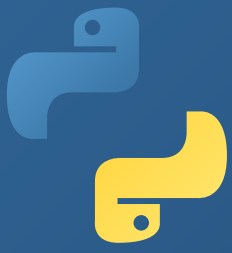


Python Programming

A Step-by-Step Guide

With Colorful Examples, Images & Scripting Tutorials

From Beginner to Professional

Alvin Robert

BAXCHAIN NETWORK LTD

📚 Table of Contents

Chapter 41: Summary & Next Steps

CHAPTER 1

Introduction to Python

Welcome to Python!

Let's pretend for a second that you just got home from vacation, and you have 200 pictures on your phone. They're all named something like "IMG_001.jpg," "IMG_002.jpg," and so on, because that's how your phone names them by default.

But if you use Python, you can rename all of them in a second with just a few lines of code!

I'm sure you are here because you want to learn to code using Python, and let me tell you this—you came to the right place. Python is not just a programming language; it is a golden ticket to a world filled with endless possibilities. Maybe you are here because you want to solve problems, build something cool, or make some money. For whatever reason, **Python can help you get there**.

> ✅ **Pro Tip**
>
> By the end of the first 5 chapters in this book, you will be able to write real Python programs—yes, real programs that can automate tasks, analyze data, or even help you understand what you need to build your first website.

What is Python?

Python is a **high-level, interpreted programming language** that is designed to be simple and easy to read. It was created by **Guido van Rossum** in 1991 and has since become one of the most widely used languages in the world.

> 💡 **Tech Term**
>
> **High-Level Language** — A programming language that is easy for humans to use and understand (close to English) and hides machine complexities from you.

> 💡 **Tech Term**
>
> **Interpreted Language** — Python executes your code one line at a time, so you can immediately see what it's doing. No need to compile first!

Think of Python as a universal tool—a Swiss Army knife of programming. It can handle a wide variety of tasks, from small scripts to large-scale applications.

Why Python Can Change Your Life

💼 Jobs and Money

Python is one of the most in-demand programming languages in the world. **Google, Netflix, and NASA** all use Python. Whether you want to become a web developer, data scientist, or machine learning engineer, Python is your golden ticket to a high-paying job.

🌐 Build Websites

Want to create your own website or start a business? Python can help you build professional, scalable websites using frameworks like **Django** and **Flask**.

💰 Cryptocurrency Development

Ever wondered how cryptocurrencies like Bitcoin and Ethereum work? Python is used extensively in blockchain development. You can even create your own cryptocurrency!

🤖 Data Science and AI

Python is the preferred language for data analysis, artificial intelligence, and machine learning. It's used to predict stock prices, analyze customer behavior, and even make self-driving cars!

🎮 Create Games

Design fun games to share with your friends or even publish them online using libraries like Pygame.

> **🎉 Fun Fact**
>
> **Did you know?** Python was named after the British comedy group "Monty Python," not the snake! Guido van Rossum, Python's creator, was a fan of Monty Python's Flying Circus and wanted the language to be fun and approachable.

Why Learn Python?

Reason	Description
Easy to Read	Python's syntax is designed to be as close to plain English as possible
Versatile	Web development, data science, AI, automation, games—Python does it all
Large Community	Millions of developers, countless tutorials, and helpful forums
Career Opportunities	High demand in the job market with excellent salaries

Here's a simple example that shows how readable Python is:

```python
# This simple line prints a message to the screen
print("Hello, World!")

# Output: Hello, World!
```

No complicated setup or syntax—just plain and simple!

📋 Chapter Summary

- Python is a beginner-friendly, versatile programming language
- Created by Guido van Rossum in 1991
- Used by major companies like Google, Netflix, and NASA
- Opens doors to web development, data science, AI, and more
- Has a simple syntax that's easy to read and write

Coming Up Next

In Chapter 2, we'll dive deeper into Python's history, features, and explore what makes it so special and popular among developers worldwide.

CHAPTER 2

Getting Started with Python

Let's Get Python Up and Running!

In this chapter, you will learn how to install Python on your computer, set up your development environment, and write your very first Python program. We'll guide you step by step for Windows, macOS, and Linux users.

Step 1: Downloading Python

Python is free and open-source. The latest version can always be downloaded from the official website:

1. Open your web browser and go to **https://www.python.org**
2. Click on the "Downloads" tab
3. Click the "Download Python [version]" button (e.g., "Download Python 3.12")

Step 2: Installing Python

For Windows Users

1. Double-click the downloaded installer
2. **IMPORTANT:** Check the box that says "Add Python to PATH"
3. Click "Install Now"

> ⚠️ **Warning**
>
> Make sure to check "Add Python to PATH" before clicking Install! This is the most common mistake beginners make, and it will cause problems later if you forget.

For macOS Users

1. Open the downloaded .pkg file
2. Follow the installation instructions
3. Python may already be installed, but it's best to install the latest version

For Linux Users

Most distributions come with Python pre-installed. Check by opening a terminal and typing:

```
python3 --version
```

If not installed, use your package manager:

```
# For Ubuntu/Debian
sudo apt install python3

# For Fedora
sudo dnf install python3
```

Step 3: Verifying Your Installation

After installation, open your terminal (Command Prompt on Windows, Terminal on macOS/Linux) and type:

```
python --version
# or
python3 --version
```

You should see something like: `Python 3.12.x`

> ✅ **Pro Tip**
>
> If you see "command not found," try restarting your terminal or computer. On some systems, you may need to use `python3` instead of `python`.

Step 4: Setting Up Your Editor

You can write Python code in any text editor, but using an **Integrated Development Environment (IDE)** makes coding much easier. Here are the popular choices:

Editor	Best For	Features
VS Code	Everyone	Free, powerful, extensions, debugging

IDLE	Beginners	Comes with Python, simple interface
PyCharm	Professionals	Full-featured IDE, code analysis
Jupyter Notebook	Data Science	Interactive coding, great for learning

> 🔍 **Hint**
>
> We recommend **VS Code** for most beginners. It's free, works on all platforms, and has excellent Python support with the Python extension.

Step 5: Writing Your First Python Program

Let's write and run your first Python script!

1. Open your text editor (VS Code, IDLE, or any editor)
2. Create a new file and type the following code:

```
# My first Python program!
print("Hello, Python World!")
print("I'm learning Python!")
print("This is amazing!")
```

3. Save the file as `hello.py`
4. Run it from your terminal:

```
python hello.py
```

Output:

```
Hello, Python World!
I'm learning Python!
This is amazing!
```

🎉 **Congratulations!** You've just written and run your first Python program!

Using Python Interactive Mode

You can also run Python commands directly in the terminal. Just type `python` or `python3` to enter interactive mode:

```
python3
>>> print("Hello!")
Hello!
>>> 2 + 2
4
>>> exit()
```

> 💡 **Tech Term**
>
> **Interactive Mode** — A way to run Python commands one at a time and see results immediately. Great for testing small pieces of code!

Troubleshooting Common Issues

Problem	Solution
"Command not found"	Make sure Python is added to PATH. Restart terminal after installation.
Multiple Python versions	Use `python3` instead of `python`
Permission issues	On Linux/macOS, use `sudo` for installation commands
File won't run	Make sure you saved the file with `.py` extension

📋 Chapter Summary

- Download Python from **python.org**
- Always check "Add Python to PATH" during installation on Windows
- Verify installation with `python --version`
- Use an IDE like VS Code for easier coding
- Python files use the `.py` extension
- Run programs with `python filename.py`

Coming Up Next

In Chapter 3, we'll learn about Python syntax—the rules that make Python code work. You'll understand indentation, case sensitivity, and how to write clean, readable code.

CHAPTER 3

Python Syntax

Understanding Python Syntax: The Foundation of Clean Code

Python syntax is the set of rules that tells the computer how to understand your code. Just like when you speak or write in any language, there are rules you follow to make sure your message is clear. In Python, syntax includes grammar, punctuation, and structure.

If you don't follow these syntax rules, the computer won't understand what you mean. It's like trying to read a sentence with missing words or jumbled letters—it becomes confusing and meaningless.

> 💡 **Tech Term**
>
> **Syntax** — The set of rules that defines how to write correctly structured programs in a programming language. Think of it as the grammar of Python.

Python Syntax vs English Grammar

Just like English has grammar rules:

- ✅ "I am learning Python" (Correct)
- ❌ "Python learning am I" (Incorrect)

Python has syntax rules:

```
print("Hello World")    # ✅ Correct syntax
Print("Hello World")    # ❌ Incorrect - wrong capitalization

if 5 > 2:               # ✅ Correct syntax
If 5 > 2:               # ❌ Incorrect - wrong capitalization
```

Key Python Syntax Rules

1. Indentation is Everything! 🎯

Most Important Rule: Python uses indentation (spaces or tabs) to define code blocks, unlike other languages that use curly braces `{}`.

✅ **Correct Indentation**

❌ **Incorrect Indentation**

```python
if 5 > 2:
    print("Five is greater!")
    print("This is indented")
print("This is outside")
```

```python
if 5 > 2:
print("IndentationError!")
# This will cause an error!
```

✅ **Pro Tip**

Best Practice: Use 4 spaces per indentation level (PEP 8 standard). Most editors can be configured to convert tabs to 4 spaces automatically.

2. Case Sensitivity 🔤

Python is **case-sensitive**, meaning uppercase and lowercase letters are treated as completely different:

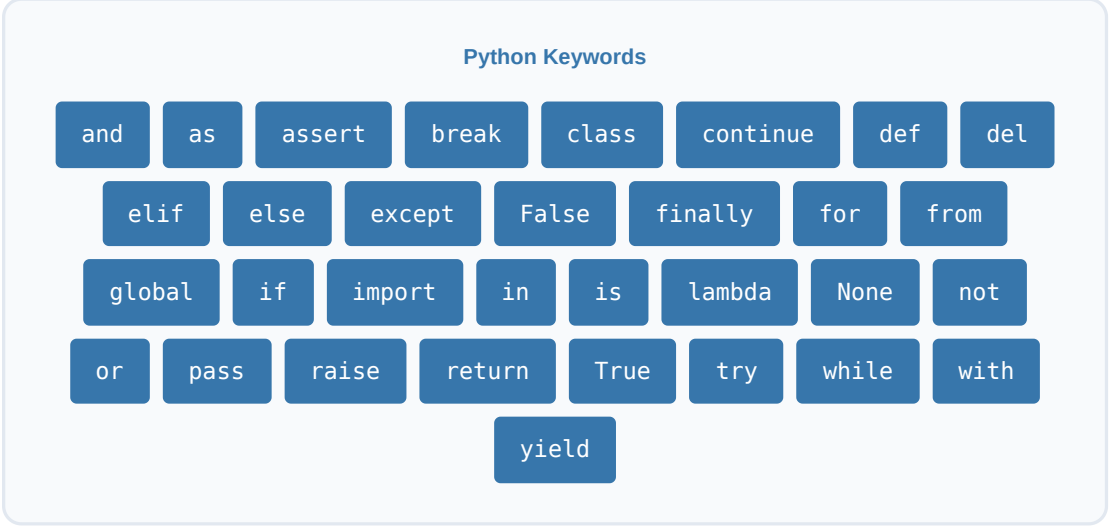
```python
# These are THREE different variables!
name = "John"
Name = "Jane"
NAME = "Bob"

print(name)  # Output: John
print(Name)  # Output: Jane
print(NAME)  # Output: Bob

# Functions are also case-sensitive
print("Hello")  # ✅ Correct
Print("Hello")  # ❌ NameError!
```

3. Python Keywords (Reserved Words)

Keywords are reserved words that have special meaning in Python. You **cannot** use them as variable names:



4. Identifier Naming Rules

Identifiers are names you give to variables, functions, classes, etc. Here are the rules:

Rule	✅ Valid	❌ Invalid
Must start with letter or underscore	`name`, `_private`	`2name`
Can contain letters, digits, underscores	`name1`, `my_var`	`my-var`
No spaces allowed	`customer_name`	`customer name`
Cannot be a keyword	`my_if`	`if`

Understanding Code Blocks

In Python, a **block** is a group of lines of code that belong together. Python uses indentation to define these blocks.

How Blocks Work

```
Main instruction 1
    ├── Sub-instruction A (indented - part of block)
    └── Sub-instruction B (indented - part of block)
Main instruction 2 (not indented - new block)
    └── Sub-instruction C
```

```python
if 10 > 5:
    print("Ten is greater than five")    # Inside if block
    print("This is part of the if block")  # Inside if block
print("This is outside the if block")      # Outside if block
```

Common Syntax Errors

Error	Cause	Fix
`IndentationError`	Incorrect or missing indentation	Use consistent 4 spaces
`SyntaxError`	Invalid syntax structure	Check parentheses, colons

`NameError`	Wrong capitalization or undefined variable	Check spelling and case

📝 Practice Exercise

Find and fix the errors in this code:

```
If 3 < 5:
print("3 is less than 5")
Print("Another message")
```

Answer: Fix capitalization of `if` and `print`, add proper indentation.

📋 Chapter Summary

- **Indentation** is crucial—it defines code blocks (use 4 spaces)
- Python is **case-sensitive**: `print()` ≠ `Print()`
- **Keywords** are reserved and cannot be used as variable names
- **Identifiers** must follow specific naming rules
- Proper syntax prevents errors and makes code readable

Coming Up Next

In Chapter 4, we'll learn about comments—how to add notes to your code that explain what it does without affecting how it runs.

CHAPTER 4

Comments in Python

What are Comments?

Comments are lines in Python code that are **not executed** by the interpreter. They are like little notes you write to yourself (or to other people) inside your code. Python ignores them completely when it runs your program.

> 💡 **Tech Term**
>
> **Comment** — A piece of text in your code that Python ignores. Used to explain code, make notes, or temporarily disable code.

Why Use Comments?

- 📝 **Explaining tricky parts:** Sometimes your code might do something complicated. A comment can explain why you did it that way.
- 📋 **Marking sections:** You can use comments to label different parts of your code, like "This section handles user login."
- ⏰ **Leaving reminders:** Need to fix something later? Leave a "TODO" comment as a reminder.
- 🔧 **Debugging:** Temporarily disable code without deleting it.

Single-Line Comments

In Python, single-line comments start with the **hash symbol (#)**. Everything after `#` on that line is ignored:

```
# This is a single-line comment
print("Hello, World!")  # This prints a message

# Calculate the total price
price = 100
tax = 0.15
total = price + (price * tax)  # Add 15% tax
```

Multi-Line Comments

Python doesn't have a special syntax for multi-line comments, but you have two options:

Option 1: Multiple # Symbols

- Alvin robert

```
# This is a multi-line comment
# written on more than one line
# Each line starts with #
print("Python is fun!")
```

Option 2: Triple Quotes (Docstrings)

```
"""
This is a multi-line string
that can be used as a comment.
Python will ignore it if not assigned
to a variable.
"""
print("Multi-line comment above!")
```

> 💡 **Tech Term**
>
> **Docstring** — A string literal that appears as the first statement in a function, class, or module. Used for documentation and can be accessed programmatically.

Best Practices for Comments

✅ **Good Comments**

```
# Calculate discount for members
# Members get 20% off orders over $50
if is_member and total > 50:
    discount = total * 0.20
```

❌ **Bad Comments**

```
# Set x to 5
x = 5  # This is obvious!

# Add 1 to counter
counter = counter + 1
```

> ✅ **Pro Tip**
>
> **Remember:** Comments should explain "WHY" you did something, not "WHAT" the code does. The code itself shows what it does!

TODO Comments

Developers often use special comment tags to mark tasks:

```
# TODO: Add error handling here
# FIXME: This calculation is wrong for negative numbers
# NOTE: This function requires Python 3.10+
# HACK: Temporary workaround, need better solution
```

📋 Chapter Summary

- Comments start with `#` and are ignored by Python
- Use comments to explain complex code and leave notes
- Multi-line comments use multiple `#` or triple quotes
- Explain "why" not "what"—don't over-comment obvious code
- Use TODO/FIXME tags for future work

Coming Up Next

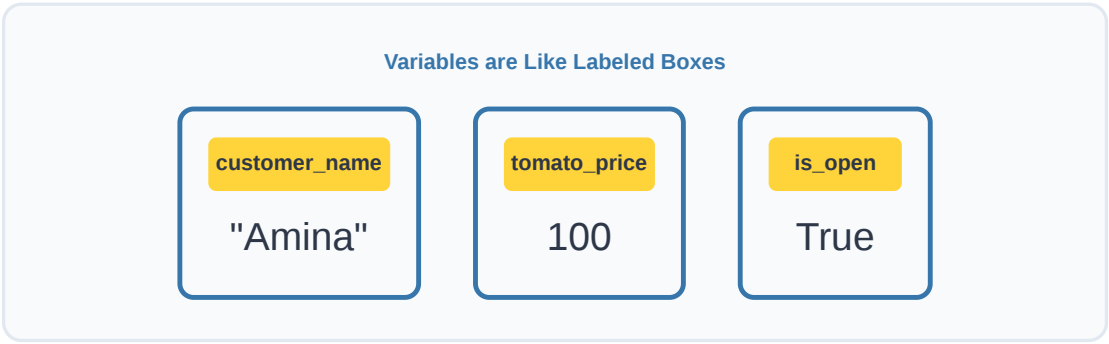
In Chapter 5, we'll explore variables—the containers that store data in your programs. You'll learn how to create, name, and use variables effectively.

CHAPTER 5

Variables

Understanding Variables the Simple Way

In Python, variables are like **boxes or containers** where you can keep things you want to use later. Imagine you're running a shop. You need to remember the price of tomatoes, the name of your customer, or how many items you have left. Variables help you keep track of all these things!



> 💡 **Tech Term**
>
> **Variable** — A name that refers to a value stored in the computer's memory. The name is like a label on a jar, and the value is what's inside the jar.

Creating Variables

To create a variable in Python, just write the name, an equals sign `=`, and the value:

```
# Creating variables
customer = "Amina"
tomato_price = 100
number_of_tomatoes = 50
shop_is_open = True

# Using variables
print(customer)        # Output: Amina
print(tomato_price)    # Output: 100
```

Variable Naming Rules

Rule	✅ Valid Examples	❌ Invalid Examples
Start with letter or underscore	`name`, `_private`, `MyVar`	`2name`, `@name`
Only letters, numbers, underscores	`customer1`, `total_price`	`customer-name`, `my var`
Case sensitive	`name`, `Name`, `NAME` (different!)	—
Cannot be Python keyword	`my_class`, `is_true`	`class`, `if`, `for`

Case Sensitivity Matters!

Python treats uppercase and lowercase letters as completely different:

```
customer = "Amina"
Customer = "Ali"
CUSTOMER = "Fatima"

# These are THREE different variables!
print(customer)   # Output: Amina
print(Customer)   # Output: Ali
print(CUSTOMER)   # Output: Fatima
```

> ⚠️ **Warning**
>
> Be very careful with capitalization! If you create `tomato_price` and later type `Tomato_Price`, Python will think it's a different variable and give you an error.

Changing Variable Values

You can change what's inside your box at any time:

```
number_of_tomatoes = 50    # Start with 50
print(number_of_tomatoes)   # Output: 50

number_of_tomatoes = 30    # After selling some
print(number_of_tomatoes)   # Output: 30

# You can even change the type!
item = "Tomato"     # String
item = 100          # Now it's an integer
item = True         # Now it's a boolean
```

> 💡 **Tech Term**
>
> **Dynamically Typed** — Python is called a dynamically typed language because you don't have to declare what type of value a variable will hold. Python figures it out automatically when you assign a value.

Multiple Assignment

Python allows you to assign values to multiple variables in one line:

```
# Assign same value to multiple variables
x = y = z = 0

# Assign different values in one line
name, age, city = "Amina", 25, "Lagos"

print(name)  # Output: Amina
print(age)   # Output: 25
print(city)  # Output: Lagos
```

Practical Example: Shop Inventory

```
# Shop information
shop_name = "Amina's Fresh Tomatoes"
tomato_price = 120.50
tomatoes_in_stock = 100
shop_is_open = True

# Display information
print("Welcome to", shop_name)
print("Price per kg:", tomato_price)
print("In stock:", tomatoes_in_stock)

# Customer buys 5 kg
sold = 5
tomatoes_in_stock = tomatoes_in_stock - sold
total_cost = sold * tomato_price

print("Sold:", sold, "kg")
print("Total cost:", total_cost)
print("Remaining:", tomatoes_in_stock)
```

> 📝 **Practice Exercise**
>
> Create variables for your own shop:

1. Shop name (string)
2. Number of items (integer)
3. Price per item (float)
4. Is shop open? (boolean)

Then print all the information!

📋 Chapter Summary

- Variables are containers for storing data
- Create with `name = value` syntax
- Names must start with letter or underscore
- Python is case-sensitive: `name` ≠ `Name`
- Python is dynamically typed—no need to declare types
- Values can be changed anytime

Coming Up Next

In Chapter 6, we'll explore Python's data types—the different kinds of values you can store in variables, including strings, numbers, and more!

CHAPTER 6

Data Types

Understanding Data Types

In Python, **data types** are like different kinds of containers for different things. Just like you use different containers in your kitchen—a glass for water, a bowl for rice, a jar for sugar—Python uses different data types to store different kinds of information.

> 💡 **Tech Term**
>
> **Data Type** — A classification that tells Python what kind of value a variable holds and what operations can be performed on it.

Python's Main Data Types

Python Data Types Overview

Category	Type	Example
Text	str	`"Hello"`
Numeric	int	`42`
	float	`3.14`
	complex	`3+4j`
Boolean	bool	`True`, `False`
Sequence	list	`[1, 2, 3]`
	tuple	`(1, 2, 3)`
	range	`range(6)`
	str	`"abc"`
Mapping	dict	`{"name": "Ali"}`

Set	set, frozenset	`{1, 2, 3}`
None	NoneType	`None`

Checking Data Types

Use the `type()` function to see what type a variable is:

```
name = "Amina"
age = 25
price = 99.99
is_open = True
fruits = ["apple", "banana"]

print(type(name))     # <class 'str'>
print(type(age))      # <class 'int'>
print(type(price))    # <class 'float'>
print(type(is_open))  # <class 'bool'>
print(type(fruits))   # <class 'list'>
```

1. Strings (str) — Text Data

Strings store text—words, sentences, or any characters. Wrap them in quotes:

```
# Single or double quotes work
name = "Amina Hassan"
greeting = 'Welcome to our shop!'
address = "123 Market Street"

# Multi-line strings with triple quotes
long_text = """This is a
multi-line
string"""

# Empty string
empty = ""
```

2. Integers (int) — Whole Numbers

Integers are whole numbers—positive, negative, or zero:

```
tomatoes = 50
customers = 25
temperature = -5
zero = 0

# Python handles very large integers!
big_number = 99999999999999999999
```

3. Floats (float) — Decimal Numbers

Floats are numbers with decimal points:

```
price = 99.99
weight = 2.5
temperature = -3.7
pi = 3.14159

# Scientific notation
distance = 1.5e6  # 1.5 million
```

4. Booleans (bool) — True or False

Booleans have only two values— `True` or `False`:

```
shop_is_open = True
is_raining = False
has_discount = True

# Results of comparisons are booleans
result = 10 > 5    # True
result = 3 == 7   # False
```

5. Lists — Ordered Collections

Lists store multiple items in order. Use square brackets:

```
fruits = ["apple", "banana", "orange"]
numbers = [1, 2, 3, 4, 5]
mixed = ["Amina", 25, True]  # Can mix types!

# Access by index (starts at 0)
print(fruits[0])  # Output: apple
```

6. Dictionaries — Key-Value Pairs

Dictionaries store data as key-value pairs. Use curly braces:

```
customer = {
    "name": "Amina",
    "age": 25,
    "city": "Lagos"
}

# Access by key
print(customer["name"])  # Output: Amina
```

7. Tuples — Immutable Lists

Tuples are like lists but cannot be changed after creation:

```
coordinates = (10, 20)
colors = ("red", "green", "blue")

# Access works like lists
print(colors[0])  # Output: red

# But you can't change them!
# colors[0] = "yellow"  # This would cause an error!
```

8. Sets — Unique Collections

Sets store unique items (no duplicates):

```
unique_numbers = {1, 2, 3, 3, 3}
print(unique_numbers)  # Output: {1, 2, 3}

# Duplicates are automatically removed!
```

9. None — The Empty Value

`None` represents the absence of a value:

```
result = None
print(result)        # Output: None
print(type(result))  # <class 'NoneType'>
```

📋 Chapter Summary

- **str** — Text data: `"Hello"`
- **int** — Whole numbers: `42`
- **float** — Decimal numbers: `3.14`
- **bool** — True or False
- **list** — Ordered, changeable: `[1, 2, 3]`
- **dict** — Key-value pairs: `{"key": "value"}`
- **tuple** — Ordered, unchangeable: `(1, 2, 3)`
- **set** — Unique values: `{1, 2, 3}`
- **None** — No value

Coming Up Next

In Chapter 7, we'll dive deeper into working with numbers—integers, floats, and mathematical operations!

CHAPTER 7

Numbers

Working with Numbers in Python

Python makes working with numbers easy and intuitive. Whether you're calculating prices, measuring distances, or analyzing data, Python has you covered.

Types of Numbers

1. Integers (int)

Whole numbers, positive or negative, without decimals:

```
x = 10
y = -5
z = 0
big = 999999999999

print(type(x))  # <class 'int'>
```

2. Floats (float)

Numbers with decimal points:

```
price = 19.99
pi = 3.14159
negative = -0.5

# Scientific notation
huge = 1.5e10    # 15,000,000,000
tiny = 2.5e-4    # 0.00025

print(type(price))  # <class 'float'>
```

3. Complex Numbers

Numbers with real and imaginary parts (advanced usage):

```
c = 3 + 4j
print(c.real)  # 3.0
print(c.imag)  # 4.0
```

Arithmetic Operations

Operator	Name	Example	Result
`+`	Addition	`10 + 3`	13
`-`	Subtraction	`10 - 3`	7
`*`	Multiplication	`10 * 3`	30
`/`	Division	`10 / 3`	3.333...
`//`	Floor Division	`10 // 3`	3
`%`	Modulus (Remainder)	`10 % 3`	1
`**`	Exponentiation	`2 ** 3`	8

```
# Examples
print(10 + 5)    # 15
print(10 - 5)    # 5
print(10 * 5)    # 50
print(10 / 3)    # 3.3333...
print(10 // 3)   # 3 (floor division)
print(10 % 3)    # 1 (remainder)
print(2 ** 10)   # 1024 (2 to the power of 10)
```

Useful Number Functions

```python
# Built-in functions
print(abs(-10))      # 10 (absolute value)
print(round(3.7))    # 4 (round to nearest integer)
print(round(3.14159, 2))  # 3.14 (round to 2 decimal places)
print(min(5, 3, 8))  # 3 (smallest value)
print(max(5, 3, 8))  # 8 (largest value)
print(pow(2, 3))     # 8 (2 raised to power 3)

# Math module for more functions
import math
print(math.sqrt(16))   # 4.0 (square root)
print(math.floor(3.9)) # 3 (round down)
print(math.ceil(3.1))  # 4 (round up)
print(math.pi)         # 3.14159...
```

Practical Example: Shopping Calculator

```python
# Shopping calculator
item_price = 150.00
quantity = 3
tax_rate = 0.15  # 15% tax
discount = 0.10  # 10% discount

# Calculate
subtotal = item_price * quantity
discount_amount = subtotal * discount
after_discount = subtotal - discount_amount
tax_amount = after_discount * tax_rate
total = after_discount + tax_amount

print(f"Subtotal: ${subtotal:.2f}")
print(f"Discount: -${discount_amount:.2f}")
print(f"Tax: +${tax_amount:.2f}")
print(f"Total: ${total:.2f}")
```

📋 Chapter Summary

- **int** — Whole numbers: `42`, `-7`, `0`
- **float** — Decimal numbers: `3.14`, `-0.5`
- Basic operators: `+` `-` `*` `/` `//` `%` `**`
- Use `round()`, `abs()`, `min()`, `max()` for common operations
- Import `math` for advanced functions

Coming Up Next

In Chapter 8, we'll learn about type casting—how to convert between different data types in Python!

CHAPTER 8

Casting (Type Conversion)

What is Casting?

Casting (or type conversion) is the process of converting a value from one data type to another. Sometimes you need a number as text, or text as a number—casting makes this possible.

> 💡 **Tech Term**
>
> **Type Casting** — Converting a value from one data type to another, such as converting a string to an integer or a float to a string.

Casting Functions

Function	Converts To	Example
`int()`	Integer	`int("42")` → `42`
`float()`	Float	`float("3.14")` → `3.14`
`str()`	String	`str(42)` → `"42"`
`bool()`	Boolean	`bool(1)` → `True`
`list()`	List	`list("abc")` → `['a', 'b', 'c']`

Converting to Integer

```python
# String to integer
age_str = "25"
age = int(age_str)
print(age + 5)  # 30

# Float to integer (truncates decimal)
price = 19.99
whole = int(price)
print(whole)  # 19 (not rounded, just cut off)

# Boolean to integer
print(int(True))   # 1
print(int(False))  # 0
```

> ⚠️ **Warning**
>
> You cannot convert a string with letters to an integer! `int("hello")` will cause a ValueError.

Converting to Float

```python
# String to float
price_str = "19.99"
price = float(price_str)
print(price * 2)  # 39.98

# Integer to float
count = 10
count_float = float(count)
print(count_float)  # 10.0
```

Converting to String

```python
# Number to string
age = 25
age_str = str(age)
print("I am " + age_str + " years old")

# Useful for concatenation
price = 99.99
message = "The price is $" + str(price)
print(message)  # The price is $99.99
```

Converting to Boolean

```
# Numbers to boolean
print(bool(1))    # True
print(bool(0))    # False
print(bool(-5))   # True (any non-zero is True)

# Strings to boolean
print(bool("Hello"))  # True
print(bool(""))       # False (empty string)

# Collections to boolean
print(bool([1, 2]))  # True
print(bool([]))      # False (empty list)
```

🔍 **Hint**

Rule of thumb: Empty values (`0` , `""` , `[]` , `None`) convert to `False` . Everything else converts to `True` .

Practical Example: User Input

```
# User input is always a string!
user_age = input("Enter your age: ")  # Returns string
print(type(user_age))  # <class 'str'>

# Convert to integer for calculations
age = int(user_age)
birth_year = 2024 - age
print(f"You were born around {birth_year}")
```

📋 Chapter Summary

- `int()` — Convert to integer (truncates decimals)
- `float()` — Convert to decimal number
- `str()` — Convert to string (text)
- `bool()` — Convert to boolean (True/False)
- User input with `input()` is always a string—convert as needed!

Coming Up Next

In Chapter 9, we'll explore strings in depth—one of the most commonly used data types in Python!

CHAPTER 9

Strings

Working with Text in Python

Strings are one of the most commonly used data types. They represent text—anything from a single character to entire documents.

Creating Strings

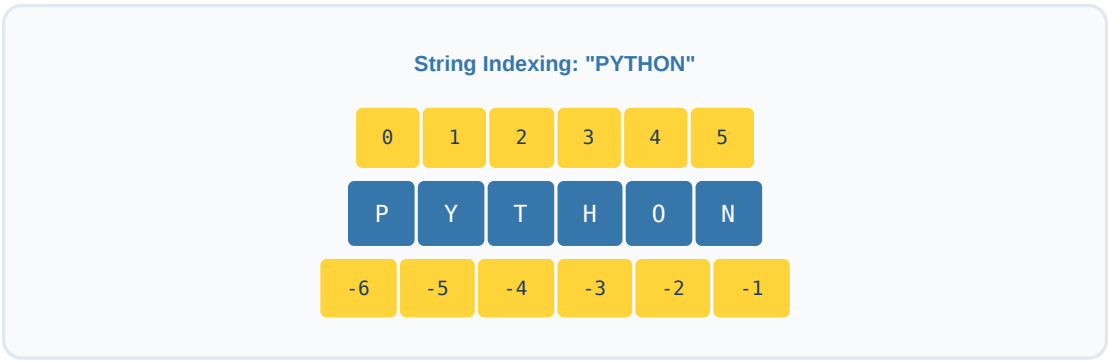
```
# Single or double quotes
name = 'Amina'
greeting = "Hello, World!"

# Triple quotes for multi-line
poem = """Roses are red,
Violets are blue,
Python is awesome,
And so are you!"""

# Empty string
empty = ""
```

String Indexing

Each character in a string has a position (index). Python uses **zero-based indexing**:



```
word = "PYTHON"
print(word[0])    # P (first character)
print(word[1])    # Y
print(word[-1])   # N (last character)
print(word[-2])   # O (second to last)
```

String Slicing

Extract portions of a string with slicing: `string[start:end:step]`

```
text = "Hello, World!"
print(text[0:5])     # Hello
print(text[7:])      # World!
print(text[:5])      # Hello
print(text[::2])     # Hlo ol! (every 2nd char)
print(text[::-1])    # !dlroW ,olleH (reversed)
```

String Methods

Method	Description	Example
`.upper()`	Convert to uppercase	`"hello".upper()` → `"HELLO"`
`.lower()`	Convert to lowercase	`"HELLO".lower()` → `"hello"`
`.title()`	Capitalize each word	`"hello world".title()` → `"Hello World"`
`.strip()`	Remove whitespace	`" hi ".strip()` → `"hi"`
`.replace()`	Replace text	`"hello".replace("l", "x")` → `"hexxo"`
`.split()`	Split into list	`"a,b,c".split(",")` → `['a','b','c']`
`.join()`	Join list into string	`"-".join(['a','b'])` → `"a-b"`
`.find()`	Find position	`"hello".find("l")` → `2`
`.count()`	Count occurrences	`"hello".count("l")` → `2`

```
name = "  amina hassan  "
print(name.strip())           # "amina hassan"
print(name.strip().title())   # "Amina Hassan"
print(name.strip().upper())   # "AMINA HASSAN"

# Check methods
text = "Hello123"
print(text.isalpha())   # False (contains numbers)
print(text.isalnum())   # True (letters and numbers)
print("123".isdigit())  # True
```

String Concatenation

```
# Using + operator
first = "Hello"
last = "World"
full = first + " " + last
print(full)  # Hello World

# String repetition
line = "=" * 20
print(line)  # ====================
```

String Formatting (f-strings)

The modern way to format strings in Python (Python 3.6+):

```
name = "Amina"
age = 25
price = 99.99

# f-string formatting
print(f"Hello, {name}!")
print(f"{name} is {age} years old")
print(f"Price: ${price:.2f}")  # 2 decimal places

# Expressions in f-strings
print(f"Next year: {age + 1}")
print(f"Name uppercase: {name.upper()}")
```

> ✅ **Pro Tip**
>
> **F-strings are the recommended way** to format strings in Python. They're readable, fast, and powerful!

Escape Characters

```
# Common escape characters
print("Line 1\nLine 2")    # New line
print("Tab\there")         # Tab
print("He said \"Hi\"")   # Quotes in string
print("Path: C:\\Users")   # Backslash
```

📋 Chapter Summary

- Strings are text enclosed in quotes
- Indexing starts at 0; negative indexes count from end
- Slicing: `string[start:end:step]`
- Many useful methods: `.upper()`, `.lower()`, `.strip()`, etc.
- F-strings for formatting: `f"Hello {name}"`

Coming Up Next

In Chapter 10, we'll explore booleans—the True/False values that control program logic!

CHAPTER 10

Booleans

True or False—The Foundation of Logic

Booleans are the simplest data type with only two possible values: `True` or `False`. They're the foundation of all decision-making in programming.

> 💡 **Tech Term**
>
> **Boolean** — A data type that can only be `True` or `False`. Named after mathematician George Boole, who developed Boolean algebra.

Creating Booleans

```
# Direct assignment
is_sunny = True
is_raining = False
shop_open = True

# From comparisons
result = 10 > 5    # True
result = 3 == 7    # False
result = 5 != 5    # False
```

Comparison Operators

Operator	Meaning	Example	Result
`==`	Equal to	`5 == 5`	True
`!=`	Not equal to	`5 != 3`	True
`>`	Greater than	`10 > 5`	True
`<`	Less than	`3 < 8`	True
`>=`	Greater or equal	`5 >= 5`	True

`<=`	Less or equal	`3 <= 2`	False

Logical Operators

```
# and - Both must be True
print(True and True)    # True
print(True and False)   # False

# or - At least one must be True
print(True or False)    # True
print(False or False)   # False

# not - Inverts the value
print(not True)         # False
print(not False)        # True
```

Practical Example

```
age = 20
has_id = True
is_member = True

# Can enter (18+ and has ID)
can_enter = age >= 18 and has_id
print(f"Can enter: {can_enter}")  # True

# Gets discount (member OR over 65)
gets_discount = is_member or age > 65
print(f"Gets discount: {gets_discount}")  # True
```

Truthy and Falsy Values

Python treats some values as `False` when converted to boolean:

```python
# Falsy values (evaluate to False)
print(bool(0))        # False
print(bool(""))       # False (empty string)
print(bool([]))       # False (empty list)
print(bool(None))     # False

# Truthy values (evaluate to True)
print(bool(1))        # True
print(bool("hello"))  # True
print(bool([1,2]))    # True
```

📋 Chapter Summary

- Booleans have only two values: `True` and `False`
- Comparison operators return booleans: `==` , `!=` , `>` , `<` , `>=` , `<=`
- Logical operators: `and` , `or` , `not`
- Empty/zero values are "falsy"; non-empty values are "truthy"

Coming Up Next

In Chapter 11, we'll explore all Python operators in depth!

CHAPTER 11

Operators

Python Operators Overview

Operators are special symbols that perform operations on values and variables. Python has several types of operators.

1. Arithmetic Operators

```
print(10 + 3)   # 13  Addition
print(10 - 3)   # 7   Subtraction
print(10 * 3)   # 30  Multiplication
print(10 / 3)   # 3.33 Division
print(10 // 3)  # 3   Floor division
print(10 % 3)   # 1   Modulus (remainder)
print(2 ** 3)   # 8   Exponentiation
```

2. Assignment Operators

```
x = 10        # Assign
x += 5       # x = x + 5  → 15
x -= 3       # x = x - 3  → 12
x *= 2       # x = x * 2  → 24
x /= 4       # x = x / 4  → 6.0
x //= 2      # x = x // 2 → 3.0
x %= 2       # x = x % 2  → 1.0
x **= 3      # x = x ** 3 → 1.0
```

3. Comparison Operators

```
print(5 == 5)   # True   Equal
print(5 != 3)   # True   Not equal
print(5 > 3)    # True   Greater than
print(5 < 3)    # False  Less than
print(5 >= 5)   # True   Greater or equal
print(5 <= 3)   # False  Less or equal
```

4. Logical Operators

```
print(True and False)  # False
print(True or False)   # True
print(not True)        # False
```

5. Identity Operators

```
a = [1, 2, 3]
b = [1, 2, 3]
c = a

print(a == b)    # True  (same values)
print(a is b)    # False (different objects)
print(a is c)    # True  (same object)
print(a is not b) # True
```

6. Membership Operators

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

print("apple" in fruits)      # True
print("grape" in fruits)      # False
print("grape" not in fruits)  # True

# Works with strings too
print("py" in "python")       # True
```

Operator Precedence

From highest to lowest priority:

Priority	Operator	Description
1	`()`	Parentheses
2	`**`	Exponentiation
3	`+x, -x`	Unary plus/minus
4	`*, /, //, %`	Multiplication, division

5	`+, -`	Addition, subtraction
6	`==, !=, <, >, <=, >=`	Comparisons
7	`not`	Logical NOT
8	`and`	Logical AND
9	`or`	Logical OR

```
# Precedence examples
print(2 + 3 * 4)      # 14 (not 20)
print((2 + 3) * 4)    # 20 (parentheses first)
print(2 ** 3 ** 2)    # 512 (right to left)
```

📋 Chapter Summary

- **Arithmetic:** `+`, `-`, `*`, `/`, `//`, `%`, `**`
- **Assignment:** `=`, `+=`, `-=`, etc.
- **Comparison:** `==`, `!=`, `<`, `>`, `<=`, `>=`
- **Logical:** `and`, `or`, `not`
- **Identity:** `is`, `is not`
- **Membership:** `in`, `not in`

Coming Up Next

In Chapter 12, we'll explore lists—one of Python's most versatile data structures!

CHAPTER 12

Lists

Python Lists—Ordered Collections

Lists are one of Python's most versatile data structures. They can store multiple items of any type, and you can add, remove, or change items at any time.

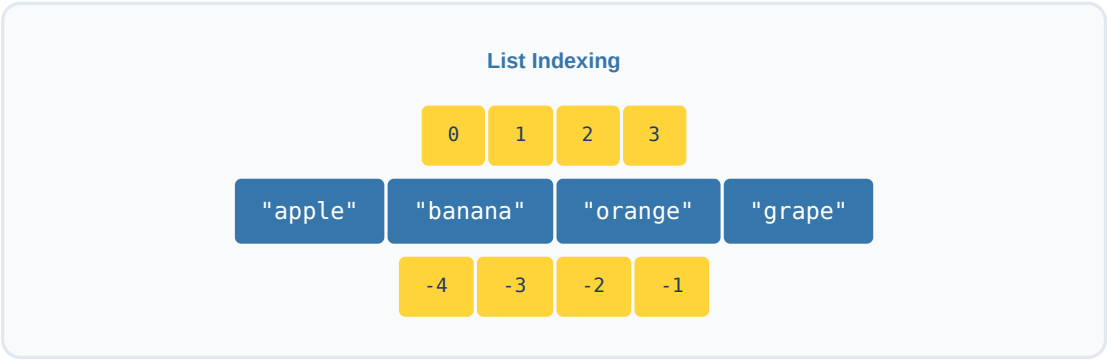
> 💡 **Tech Term**
>
> **List** — An ordered, mutable (changeable) collection of items. Items are accessed by their index position.

Creating Lists

```
# Different ways to create lists
fruits = ["apple", "banana", "orange"]
numbers = [1, 2, 3, 4, 5]
mixed = ["Amina", 25, True, 3.14]
empty = []

# Using list() function
letters = list("hello")  # ['h', 'e', 'l', 'l', 'o']
```

Accessing List Items



```
fruits = ["apple", "banana", "orange", "grape"]

print(fruits[0])     # apple
print(fruits[-1])    # grape (last)
print(fruits[1:3])   # ['banana', 'orange'] (slicing)
print(fruits[:2])    # ['apple', 'banana']
print(fruits[2:])    # ['orange', 'grape']
```

Modifying Lists

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

# Change an item
fruits[1] = "mango"
print(fruits)  # ['apple', 'mango', 'orange']

# Add items
fruits.append("grape")     # Add to end
fruits.insert(1, "kiwi")  # Insert at position

# Remove items
fruits.remove("apple")     # Remove by value
del fruits[0]              # Remove by index
last = fruits.pop()        # Remove and return last
fruits.clear()             # Remove all items
```

Common List Methods

Method	Description
`.append(x)`	Add item to end
`.insert(i, x)`	Insert item at index i
`.remove(x)`	Remove first occurrence of x
`.pop(i)`	Remove and return item at index i
`.index(x)`	Find index of first occurrence
`.count(x)`	Count occurrences of x
`.sort()`	Sort list in place

`.reverse()`	Reverse list in place
`.copy()`	Return a copy of the list

```
numbers = [3, 1, 4, 1, 5, 9, 2]

print(len(numbers))     # 7 (length)
print(min(numbers))     # 1
print(max(numbers))     # 9
print(sum(numbers))     # 25

numbers.sort()
print(numbers)  # [1, 1, 2, 3, 4, 5, 9]

numbers.reverse()
print(numbers)  # [9, 5, 4, 3, 2, 1, 1]
```

List Comprehensions

A powerful, concise way to create lists:

```
# Traditional way
squares = []
for x in range(5):
    squares.append(x ** 2)

# List comprehension (much cleaner!)
squares = [x ** 2 for x in range(5)]
print(squares)  # [0, 1, 4, 9, 16]

# With condition
evens = [x for x in range(10) if x % 2 == 0]
print(evens)  # [0, 2, 4, 6, 8]
```

📋 Chapter Summary

- Lists store multiple items: `[1, 2, 3]`
- Access by index: `list[0]`, `list[-1]`
- Mutable—can add, remove, change items
- Useful methods: `append()`, `remove()`, `sort()`
- List comprehensions: `[x for x in range(5)]`

Coming Up Next

In Chapter 13, we'll explore tuples—immutable sequences that are perfect for fixed data!

CHAPTER 13

Tuples

Immutable Sequences

Tuples are like lists, but they **cannot be changed** after creation. This makes them perfect for data that shouldn't be modified.

> 💡 **Tech Term**
>
> **Tuple** — An ordered, immutable (unchangeable) collection of items. Created using parentheses `()`.

Creating Tuples

```
# Different ways to create tuples
coordinates = (10, 20)
colors = ("red", "green", "blue")
mixed = ("Amina", 25, True)

# Single item tuple (note the comma!)
single = (5,)  # Without comma, it's just a number

# Without parentheses (tuple packing)
point = 3, 4, 5
print(type(point))  # <class 'tuple'>
```

Accessing Tuple Items

```
colors = ("red", "green", "blue")

print(colors[0])     # red
print(colors[-1])    # blue
print(colors[0:2])   # ('red', 'green')

# Unpacking
r, g, b = colors
print(r)  # red
print(g)  # green
```

Tuples Are Immutable

```
colors = ("red", "green", "blue")

# This will cause an error!
# colors[0] = "yellow"  # TypeError!

# But you can create a new tuple
colors = ("yellow",) + colors[1:]
print(colors)  # ('yellow', 'green', 'blue')
```

Why Use Tuples?

- **Data Protection:** Can't accidentally change values
- **Performance:** Slightly faster than lists
- **Dictionary Keys:** Can be used as dictionary keys (lists can't)
- **Multiple Returns:** Functions often return multiple values as tuples

```
# Function returning multiple values
def get_stats(numbers):
    return min(numbers), max(numbers), sum(numbers)

minimum, maximum, total = get_stats([1, 2, 3, 4, 5])
print(f"Min: {minimum}, Max: {maximum}, Sum: {total}")
```

📋 Chapter Summary

- Tuples use parentheses: `(1, 2, 3)`
- Immutable—cannot be changed after creation
- Access items same as lists: `tuple[0]`
- Great for fixed data and multiple return values
- Single-item tuple needs comma: `(5,)`

Coming Up Next

In Chapter 14, we'll explore sets—collections of unique items!

CHAPTER 14

Sets

Collections of Unique Items

Sets are unordered collections that automatically remove duplicates. They're perfect for membership testing and mathematical operations.

> 💡 **Tech Term**
>
> **Set** — An unordered collection of unique items. Duplicates are automatically removed.

Creating Sets

```
# Creating sets
fruits = {"apple", "banana", "orange"}
numbers = {1, 2, 3, 3, 3}  # Duplicates removed
print(numbers)  # {1, 2, 3}

# From a list (removes duplicates)
my_list = [1, 2, 2, 3, 3, 3]
unique = set(my_list)
print(unique)  # {1, 2, 3}

# Empty set (not {}!)
empty = set()  # {} creates empty dict
```

Set Operations

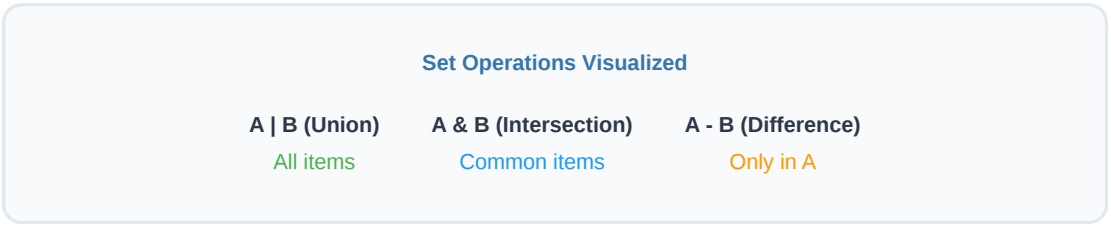
```python
set_a = {1, 2, 3, 4}
set_b = {3, 4, 5, 6}

# Union (all items from both)
print(set_a | set_b)  # {1, 2, 3, 4, 5, 6}

# Intersection (items in both)
print(set_a & set_b)  # {3, 4}

# Difference (in A but not B)
print(set_a - set_b)  # {1, 2}

# Symmetric difference (in A or B, not both)
print(set_a ^ set_b)  # {1, 2, 5, 6}
```



Modifying Sets

```python
fruits = {"apple", "banana"}

# Add items
fruits.add("orange")
fruits.update(["grape", "kiwi"])

# Remove items
fruits.remove("apple")    # Error if not found
fruits.discard("mango")   # No error if not found
fruits.pop()              # Remove random item
fruits.clear()            # Remove all
```

Membership Testing (Fast!)

```python
large_set = set(range(1000000))

# Very fast lookup!
print(999999 in large_set)  # True (instant)
```

📋 Chapter Summary

- Sets use curly braces: `{1, 2, 3}`
- Automatically remove duplicates
- Unordered—no indexing
- Operations: `|` union, `&` intersection, `-` difference
- Very fast membership testing with `in`

Coming Up Next

In Chapter 15, we'll explore dictionaries—powerful key-value data structures!

CHAPTER 15

Dictionaries

Key-Value Pairs

Dictionaries store data as **key-value pairs**. Think of them like a real dictionary where you look up a word (key) to find its definition (value).

> 💡 **Tech Term**
>
> **Dictionary** — A collection of key-value pairs where each key maps to a value. Keys must be unique and immutable.

Creating Dictionaries

```
# Creating dictionaries
customer = {
    "name": "Amina",
    "age": 25,
    "city": "Lagos",
    "is_member": True
}

# Using dict() function
person = dict(name="Ali", age=30)

# Empty dictionary
empty = {}
```

Accessing Values

```
customer = {"name": "Amina", "age": 25, "city": "Lagos"}

# Using brackets
print(customer["name"])  # Amina

# Using get() - safer (returns None if key doesn't exist)
print(customer.get("name"))      # Amina
print(customer.get("email"))     # None
print(customer.get("email", "N/A"))  # N/A (default)
```

Modifying Dictionaries

```
customer = {"name": "Amina", "age": 25}

# Add or update
customer["email"] = "amina@email.com"
customer["age"] = 26

# Update multiple
customer.update({"city": "Lagos", "phone": "123-456"})

# Remove
del customer["phone"]
age = customer.pop("age")  # Remove and return
```

Dictionary Methods

```
customer = {"name": "Amina", "age": 25, "city": "Lagos"}

# Get keys, values, items
print(customer.keys())    # dict_keys(['name', 'age', 'city'])
print(customer.values())  # dict_values(['Amina', 25, 'Lagos'])
print(customer.items())   # dict_items([('name', 'Amina'), ...])

# Check if key exists
print("name" in customer)   # True
print("email" in customer)  # False
```

Iterating Through Dictionaries

```
customer = {"name": "Amina", "age": 25, "city": "Lagos"}

# Loop through keys
for key in customer:
    print(key, "->", customer[key])

# Loop through key-value pairs
for key, value in customer.items():
    print(f"{key}: {value}")
```

Nested Dictionaries

```
shop = {
    "products": {
        "tomatoes": {"price": 100, "stock": 50},
        "onions": {"price": 80, "stock": 30}
    },
    "owner": "Amina"
}

# Access nested values
print(shop["products"]["tomatoes"]["price"])  # 100
```

📋 Chapter Summary

- Dictionaries store key-value pairs: `{"key": "value"}`
- Access values: `dict["key"]` or `dict.get("key")`
- Keys must be unique and immutable
- Methods: `.keys()`, `.values()`, `.items()`
- Check membership: `"key" in dict`

Coming Up Next

In Chapter 16, we'll explore arrays and when to use them instead of lists!

CHAPTER 16

Arrays

Arrays vs Lists

Python doesn't have built-in arrays like some languages. For most purposes, use **lists**. However, for numerical computing, the `array` module or **NumPy** provides efficient arrays.

Using the array Module

```
from array import array

# Create an array (all items must be same type)
numbers = array('i', [1, 2, 3, 4, 5])  # 'i' for integers
print(numbers[0])  # 1

# Type codes: 'i' (int), 'f' (float), 'd' (double)
prices = array('f', [19.99, 29.99, 39.99])
```

NumPy Arrays (Recommended)

```
import numpy as np

# Create array
arr = np.array([1, 2, 3, 4, 5])
print(arr * 2)  # [2 4 6 8 10] - element-wise!

# 2D array
matrix = np.array([[1, 2], [3, 4]])
print(matrix.shape)  # (2, 2)
```

> ✅ **Pro Tip**
>
> For general purposes, **use lists**. For numerical/scientific computing with large datasets, use **NumPy arrays**.

📋 Chapter Summary

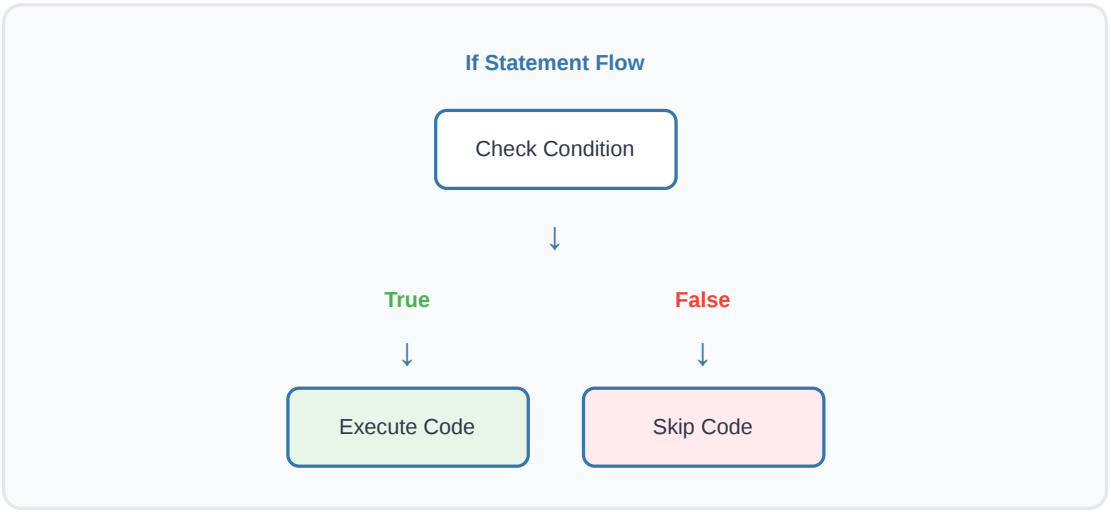
- Python lists serve most array needs
- `array` module: same-type arrays
- NumPy: powerful numerical arrays
- NumPy supports vectorized operations

CHAPTER 17

If...Else Statements

Making Decisions in Python

Conditional statements let your program make decisions based on conditions. They're the foundation of program logic.



Basic If Statement

```
age = 18

if age >= 18:
    print("You can vote!")
```

If...Else

```
age = 15

if age >= 18:
    print("You can vote!")
else:
    print("Too young to vote")
```

If...Elif...Else

```
score = 85

if score >= 90:
    grade = "A"
elif score >= 80:
    grade = "B"
elif score >= 70:
    grade = "C"
elif score >= 60:
    grade = "D"
else:
    grade = "F"

print(f"Your grade: {grade}")
```

Nested If Statements

```
age = 25
has_license = True

if age >= 18:
    if has_license:
        print("You can drive!")
    else:
        print("Get a license first")
else:
    print("Too young to drive")
```

Conditional Expressions (Ternary)

```
# One-liner if/else
age = 20
status = "adult" if age >= 18 else "minor"
print(status)  # adult
```

Multiple Conditions

```python
age = 25
income = 50000

# Using and
if age >= 18 and income >= 30000:
    print("Loan approved")

# Using or
if age < 18 or income < 20000:
    print("Loan denied")
```

📋 Chapter Summary

- `if` : Execute if condition is True
- `elif` : Check additional conditions
- `else` : Execute if all conditions are False
- Combine with `and` , `or` , `not`
- Ternary: `x if condition else y`

CHAPTER 18

Match Statements

Pattern Matching (Python 3.10+)

The `match` statement provides a cleaner way to handle multiple conditions, similar to switch/case in other languages.

```
status = 404

match status:
    case 200:
        print("Success!")
    case 404:
        print("Not found")
    case 500:
        print("Server error")
    case _:
        print("Unknown status")  # Default case
```

Matching Multiple Values

```
day = "Saturday"

match day:
    case "Saturday" | "Sunday":
        print("Weekend! 🎉")
    case _:
        print("Weekday")
```

Matching with Guards

```python
age = 25

match age:
    case n if n < 0:
        print("Invalid age")
    case n if n < 18:
        print("Minor")
    case n if n < 65:
        print("Adult")
    case _:
        print("Senior")
```

📋 Chapter Summary

- `match` provides pattern matching (Python 3.10+)
- Use `_` as the default case
- Use `|` to match multiple values
- Guards add extra conditions with `if`

CHAPTER 19

While Loops

Repeating While a Condition is True

A `while` loop continues executing as long as its condition remains True.

```
count = 0

while count < 5:
    print(f"Count: {count}")
    count += 1

# Output: Count: 0, 1, 2, 3, 4
```

> ⚠️ **Warning**
>
> Always ensure your while loop has a way to end! Otherwise, you'll create an infinite loop that never stops.

Break Statement

```
while True:
    user_input = input("Enter 'quit' to exit: ")
    if user_input == "quit":
        break  # Exit the loop
    print(f"You entered: {user_input}")
```

Continue Statement

```
count = 0

while count < 10:
    count += 1
    if count % 2 == 0:
        continue  # Skip even numbers
    print(count)  # Only prints odd: 1, 3, 5, 7, 9
```

While with Else

```
count = 0

while count < 3:
    print(count)
    count += 1
else:
    print("Loop completed!")  # Runs if no break
```

📋 Chapter Summary

- `while` loops repeat while condition is True
- `break` exits the loop immediately
- `continue` skips to next iteration
- Always ensure loop can terminate

CHAPTER 20

For Loops

Iterating Over Sequences

A `for` loop iterates over items in a sequence (list, string, range, etc.).

```
# Loop through a list
fruits = ["apple", "banana", "orange"]

for fruit in fruits:
    print(fruit)
```

Using range()

```
# range(stop)
for i in range(5):
    print(i)  # 0, 1, 2, 3, 4

# range(start, stop)
for i in range(1, 6):
    print(i)  # 1, 2, 3, 4, 5

# range(start, stop, step)
for i in range(0, 10, 2):
    print(i)  # 0, 2, 4, 6, 8
```

Loop Through Strings

```
for char in "Python":
    print(char)  # P, y, t, h, o, n
```

Enumerate for Index + Value

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

for index, fruit in enumerate(fruits):
    print(f"{index}: {fruit}")
# 0: apple
# 1: banana
# 2: orange
```

Loop Through Dictionaries

```
person = {"name": "Amina", "age": 25}

for key, value in person.items():
    print(f"{key}: {value}")
```

Nested Loops

```
for i in range(3):
    for j in range(3):
        print(f"({i}, {j})", end=" ")
    print()  # New line
```

📋 Chapter Summary

- `for` iterates over sequences
- `range()` generates number sequences
- `enumerate()` gives index + value
- Works with lists, strings, dicts, etc.
- Supports `break` and `continue`

CHAPTER 21

Functions

Reusable Blocks of Code

Functions are reusable blocks of code that perform specific tasks. They help organize code and avoid repetition.

> 💡 **Tech Term**
>
> **Function** — A named block of code that performs a specific task and can be called (executed) whenever needed.

Creating Functions

```
# Define a function
def greet():
    print("Hello, World!")

# Call the function
greet()  # Output: Hello, World!
```

Parameters and Arguments

```
# Function with parameters
def greet(name):
    print(f"Hello, {name}!")

greet("Amina")  # Output: Hello, Amina!

# Multiple parameters
def add(a, b):
    return a + b

result = add(5, 3)  # 8
```

Default Parameters

```
def greet(name="Guest"):
    print(f"Hello, {name}!")

greet()          # Hello, Guest!
greet("Amina")   # Hello, Amina!
```

Return Values

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

area = calculate_area(5, 3)
print(f"Area: {area}")  # Area: 15

# Return multiple values
def get_stats(numbers):
    return min(numbers), max(numbers), sum(numbers)

minimum, maximum, total = get_stats([1, 2, 3, 4, 5])
```

*args and **kwargs

```
# *args - variable number of arguments
def add_all(*numbers):
    return sum(numbers)

print(add_all(1, 2, 3))        # 6
print(add_all(1, 2, 3, 4, 5))  # 15

# **kwargs - keyword arguments
def print_info(**info):
    for key, value in info.items():
        print(f"{key}: {value}")

print_info(name="Amina", age=25, city="Lagos")
```

Docstrings

```python
def calculate_area(length, width):
    """
    Calculate the area of a rectangle.

    Parameters:
        length: The length of the rectangle
        width: The width of the rectangle

    Returns:
        The area (length * width)
    """
    return length * width

# Access docstring
print(calculate_area.__doc__)
```

📋 Chapter Summary

- Define with `def function_name():`
- Parameters receive input values
- `return` sends back results
- Default parameters: `def f(x=5):`
- `*args` for variable arguments
- `**kwargs` for keyword arguments

CHAPTER 22

Lambda Functions

Anonymous Functions

Lambda functions are small, anonymous functions defined in a single line.

```
# Regular function
def add(a, b):
    return a + b

# Lambda equivalent
add = lambda a, b: a + b
print(add(5, 3))  # 8
```

Lambda with map()

```
numbers = [1, 2, 3, 4, 5]

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

Lambda with filter()

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

# Filter even numbers
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(evens)  # [2, 4, 6]
```

Lambda with sorted()

```
students = [
    {"name": "Ali", "age": 25},
    {"name": "Amina", "age": 22}
]

# Sort by age
sorted_students = sorted(students, key=lambda x: x["age"])
print(sorted_students)
```

📋 Chapter Summary

- Syntax: `lambda args: expression`
- Small, one-line functions
- Great with `map()`, `filter()`, `sorted()`
- Use for simple operations only

CHAPTER 23

Scope

Variable Visibility

Scope determines where variables can be accessed in your code.

Local vs Global

```
x = 10  # Global variable

def my_function():
    y = 5  # Local variable
    print(x)  # Can access global
    print(y)  # Can access local

my_function()
print(x)    # Works
# print(y)  # Error! y is local to function
```

Using global Keyword

```
count = 0

def increment():
    global count
    count += 1

increment()
print(count)  # 1
```

📋 Chapter Summary

- **Local:** Inside function only
- **Global:** Accessible everywhere

- `global` modifies global from inside function

CHAPTER 24

Modules

Organizing Code with Modules

Modules are Python files containing functions, classes, and variables that you can import and use.

Importing Modules

```
# Import entire module
import math
print(math.sqrt(16))  # 4.0
print(math.pi)        # 3.14159...

# Import specific items
from math import sqrt, pi
print(sqrt(16))  # 4.0

# Import with alias
import math as m
print(m.sqrt(16))  # 4.0
```

Creating Your Own Module

```
# mymodule.py
def greet(name):
    return f"Hello, {name}!"

PI = 3.14159

# main.py
import mymodule
print(mymodule.greet("Amina"))
print(mymodule.PI)
```

Common Built-in Modules

Module	Purpose

`math`	Mathematical functions
`random`	Random numbers
`datetime`	Date and time
`os`	Operating system interface
`json`	JSON handling
`re`	Regular expressions

📋 Chapter Summary

- `import module` — import entire module
- `from module import x` — import specific items
- `import module as alias` — use alias
- Create modules by making .py files

CHAPTER 25

Introduction to OOP

Object-Oriented Programming

OOP is a programming paradigm that organizes code into objects—bundles of data (attributes) and functionality (methods).

> 💡 **Tech Term**
>
> **Object-Oriented Programming (OOP)** — A programming approach based on the concept of "objects" which contain data and code that operates on that data.

Key OOP Concepts

- **Class:** A blueprint for creating objects
- **Object:** An instance of a class
- **Attributes:** Data stored in an object
- **Methods:** Functions that belong to an object
- **Inheritance:** Creating new classes from existing ones
- **Encapsulation:** Hiding internal details
- **Polymorphism:** Same interface, different implementations

Real-World Analogy

Think of a **class** as a cookie cutter and **objects** as the cookies made from it. The cookie cutter (class) defines the shape, but each cookie (object) can have different decorations (attribute values).

📋 Chapter Summary

- OOP organizes code into objects
- Classes are blueprints
- Objects are instances of classes

- Promotes reusability and organization

CHAPTER 26

Classes and Objects

Creating Classes

```python
class Person:
    # Constructor (initializer)
    def __init__(self, name, age):
        self.name = name  # Attribute
        self.age = age    # Attribute

    # Method
    def greet(self):
        return f"Hello, I'm {self.name}!"

    # Another method
    def birthday(self):
        self.age += 1
        return f"Happy birthday! Now {self.age}"

# Create objects
person1 = Person("Amina", 25)
person2 = Person("Ali", 30)

print(person1.name)      # Amina
print(person1.greet())   # Hello, I'm Amina!
print(person1.birthday()) # Happy birthday! Now 26
```

The self Parameter

`self` refers to the current instance of the class. It's how methods access the object's attributes and other methods.

Class vs Instance Attributes

```python
class Dog:
    species = "Canine"  # Class attribute (shared)

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

dog1 = Dog("Buddy")
dog2 = Dog("Max")

print(dog1.species)  # Canine (shared)
print(dog2.species)  # Canine (shared)
print(dog1.name)     # Buddy (unique)
print(dog2.name)     # Max (unique)
```

📋 Chapter Summary

- Classes define with `class ClassName:`
- `__init__` is the constructor
- `self` refers to current instance
- Attributes store data
- Methods are functions in a class

CHAPTER 27

Inheritance

Creating Child Classes

Inheritance allows a class to inherit attributes and methods from another class.

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

    def speak(self):
        return "Some sound"

# Child class
class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

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

print(dog.name)     # Buddy (inherited)
print(dog.speak())  # Woof! (overridden)
print(cat.speak())  # Meow! (overridden)
```

Using super()

```
class Student(Person):
    def __init__(self, name, age, grade):
        super().__init__(name, age)  # Call parent __init__
        self.grade = grade

    def study(self):
        return f"{self.name} is studying"
```

📋 Chapter Summary

- Syntax: `class Child(Parent):`
- Child inherits parent's attributes/methods
- Override methods by redefining them
- `super()` accesses parent class

CHAPTER 28

Polymorphism

Same Interface, Different Behavior

Polymorphism allows different classes to be treated the same way through a common interface.

```
class Dog:
    def speak(self):
        return "Woof!"

class Cat:
    def speak(self):
        return "Meow!"

class Bird:
    def speak(self):
        return "Tweet!"

# Polymorphism in action
animals = [Dog(), Cat(), Bird()]

for animal in animals:
    print(animal.speak())  # Each calls its own speak()
```

📋 Chapter Summary

- Same method name, different implementations
- Enables flexible, extensible code
- Common in inheritance hierarchies

CHAPTER 29

Iterators

Iterating Through Data

An iterator is an object that allows you to traverse through all elements of a collection.

```
# Using iter() and next()
fruits = ["apple", "banana", "orange"]
iterator = iter(fruits)

print(next(iterator))  # apple
print(next(iterator))  # banana
print(next(iterator))  # orange
```

Creating Custom Iterator

```
class CountDown:
    def __init__(self, start):
        self.start = start

    def __iter__(self):
        return self

    def __next__(self):
        if self.start < 0:
            raise StopIteration
        current = self.start
        self.start -= 1
        return current

for num in CountDown(5):
    print(num)  # 5, 4, 3, 2, 1, 0
```

📋 Chapter Summary

- Iterators traverse collections
- `iter()` creates iterator
- `next()` gets next item

- Implement `__iter__` and `__next__`

CHAPTER 30

Error Handling (Try...Except)

Handling Errors Gracefully

Errors happen! Good programs handle them gracefully instead of crashing.

```
try:
    number = int(input("Enter a number: "))
    result = 10 / number
    print(f"Result: {result}")
except ValueError:
    print("That's not a valid number!")
except ZeroDivisionError:
    print("Cannot divide by zero!")
except:
    print("Something went wrong!")
finally:
    print("This always runs")
```

Common Exceptions

Exception	When It Occurs
`ValueError`	Wrong value type
`TypeError`	Wrong operation for type
`ZeroDivisionError`	Division by zero
`FileNotFoundError`	File doesn't exist
`IndexError`	Index out of range
`KeyError`	Dictionary key not found

Raising Exceptions

```python
def set_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative")
    return age

try:
    set_age(-5)
except ValueError as e:
    print(f"Error: {e}")
```

📋 Chapter Summary

- `try` : Code that might cause error
- `except` : Handle specific errors
- `finally` : Always runs (cleanup)
- `raise` : Throw your own exceptions

CHAPTER 31

File Handling

Working with Files

Reading Files

```python
# Reading entire file
with open("myfile.txt", "r") as file:
    content = file.read()
    print(content)

# Reading line by line
with open("myfile.txt", "r") as file:
    for line in file:
        print(line.strip())
```

Writing Files

```python
# Write (overwrites existing)
with open("output.txt", "w") as file:
    file.write("Hello, World!\n")
    file.write("Second line\n")

# Append (adds to end)
with open("output.txt", "a") as file:
    file.write("New line added\n")
```

File Modes

Mode	Description
`"r"`	Read (default)
`"w"`	Write (creates/overwrites)
`"a"`	Append
`"x"`	Create (fails if exists)

`"b"`	Binary mode

> ✅ **Pro Tip**
>
> Always use `with` statement—it automatically closes files!

📋 Chapter Summary

- Use `with open()` for safe file handling
- `"r"` read, `"w"` write, `"a"` append
- `.read()`, `.write()`, `.readline()`

CHAPTER 32

Working with Dates

The datetime Module

```
from datetime import datetime, date, timedelta

# Current date and time
now = datetime.now()
print(now)  # 2024-01-15 14:30:45.123456

# Accessing components
print(now.year)    # 2024
print(now.month)   # 1
print(now.day)     # 15
print(now.hour)    # 14

# Formatting dates
print(now.strftime("%Y-%m-%d"))       # 2024-01-15
print(now.strftime("%B %d, %Y"))     # January 15, 2024
print(now.strftime("%H:%M:%S"))      # 14:30:45
```

Date Arithmetic

```
# Adding/subtracting time
tomorrow = now + timedelta(days=1)
last_week = now - timedelta(weeks=1)

# Difference between dates
date1 = datetime(2024, 1, 1)
date2 = datetime(2024, 12, 31)
diff = date2 - date1
print(diff.days)  # 365
```

📋 Chapter Summary

- `datetime.now()` — current date/time
- `.strftime()` — format dates as strings

- `timedelta` — add/subtract time

CHAPTER 33

Working with JSON

JSON for Data Exchange

JSON (JavaScript Object Notation) is a popular format for storing and exchanging data.

```
import json

# Python dict to JSON string
person = {"name": "Amina", "age": 25}
json_string = json.dumps(person)
print(json_string)  # {"name": "Amina", "age": 25}

# JSON string to Python dict
json_data = '{"name": "Ali", "age": 30}'
person = json.loads(json_data)
print(person["name"])  # Ali
```

Working with JSON Files

```
# Write to JSON file
with open("data.json", "w") as file:
    json.dump(person, file, indent=4)

# Read from JSON file
with open("data.json", "r") as file:
    data = json.load(file)
    print(data)
```

📋 Chapter Summary

- `json.dumps()` — Python to JSON string
- `json.loads()` — JSON string to Python
- `json.dump()` — Write to file

- `json.load()` — Read from file

CHAPTER 34

Regular Expressions

Pattern Matching

```python
import re

text = "Contact us at hello@example.com or support@company.org"

# Find all email addresses
pattern = r"[\w.]+@[\w.]+"
emails = re.findall(pattern, text)
print(emails)  # ['hello@example.com', 'support@company.org']

# Check if pattern matches
if re.search(r"\d+", "Price: 100"):
    print("Found a number!")

# Replace patterns
result = re.sub(r"\s+", " ", "Too   many   spaces")
print(result)  # "Too many spaces"
```

Common Patterns

Pattern	Matches
`\d`	Any digit (0-9)
`\w`	Word character (a-z, A-Z, 0-9, _)
`\s`	Whitespace
`.`	Any character
`*`	0 or more
`+`	1 or more
`?`	0 or 1

`[abc]`	a, b, or c
`^`	Start of string
`$`	End of string

📋 Chapter Summary

- `re.findall()` — Find all matches
- `re.search()` — Find first match
- `re.sub()` — Replace matches
- Use `r"..."` for raw strings

CHAPTER 35

PIP - Package Manager

Installing Packages

PIP is Python's package installer, allowing you to install third-party libraries.

```
# Install a package
pip install requests

# Install specific version
pip install requests==2.28.0

# Upgrade package
pip install --upgrade requests

# Uninstall package
pip uninstall requests

# List installed packages
pip list

# Show package info
pip show requests
```

Requirements File

```
# Create requirements.txt
pip freeze > requirements.txt

# Install from requirements.txt
pip install -r requirements.txt
```

📋 Chapter Summary

- `pip install package` — Install package
- `pip list` — List installed packages

- `pip freeze > requirements.txt` — Export dependencies

CHAPTER 36

Virtual Environments

Isolated Python Environments

Virtual environments allow you to have separate Python installations for different projects.

```
# Create virtual environment
python -m venv myenv

# Activate (Windows)
myenv\Scripts\activate

# Activate (Mac/Linux)
source myenv/bin/activate

# Deactivate
deactivate
```

✅ **Pro Tip**

Best Practice: Always use virtual environments for your projects to avoid dependency conflicts!

📋 Chapter Summary

- `python -m venv name` — Create venv
- `source name/bin/activate` — Activate
- `deactivate` — Exit venv
- Isolates project dependencies

CHAPTER 37

Practice Projects

Project 1: Number Guessing Game

```python
import random

number = random.randint(1, 100)
attempts = 0

print("Guess the number between 1 and 100!")

while True:
    guess = int(input("Your guess: "))
    attempts += 1

    if guess < number:
        print("Too low!")
    elif guess > number:
        print("Too high!")
    else:
        print(f"Correct! You got it in {attempts} attempts!")
        break
```

Project 2: Simple Calculator

```python
def calculator():
    print("Simple Calculator")
    print("Operations: +, -, *, /")

    num1 = float(input("First number: "))
    op = input("Operation: ")
    num2 = float(input("Second number: "))

    if op == "+":
        result = num1 + num2
    elif op == "-":
        result = num1 - num2
    elif op == "*":
        result = num1 * num2
    elif op == "/":
        result = num1 / num2 if num2 != 0 else "Cannot divide by zero"
    else:
        result = "Invalid operation"

    print(f"Result: {result}")

calculator()
```

Project 3: Contact Book

```python
contacts = {}

def add_contact(name, phone):
    contacts[name] = phone
    print(f"Added {name}")

def search_contact(name):
    if name in contacts:
        print(f"{name}: {contacts[name]}")
    else:
        print("Contact not found")

def show_all():
    for name, phone in contacts.items():
        print(f"{name}: {phone}")

# Usage
add_contact("Amina", "123-456-7890")
add_contact("Ali", "098-765-4321")
show_all()
search_contact("Amina")
```

📝 More Project Ideas

- To-Do List Application
- Password Generator
- Quiz Game
- File Organizer
- Simple Web Scraper
- Weather App (using APIs)

FINAL CHAPTER

Summary & Next Steps

🎉 Congratulations!

You've completed this Python programming guide! Let's recap what you've learned:

✅ Python Basics

- Variables, data types, and operators
- Strings, numbers, and type casting
- Comments and code organization

✅ Control Flow

- If/elif/else statements
- Match statements (Python 3.10+)
- While and for loops

✅ Data Structures

- Lists, tuples, sets, and dictionaries
- List comprehensions
- Working with collections

✅ Functions & OOP

- Defining and calling functions
- Classes, objects, and inheritance
- Polymorphism and iterators

✅ Advanced Topics

- Error handling with try/except
- File handling and JSON
- Modules, PIP, and virtual environments

🚀 What's Next?

✅ **Pro Tip**

Keep Learning!

- **Web Development:** Django, Flask
- **Data Science:** Pandas, NumPy, Matplotlib
- **Machine Learning:** Scikit-learn, TensorFlow
- **Automation:** Selenium, Beautiful Soup
- **APIs:** Requests, FastAPI

💡 Final Tips

1. **Practice Daily:** Code every day, even if just for 15 minutes
2. **Build Projects:** Apply what you learn to real projects
3. **Read Code:** Study open-source projects on GitHub
4. **Join Communities:** Stack Overflow, Reddit, Discord
5. **Don't Give Up:** Everyone struggles at first—keep going!

🎉 **Fun Fact**

Remember: Python was named after Monty Python—programming should be fun! Enjoy the journey!

Thank You for Reading!

Now go build something amazing with Python! 🐍✨

ALVIN ROBERT