticket  = False
can_enter   = has_id_card and has_ticket
print("Can this person enter?", can_enter)

value_1 = 4 >= 4
value_2 = (10 % 3) == (3 // 2)
result  = value_1 and value_2
print("The result is:", result)
```

**Output:**

```
Can this person enter? False
The result is: True
```

**3.5.3 Logical OR (or)**

**What it does:** Returns **True** if at least one operand is **True**.

A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

Table 3.6: Truth table for the logical or operator

**Note:**

- **A** and **B** can be the result of a logical operators.
- **or** stops as soon as it sees a **True** (because the whole expression is already **True**).



**Example 1:**

```

has_id_card = True
has_ticket  = False
can_enter = has_id_card or has_ticket
print("Can this person enter?", can_enter)

value_1 = 4 >= 4
value_2 = (10 % 3) == (4 // 2)
result = value_1 and value_2
print("The result is:", result)

```

**Output:**

```

Can this person enter? True
The result is: False

```

**Example 2:**

```

age = 17
has_parent_permission = True

can_watch_movie = (age >= 18) or (age >= 13 and has_parent_permission)
print("Enery into movies:", can_watch_movie)

```

**Output:**

```

Enery into movies: True

```

**3.5.4 Logical NOT (not)**

**What it does:** Inverts the boolean value.

A	not A
True	False
False	True

Table 3.7: Truth table for the logical not operator

**Note:** **A** can be the result of a logical operators.

**Example:**

```

is_raining = False

will_sunshine = not is_raining
print(will_sunshine)

```

Output:

True

## 3.6 Special Operators

### 3.6.1 Membership Operators

Test for membership in a sequence (string, list, tuple, etc.).

- `in` : True if value is found
- `not in` : True if value is not found

Example:

```
fruits = ['apple', 'banana', 'cherry']
check_1 = 'banana' in fruits
check_2 = 'pear' not in fruits
print(check_1)
print(check_2)
```

Output:

True  
True

### 3.6.2 Identity Operators

Compare object identities (not values).

- `is` : True if both refer to the same object
- `is not` : True if they refer to different objects

Example:

```
value_a = [1,2,3]
value_b = value_a
value_c = [1,2,3]
print(value_a is value_b)
print(value_a is not value_c)
```

Output:

True  
True

## Chapter 4

# Error and Exception Handling

### 4.1 Introduction

Programming errors (often called “bugs”) are mistakes in your code that prevent it from working as intended. No one writes perfect code on the first try – in fact, making errors is a normal part of learning to program. Each error you encounter is an opportunity to learn and improve your coding skills. It’s important to understand different types of errors so you can identify problems quickly and fix your programs. In Python (and most languages), errors generally fall into three main categories:

- Syntax Errors
- Runtime Errors
- Logical Errors

In the sections below, we’ll explain each type in a friendly way, with examples, the exact error messages or outputs you’d see, and tips on how to fix the errors.

### 4.2 Syntax Errors

**Syntax errors are like grammar mistakes in a sentence.** Every programming language has rules (syntax) about how code must be written. If you break these rules – for example, by misspelling a keyword or forgetting a symbol – Python doesn’t understand the code and will stop running it. In other words, a syntax error means the code cannot even be executed until the mistake is corrected. Python usually detects these errors before running the program and will point out where the problem is. Just as a bad sentence can’t be understood, bad syntax prevents Python from understanding your instructions.

#### 4.2.1 Common causes

1. **Missing punctuation:** Forgetting a required symbol, such as a colon : at the end of an if or for statement, or a quote or parenthesis in a pair.
2. **Indentation mistakes:** In Python, indentation (spaces or tabs at the beginning of a line) matters. If you indent code incorrectly or inconsistently, it can cause a syntax error.
3. **Typos or misspelled keywords:** For example, writing `print(“Hello”)` instead of `print(“Hello”)` will cause a syntax error because `print` is not a valid keyword.

4. **Unmatched pairs:** Not closing a string with a matching quote, or not closing parentheses/brackets. For instance, `print("Hello World!` (missing the ending quote) is a syntax error.

#### 4.2.2 Illustration

Example with Syntax Error:

```
length = 5                # line 1
width = 4                  # line 2
area_rectangle = length * width # line 3
print("The area is" area_rectangle) # line 4
```

In the if statement above, we forgot to put a comma `,` in **line 4** after closing the double quotes. Running this code will produce a syntax error. Python will highlight the location of the mistake and show an error message.

[27]:

```
length = 5                # line 1
width = 4                  # line 2
area_rectangle = length * width # line 3
print("The area is" area_rectangle) # line 4

Cell In[27], line 4
    print("The area is" area_rectangle) # line 4
    ^
SyntaxError: invalid syntax. Perhaps you forgot a comma?
```

Figure 4.1: Syntax Error

To fix the error, we simply add the missing `,` in **line 4** after closing the double quotes and the code will execute without a syntax error.

Example with Syntax Error:

```
length = 5                # line 1
width = 4                  # line 2
area_rectangle = length * width # line 3
print("The area is", area_rectangle) # line 4
```

Output:

```
The area is 20
```

### 4.3 Runtime Errors

Runtime errors (also known as exceptions) are problems that occur while the program is running. Unlike syntax errors, the code passes the initial syntax check and starts executing, but something unexpected happens during execution that causes the program to crash or stop. In simpler terms, a runtime error means **“Python understood your code, but couldn’t complete the execution due to an issue that arose as it was running.”** These errors will produce an error message explaining what went wrong. Runtime errors can be frustrating, but the messages Python gives are very helpful for debugging.

#### 4.3.1 Common causes

1. Using a variable that was never defined (this causes a `NameError` – Python doesn’t know what that name refers to).
2. Dividing by zero (this causes a `ZeroDivisionError`, because mathematically division by zero is undefined).
3. Trying to access a list index that doesn’t exist (this causes an `IndexError` – for example, asking for the 10th item in a list that has only 5 items).
4. Performing an operation on incompatible types (this causes a `TypeError`. For instance, trying to add a number and a string together will error out).

#### 4.3.2 Illustration

Example with Runtime Error (`ZeroDivisionError`):

```
num_1 = 1
num_2 = 23
num_1 -= 1
result = num_2 / num_1
```

In the above code, we try to divide 23 by 0. Dividing by zero is not allowed, so when Python executes `23/0`, it will throw an error at runtime. The program will stop and output an error message.

[28]:

```
1 num_1 = 1
2 num_2 = 23
3 num_1 -= 1
4 result = num_2 / num_1
```

```
-----
ZeroDivisionError                                Traceback (most recent call last)
Cell In[28], line 4
      2 num_2 = 23
      3 num_1 -= 1
----> 4 result = num_2 / num_1

ZeroDivisionError: division by zero
```

Figure 4.2: Runtime Error

The message `ZeroDivisionError: division by zero` is telling us exactly what went wrong: we attempted to divide by zero. For instance, we could add a check condition in the code to avoid dividing by zero, or use a `try/except` block to catch the error and handle it gracefully.

**Note:** We will dive deep into working around these errors when we handle selection and loops.

## 4.4 Logical Errors

Logical errors are the trickiest type of errors to deal with. Unlike syntax errors or most runtime errors, a logic error does not crash your program or produce any obvious error message. The code will run to completion, but the result will be wrong – not what you intended. In other words, the program's logic is flawed. The computer is doing exactly what you told it to do, but because of a mistake in your reasoning or algorithm, it's not doing what you wanted it to do. These errors can be subtle and hard to spot because Python won't point to a specific line – you have to notice the incorrect output yourself and deduce where the problem is.

### 4.4.1 Characteristics

1. No syntax error: The code runs without any syntax or runtime complaints. Python thinks everything is fine because from a language perspective, your code is valid.
2. Unexpected or incorrect output: The program's result is not what the programmer intended or expected. The output might look plausible, but it's wrong for the given problem.
3. Difficult to detect: Since there's no crash or loud error message, you might not realize something is wrong until you notice the results. You often have to test your program with different inputs or manually check results to catch logical errors.
4. Cause: faulty logic or assumptions: Logical errors come from mistakes in the algorithm or reasoning – for example, using the wrong formula, wrong conditions, or misunderstanding the problem. They are essentially bugs in your thinking that translate into bugs in the code.

### 4.4.2 Illustration

Imagine you want to find the area of a rectangle. We write the code as:

#### Example

```
length = 5           # line 1
width = 4             # line 2
area_rectangle = length + width  # line 3
print("The area is", area_rectangle)  # line 4
```

If we run this code, it will execute without any syntax or runtime errors, but the output will be incorrect.

#### Output:

The area is 9

The code ran completely, but the result 9 is wrong (the correct answer should be 20). This is a logical error. Why did it happen? The mistake was that in **line 3**, we use addition operator (+) instead of multiplication operator \*.

Generally, fixing a logic error involves revisiting your thought process: you might add print statements or use a debugger to trace what the program is doing, then adjust the code to do the right thing.

**Note:** Python didn't give any error message for the above bug – the only indication of a problem was the unexpected output. This is why testing your code and knowing the expected result is crucial for catching logical errors.

## 4.5 Exception Handling in Python

### 4.5.1 What is an Exception?

An **exception** is an error that occurs during the execution of a program, which disrupts the normal flow of the program's instructions. Unlike syntax errors that are detected before running the program, exceptions happen at runtime.

**Example:** Dividing a number by zero raises an exception, `ZeroDivisionError`.

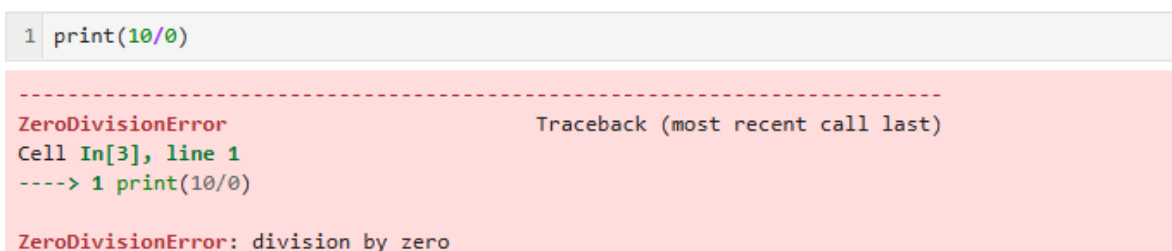
A screenshot of a Python environment showing a runtime error. At the top, a code cell contains the line `1 print(10/0)`. Below the code cell, a red-shaded area displays the error message. The message starts with a dashed line, followed by `ZeroDivisionError` and `Traceback (most recent call last)`. Below that, it shows `Cell In[3], line 1` and `----> 1 print(10/0)`. At the bottom of the red area, the full error message is `ZeroDivisionError: division by zero`.

Figure 4.3: Runtime Error

### 4.5.2 Why Handle Exceptions?

Exception handling allows programs to continue running or terminate gracefully when unexpected events occur, such as:

- **Prevent Program Crashes:** Without exception handling, an unexpected error will cause the program to terminate abruptly, leading to a poor user experience. By handling exceptions, the program can continue running or terminate gracefully.
- **Provide Informative Feedback:** When an error occurs, exception handling allows the program to display meaningful error messages to users instead of confusing tracebacks. This helps users understand what went wrong and how to fix it.
- **Manage Unexpected Situations:** Programs often interact with external systems (like files, databases, or networks) or take user input, which can be unpredictable. Exception handling prepares the program to handle such uncertain conditions gracefully.
- **Maintain Program Flow Control:** Exception handling separates error management code from regular logic, improving code readability and maintainability.

- **Enable Recovery or Alternative Actions:** In some cases, programs can recover from errors by trying alternative approaches, retrying operations, or cleaning up resources before exiting.

#### Example Scenario:

Consider a program that reads data from a file. If the file is missing and exceptions are not handled, the program will crash with an error message. Using exception handling, the program can catch the `FileNotFoundError` and inform the user to check the file location, or create a new file if appropriate.

### 4.5.3 Basic Exception Handling Syntax

Python provides a way to handle runtime errors (exceptions) using the `try-except` block. The basic structure is as follows:

#### Syntax:

```
try:
    # Code that might raise an exception
    <some risky operation>
except SomeException:
    # Code to handle the exception
    <handle error>
```

#### How it works:

- The code inside the `try` block is executed first.
- If no exception occurs, the `except` block is skipped.
- If an exception matching `SomeException` occurs inside the `try` block, Python immediately jumps to the corresponding `except` block.
- The `except` block executes, handling the error gracefully.
- After handling, the program continues with the rest of the code.

#### Example

```
try:
    x = int(input("Enter a number: "))
    result = 10 / x
    print("Result is:", result)
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
except ValueError:
    print("Error: Invalid input. Please enter an integer.")
```

#### Explanation:

- The user is prompted to enter a number.
- If the input is not an integer, a `ValueError` is raised and caught by the second `except` block.



- If the user enters zero, dividing by zero raises a `ZeroDivisionError` caught by the first `except` block.
- For any other valid integer input, the division occurs and the result is printed.

#### 4.5.4 Catching Multiple Exceptions

You can handle multiple exceptions separately with multiple `except` blocks or together using a tuple (collection of immutable items).

##### Example

```
try:
    x = int(input("Enter a number: "))
    print(10 / x)
except (ZeroDivisionError, ValueError):
    print("Invalid operation.")
```

This above example catches both `ZeroDivisionError` and `ValueError` in a single handler.

#### 4.5.5 The `else` Clause

The `else` block runs if no exceptions are raised in the `try` block.

##### Example

```
try:
    f = open("file.txt")
except FileNotFoundError:
    print("File not found.")
else:
    print("File opened successfully.")
    f.close()
```

This is useful for code that should only run if the `try` block succeeds without errors.

#### 4.5.6 The `finally` Clause

The `finally` block always runs after the `try` and `except` blocks, regardless of whether an exception was raised or not. It is commonly used for cleanup actions.

##### Example

```
try:
    f = open("file.txt")
except FileNotFoundError:
    print("File not found.")
finally:
    print("Execution complete.")
```

Even if an exception occurs, the `finally` block executes.

## Best Practices and Common Pitfalls

- **Avoid bare except:** Using `except:` without specifying an exception type can hide bugs and make debugging difficult.
- **Catch specific exceptions:** Handle only those exceptions you expect and can manage properly.
- **Keep try blocks small:** This localizes exceptions and makes it easier to identify and fix problems.
- **Use exceptions for exceptional cases:** Do not use exception handling for regular program flow control.

### Example of bad practice:

#### Example

```
try:
    x = int(input("Enter a number: "))
    print(10 / x)
except:
    print("Something went wrong.")  # Too broad, hides details
```

### Better approach:

#### Example

```
try:
    x = int(input("Enter a number: "))
    print(10 / x)
except ValueError:
    print("Invalid input. Please enter a valid integer.")
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

### Summary:

- Exceptions are runtime errors that disrupt normal program flow.
- Use `try-except` blocks to catch and handle exceptions.
- Use multiple `except` blocks or a tuple to handle several exceptions.
- Use `else` for code that runs only if no exceptions occur.
- Use `finally` for cleanup code that runs regardless of exceptions.
- Follow best practices to write clean, maintainable, and robust exception handling code.