Python Fundamentals for Beginners

A Practical Guide to Learn Python Step-by-Step

By
Sukhpreet Singh

About This Book: This book is designed for beginners looking to learn Python programming from the ground up. Through a combination of theory and hands-on exercises, readers will develop a solid foundation in Python fundamentals and gain confidence in writing simple Python programs. No prior programming experience is required!

Book Outline:

Introduction to Python

- What is Python?
- Why learn Python?
- Installing Python and setting up the environment
- Writing your first Python program

Basic Python Syntax

- Variables and Data Types (strings, integers, floats, booleans)
- Comments and Code Structure
- Input and Output

Control Flow

- If statements
- Loops: for, while
- Break and continue statements

Functions and Modules

- Defining and calling functions
- Function arguments and return values
- Importing modules and libraries

Data Structures

- Lists, Tuples, Sets, and Dictionaries
- Common operations on data structures (indexing, slicing, looping)

Error Handling

- Exceptions and try-except blocks
- Debugging basics

Object-Oriented Programming

- Classes and Objects
- Methods and attributes
- Inheritance and encapsulation

Working with Files

- Reading and writing files
- Working with different file types (txt, csv, etc.)

Project: Building a Simple Python Program

- Integrating the knowledge gained so far to build a small project, such as a calculator or simple text-based game.

Conclusion and Next Steps

- Where to go from here: suggestions for further learning (e.g., web development, data science, etc.).

Chapter 1: Introduction to Python

1.1 What is Python?

Python is an open-source, high-level programming language created by Guido van Rossum and released in 1991. Its simplicity and readability make it an ideal choice for beginners, and it has a rich ecosystem of libraries for more advanced projects.

Key Points:

- Python is easy to read and write because of its clean syntax.
- It is a general-purpose language, meaning it can be used for many different types of programming, such as web development, data science, scripting, and automation.
- Python supports multiple programming paradigms: object-oriented, functional, and procedural.

```
# A simple Python command

print("Welcome to Python Programming!")
```

1.2 Why Learn Python?

Python is one of the most popular programming languages in the world. Its application in various industries makes it a great choice for career opportunities and problem-solving.

Why Python stands out:

- **Easy to Learn**: Python's syntax is similar to the English language, making it easy for beginners to understand and use.
- **Versatility**: It can be used for a wide range of tasks, from automating scripts to developing complex applications.
- **Large Community**: Python has a huge online community, meaning you can easily find help, tutorials, and open-source libraries.
- **Demand in the Job Market**: Python is heavily used in fields like web development, data science, machine learning, and automation, which are growing rapidly.

Example Use Cases:

- Web Development (e.g., Django, Flask)
- Data Science (e.g., NumPy, Pandas)
- Machine Learning (e.g., TensorFlow, Scikit-learn)
- Automation (e.g., automating daily tasks)

1.3 Installing Python and Setting Up the Environment

Before starting to write Python code, you need to set up Python on your computer.

Step-by-Step Installation Guide:

1. **For Windows:**
   - Download the latest version of Python from python.org.
   - Run the installer, and make sure to check "Add Python to PATH" before clicking "Install."
2. **For macOS:**
   - Python 2.x is usually pre-installed. However, you need to install Python 3.x.
   - Use Homebrew to install Python: Open Terminal and type `brew install python`.
3. **For Linux:**
   - Python is pre-installed on most Linux distributions. To install Python 3, open the terminal and type:
   - sudo apt-get update
   - sudo apt-get install python3

Choosing an IDE (Integrated Development Environment):

- Beginners can use a simple text editor like **Sublime Text** or **VSCode**.
- More advanced IDEs for Python include **PyCharm** and **Jupyter Notebook** (ideal for data science).

Installing Python Packages: You can use `pip`, Python's package manager, to install third-party libraries.

```
pip install package_name
```

1.4 Writing Your First Python Program

Let's write a simple Python program to see how it works.

1. Open your text editor or IDE.
2. Write the following code:

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

3. Save the file with a `.py` extension (e.g., `hello.py`).
4. Run the program by opening your terminal (or command prompt) and navigating to the file's location. Then, type:

Explanation:

- **`print()`** is a built-in function that displays the text inside the parentheses to the screen.

- In Python, statements don’t require semicolons at the end, unlike some other languages.

Hands-on Exercise:

- **Task 1**: Write and run a program that prints "Hello, World!".
- **Task 2**: Experiment with printing different text, such as your name, and use comments to explain your code.

Example:

```
# This program prints my name

print("Hello, my name is Your Name")
```

Chapter 2: Basic Python Syntax

2.1 Variables and Data Types

In Python, variables are used to store data, and each piece of data has a type. Unlike some other programming languages, Python is dynamically typed, meaning you don't need to declare the type of a variable explicitly. Python will infer the type based on the value you assign.

Example:

```
# Assigning values to variables
name = "John"   # String
age = 25        # Integer
height = 5.8    # Float
is_student = True  # Boolean
```

Key Data Types:

- **String (`str`)**: Text data, e.g., `"Hello, World!"`
- **Integer (`int`)**: Whole numbers, e.g., `42`
- **Float (`float`)**: Decimal numbers, e.g., `3.14`
- **Boolean (`bool`)**: True or False values, e.g., `True`, `False`

2.2 Comments and Code Structure

Comments are used to explain code, making it easier to understand. Python supports two types of comments:

- **Single-line comments**: Begin with `#`
- **Multi-line comments**: Enclosed by triple quotes `''' or """`

Example:

```
# This is a single-line comment

"""
This is a
multi-line comment
"""
```

Python code is structured using indentation (spaces or tabs), rather than curly braces or keywords. This means the code that belongs to a block, such as a function or loop, must be indented at the same level.

Example:

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

2.3 Input and Output

Input allows users to provide data to a program, while **Output** displays data to the screen.

- **`input()`**: Used to get input from the user.
- **`print()`**: Used to display text or variables on the screen.

Example:

```
# Input example
name = input("Enter your name: ")

# Output example
print("Hello, " + name)
```

- The `input()` function reads input as a string, so you may need to convert it to another data type.

Example:

```
# Convert input to an integer
age = int(input("Enter your age: "))
```

2.4 Basic Operators

Python supports a variety of operators to perform calculations and logical comparisons.

Arithmetic Operators:

- `+` : Addition
- `-` : Subtraction
- `*` : Multiplication
- `/` : Division
- `**` : Exponentiation
- `%` : Modulus (remainder)

Example:

```
# Arithmetic operations
x = 5
y = 3
result = x + y  # result is 8
```

Comparison Operators:

- `==` : Equal to
- `!=` : Not equal to
- `>` : Greater than
- `<` : Less than
- `>=` : Greater than or equal to
- `<=` : Less than or equal to

Example:
```
# Comparison
print(5 > 3)   # True
print(5 == 3)  # False
```

Logical Operators:

- `and`: Returns `True` if both conditions are `True`
- `or`: Returns `True` if at least one condition is `True`
- `not`: Reverses the result, returns `True` if the result is `False`

Example:
```
# Logical operations
print(True and False)  # False
print(True or False)   # True
```

Hands-on Exercises:

Exercise 1:

- Write a program that asks the user for their name and age, then prints a message like "Hello, [Name], you are [Age] years old!"

Example:
```
# Asking for name and age
name = input("What is your name? ")
age = input("How old are you? ")

# Displaying the output
print("Hello, " + name + ", you are " + age + " years old!")
```

Exercise 2:

- Create a program that takes two numbers as input from the user, adds them, and prints the result.

Example:

```
# Input two numbers
num1 = int(input("Enter the first number: "))
num2 = int(input("Enter the second number: "))

# Add the numbers and print the result
sum = num1 + num2
print("The sum is:", sum)
```

Chapter 3: Control Flow

3.1 If Statements

In Python, **if statements** are used to make decisions. Based on the condition provided, the program will execute certain blocks of code.

Syntax:

```
if condition:
    # Code to execute if condition is True
```

You can also include **else** and **elif** (else-if) statements to handle multiple conditions.

Example:

```
age = 18

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

- **if**: Executes the block of code if the condition is `True`.
- **else**: Executes if the condition in the `if` statement is `False`.
- **elif**: Allows you to check multiple conditions.

Example with `elif`:

```
grade = 85

if grade >= 90:
    print("You got an A.")
elif grade >= 80:
    print("You got a B.")
else:
    print("You need to improve.")
```

3.2 While Loops

A **while loop** repeats a block of code as long as a condition is `True`.

Syntax:

```
while condition:
    # Code to execute repeatedly while condition is True
```

Example:
```
count = 1

while count <= 5:
   print(count)
   count += 1  # This increments count by 1 in each iteration
```

- Be careful with while loops—if the condition never becomes `False`, the loop will run infinitely.

Example of an infinite loop:
```
while True:
   print("This will run forever!")
```

3.3 For Loops

A **for loop** is used to iterate over a sequence (like a list, tuple, string, or range of numbers).

Syntax:
```
for variable in sequence:
   # Code to execute for each element in the sequence
```

Example:
```
# Looping through a list
fruits = ["apple", "banana", "cherry"]

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

Example with `range()`:
```
# Looping through a range of numbers
for i in range(1, 6):  # Loops from 1 to 5
   print(i)
```

3.4 Break and Continue

Sometimes, you need more control over your loops. Python provides the **break** and **continue** statements:

- **break**: Exits the loop entirely.
- **continue**: Skips the current iteration and moves to the next one.

Example of `break`:

```
for i in range(1, 11):
    if i == 5:
        break  # Stops the loop when i is 5
    print(i)
```

Example of `continue`:

```
for i in range(1, 11):
    if i == 5:
        continue  # Skips the iteration when i is 5
    print(i)
```

Hands-on Exercises:

Exercise 1:

- Write a program that asks the user for a number and checks if it is positive, negative, or zero using an `if-elif-else` statement.

Example:

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

if number > 0:
    print("The number is positive.")
elif number < 0:
    print("The number is negative.")
else:
    print("The number is zero.")
```

Exercise 2:

- Create a while loop that prints numbers from 1 to 10.

Example:

```
count = 1

while count <= 10:
    print(count)
    count += 1
```

Exercise 3:

- Write a program that prints all even numbers between 1 and 20 using a for loop.

Example:

```
for i in range(1, 21):
    if i % 2 == 0:  # Check if the number is even
        print(i)
```

Chapter 4: Functions and Modules

4.1 Defining and Calling Functions

A **function** is a block of reusable code that performs a specific task. Functions are useful because they allow you to avoid repeating code and can make programs easier to read.

Syntax:

```
def function_name(parameters):
    # Code block
    return value  # Optional
```

Example:

```
# Defining a function
def greet(name):
    print("Hello, " + name + "!")

# Calling the function
greet("Alice")
```

In this example, `greet` is a function that takes one parameter (`name`). When the function is called with `"Alice"`, it prints "Hello, Alice!".

4.2 Function Arguments and Return Values

You can pass **arguments** to functions and return values from them.

Example with Multiple Arguments:

```
# Function with multiple arguments
def add_numbers(num1, num2):
    return num1 + num2

# Calling the function and using the return value
result = add_numbers(3, 5)
print(result)  # Output: 8
```

In this example, the function `add_numbers` accepts two arguments (`num1` and `num2`) and returns their sum.

4.3 Default and Keyword Arguments

Functions can have **default arguments**, which provide a default value if no argument is passed.

Example:

```
# Function with a default argument
def greet(name="Stranger"):
    print("Hello, " + name + "!")

greet()          # Output: Hello, Stranger!
greet("Alice")   # Output: Hello, Alice!
```

You can also use **keyword arguments**, where you explicitly define which argument you're passing to a function.

Example:

```
def introduction(name, age):
    print(f"My name is {name} and I am {age} years old.")

introduction(age=25, name="Bob")
```

4.4 Scope of Variables

Variables inside a function have **local scope**, meaning they can only be accessed within that function. Variables outside any function have **global scope**.

Example:

```
def my_function():
    local_var = "I am local"
    print(local_var)

my_function()

# This will cause an error because local_var is not defined outside the function
# print(local_var)
```

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

Example:

```
global_var = "I am global"

def my_function():
    global global_var
    global_var = "Modified globally"

my_function()
print(global_var)  # Output: Modified globally
```

4.5 Importing Modules

A **module** is a file containing Python definitions and statements, which you can use in other programs by importing them.

Python has many built-in modules, such as `math` and `random`. You can also create your own modules.

Example: Importing a Built-in Module

```
import math

# Using a function from the math module
result = math.sqrt(16)
print(result)  # Output: 4.0
```

Example: Creating and Importing Your Own Module

1. Create a file called `my_module.py`:

   ```
   def say_hello()

       print("Hello from my module!")
   ```

2. In another Python file, import and use your module:

   ```
   import my_module

   my_module.say_hello()  # Output: Hello from my module!
   ```

You can also import specific functions or variables from a module using:

```
from math import sqrt

result = sqrt(25)

print(result)  # Output: 5.0
```

Hands-on Exercises:

Exercise 1:

- Write a function that takes two numbers as input and returns their product.

  Example:

  ```
  def multiply(num1, num2):
  ```

```
    return num1 * num2


# Calling the function

result = multiply(4, 5)

print(result)  # Output: 20
```

Exercise 2:

- Create a module with a function that greets the user and import it into another file to use it.

1. Create a file named `greetings.py`:

```
def say_hello(name):

    print(f"Hello, {name}!")
```

In another file:

```
import greetings


greetings.say_hello("Alice")
```

Exercise 3:

- Write a function that checks whether a number is even or odd. Use this function to print whether numbers from 1 to 10 are even or odd.

Example:

```
def check_even_odd(number):

    if number % 2 == 0:

        return "Even"

    else:

        return "Odd"

for i in range(1, 11):
    print(f"{i} is {check_even_odd(i)}")
```

Chapter 5: Data Structures

5.1 Lists

A **list** is an ordered collection of items that is mutable (i.e., you can change its content). Lists can hold items of different types, and you can access them using an index.

Example:

```
# Creating a list

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


# Accessing list items

print(fruits[0])  # Output: apple


# Modifying a list

fruits[1] = "blueberry"

print(fruits)  # Output: ['apple', 'blueberry', 'cherry']


# Adding an item to the list

fruits.append("orange")

print(fruits)  # Output: ['apple', 'blueberry', 'cherry', 'orange']
```

Common List Methods:

- `append()`: Adds an item to the end of the list.
- `remove()`: Removes the first occurrence of an item.
- `pop()`: Removes and returns an item at the given index.
- `sort()`: Sorts the list in ascending order.

Example:

```
# Sorting a list
```

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

numbers.sort()

print(numbers)  # Output: [1, 1, 3, 4, 5, 9]
```

5.2 Tuples

A **tuple** is similar to a list, but it is **immutable** (i.e., it cannot be changed after creation). Tuples are often used for data that should not be modified.

Example:

```
# Creating a tuple
coordinates = (4, 5)

# Accessing tuple items
print(coordinates[0])  # Output: 4

# Tuples cannot be modified, so this will cause an error:
# coordinates[0] = 10  # Error!
```

5.3 Sets

A **set** is an unordered collection of unique items. Sets are useful when you need to store distinct elements and don't care about the order.

Example:

```
# Creating a set
unique_numbers = {1, 2, 3, 1, 2}

print(unique_numbers)  # Output: {1, 2, 3} (duplicates are removed)
```

Common Set Operations:

- `add()`: Adds an element to the set.
- `remove()`: Removes a specific element from the set.
- `union()`: Returns the union of two sets.
- `intersection()`: Returns the intersection of two sets.

Example:

```
# Set operations
set_a = {1, 2, 3}
set_b = {3, 4, 5}
```

```
# Union of sets
print(set_a.union(set_b))  # Output: {1, 2, 3, 4, 5}

# Intersection of sets
print(set_a.intersection(set_b))  # Output: {3}
```

5.4 Dictionaries

A **dictionary** is a collection of key-value pairs, where each key is unique. Dictionaries allow for fast lookups and modifications based on keys.

Example:

```
# Creating a dictionary
person = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}

# Accessing dictionary values
print(person["name"])  # Output: Alice

# Adding a new key-value pair
person["job"] = "Engineer"
print(person)  # Output: {'name': 'Alice', 'age': 25, 'city': 'New York', 'job': 'Engineer'}
```

Common Dictionary Methods:

- `keys()`: Returns a list of all the keys.
- `values()`: Returns a list of all the values.
- `items()`: Returns a list of key-value pairs as tuples.
- `get()`: Returns the value for a given key, or a default value if the key doesn't exist.

Example:

```
# Using get() to safely access a key
print(person.get("name"))  # Output: Alice
print(person.get("salary", "Not available"))  # Output: Not available
```

5.5 List Comprehension

List comprehensions provide a concise way to create lists by using an expression and a loop in a single line of code.

Example:

```
# Without list comprehension
squares = []
```

```
for i in range(1, 6):
    squares.append(i ** 2)

# With list comprehension
squares = [i ** 2 for i in range(1, 6)]
print(squares)  # Output: [1, 4, 9, 16, 25]
```

Hands-on Exercises:

Exercise 1:

- Create a list of five of your favorite fruits and print the second item in the list.

Example:

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

Exercise 2:

- Write a program that creates a dictionary to store information about a car (e.g., make, model, year). Then, print the car's model and year.

Example:

```
car = {
    "make": "Toyota",
    "model": "Corolla",
    "year": 2020
}

print("Model:", car["model"])
print("Year:", car["year"])
```

Exercise 3:

- Use a set to store unique values from a list that contains duplicate numbers.

Example:

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

Exercise 4:

- Write a program that uses list comprehension to create a list of the squares of numbers from 1 to 10.

Example:

```
squares = [i ** 2 for i in range(1, 11)]
print(squares)  # Output: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Chapter 6: Error Handling

6.1 What Are Exceptions?

In Python, **exceptions** are errors that occur during the execution of a program. When Python encounters an error, it raises an exception and stops the program unless the error is handled.

Common Types of Exceptions:

- **SyntaxError**: Raised when there's a syntax mistake in the code.
- **TypeError**: Raised when an operation is performed on an inappropriate data type.
- **ValueError**: Raised when a function receives an argument of the correct type but an inappropriate value.
- **IndexError**: Raised when trying to access an index that is out of range in a list or other sequence.
- **KeyError**: Raised when trying to access a key that does not exist in a dictionary.

Example:

```
# This will raise a ZeroDivisionError

result = 10 / 0  # Error: Division by zero
```

6.2 The try-except Block

You can handle exceptions using the **try-except** block, which allows the program to continue running even if an error occurs.

Syntax:

```
try:

        # Code that may raise an exception

except ExceptionType:

         # Code that runs if the exception occurs
```

Example:

```
try:

    result = 10 / 0
```

```
except ZeroDivisionError:

    print("Cannot divide by zero.")
```

In this example, instead of the program crashing, Python catches the `ZeroDivisionError` and prints a custom message.

6.3 Handling Multiple Exceptions

You can handle multiple types of exceptions by specifying multiple `except` blocks.

Example:

```
try:

    number = int(input("Enter a number: "))

    result = 10 / number

except ValueError:

    print("That's not a valid number!")

except ZeroDivisionError:

    print("Cannot divide by zero.")
```

In this example, a `ValueError` will occur if the user inputs something that can't be converted to an integer, and a `ZeroDivisionError` will occur if the user enters 0.

6.4 The else and finally Clauses

- The **else** clause is optional and runs if no exception was raised.
- The **finally** clause runs no matter what, whether an exception occurred or not. It's often used for cleaning up resources like closing files or database connections.

Example:

```
try:

    number = int(input("Enter a number: "))

    result = 10 / number

except ZeroDivisionError:

    print("Cannot divide by zero.")

else:

    print("The result is:", result)
```

```
finally:
    print("This will always be executed.")
```

In this example:

- The `else` block runs if no exception occurs.
- The `finally` block runs regardless of whether an exception occurs or not.

6.5 Raising Exceptions

You can manually raise an exception in your code using the **raise** keyword.

Example:

```
def check_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative.")
    else:
        print("Age is valid.")


try:
    check_age(-5)
except ValueError as e:
    print(e)
```

In this example, if a negative age is provided, a `ValueError` is raised with a custom error message.

Hands-on Exercises:

Exercise 1:

- Write a program that asks the user for two numbers and divides them. Handle the cases where the user inputs something other than a number or tries to divide by zero.

Example:

```
try:
    num1 = float(input("Enter the first number: "))
    num2 = float(input("Enter the second number: "))
    result = num1 / num2
    print("The result is:", result)
except ValueError:
    print("Please enter valid numbers.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

Exercise 2:

- Create a function that checks whether a number is positive. If the number is negative, raise a `ValueError`. Handle this exception and print an appropriate message.

Example:

```
def check_positive(number):
    if number < 0:
        raise ValueError("The number cannot be negative.")
    else:
        print(f"{number} is positive.")


try:
    check_positive(-3)
except ValueError as e:
    print(e)
```

Exercise 3:

- Write a program that reads a file. If the file does not exist, handle the `FileNotFoundError` and print an error message.

Example:

```
try:
    with open("non_existent_file.txt", "r") as file:
        content = file.read()

except FileNotFoundError:

    print("The file was not found.")
```

Chapter 7: Object-Oriented Programming (OOP)

7.1 What is Object-Oriented Programming?

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of **objects**, which can contain both **data** (attributes) and **methods** (functions). It allows for organizing code in a more modular and reusable way.

Key Concepts of OOP:

- **Class**: A blueprint for creating objects.
- **Object**: An instance of a class.
- **Attributes**: Variables that store data specific to an object.
- **Methods**: Functions that define the behavior of an object.

7.2 Defining a Class and Creating Objects

A **class** is like a blueprint or template. Once a class is defined, you can create **objects** (instances) of that class.

Syntax:

```
class ClassName:
    # Class attributes and methods go here
```

Example:
```
# Defining a class
class Dog:
    def __init__(self, name, breed):
        self.name = name  # Attribute
        self.breed = breed  # Attribute

    def bark(self):  # Method
        print(f"{self.name} says woof!")

# Creating an object (instance of the class)
```

```
my_dog = Dog("Buddy", "Golden Retriever")

print(my_dog.name)  # Output: Buddy

my_dog.bark()  # Output: Buddy says woof!
```

- `__init__()`: The constructor method, automatically called when an object is created. It initializes the object's attributes.
- `self`: Refers to the current instance of the class. It's used to access the attributes and methods of the class.

7.3 Methods and the `self` Parameter

Methods are functions defined inside a class. The `self` parameter is a reference to the current instance of the class and must be the first parameter in any method.

Example:

```
class Car:

    def __init__(self, make, model):

        self.make = make

        self.model = model


    def start(self):

        print(f"The {self.make} {self.model} is starting.")


my_car = Car("Toyota", "Corolla")

my_car.start()  # Output: The Toyota Corolla is starting.
```

7.4 Inheritance

Inheritance allows a class to inherit the attributes and methods of another class. This promotes code reuse and hierarchical relationships between classes.

- **Parent (Base) Class**: The class that is inherited from.
- **Child (Derived) Class**: The class that inherits from the parent class.

Syntax:

```
class ChildClass(ParentClass):

    # Child class can add or override methods and attributes
```

Example:

```
class Animal:

    def __init__(self, name):

        self.name = name


    def speak(self):

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


class Dog(Animal):  # Inherits from Animal

    def speak(self):

        print(f"{self.name} barks.")


my_dog = Dog("Buddy")

my_dog.speak()  # Output: Buddy barks.
```

In this example, the `Dog` class inherits from the `Animal` class but overrides the `speak()` method to provide a custom behavior.

7.5 Encapsulation and Access Modifiers

Encapsulation is the concept of bundling the data (attributes) and methods that operate on that data within one class. It also includes restricting direct access to some attributes or methods to prevent accidental modification.

- **Private Attributes**: Attributes prefixed with two underscores (`__`) are private and cannot be accessed directly from outside the class.

Example:

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):
        return self.__balance

account = BankAccount(100)
account.deposit(50)
print(account.get_balance())  # Output: 150
# print(account.__balance)  # This will raise an error: AttributeError
```

7.6 Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common parent class. The most common form of polymorphism in Python is through method overriding.

Example:

```
class Animal:
    def speak(self):
        print("This animal makes a sound.")

class Dog(Animal):
    def speak(self):
        print("The dog barks.")
```

```
class Cat(Animal):
    def speak(self):
        print("The cat meows.")

animals = [Dog(), Cat()]

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

In this example, the `speak()` method behaves differently depending on whether the object is a `Dog` or a `Cat`.

Hands-on Exercises:

Exercise 1:

- Define a class `Person` with attributes `name` and `age`, and a method `greet()` that prints a greeting message. Create an instance of the class and call the method.

Example:

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

    def greet(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")

person = Person("Alice", 30)
person.greet()  # Output: Hello, my name is Alice and I am 30 years old.
```

Exercise 2:

- Create a class `Employee` that inherits from `Person` and adds an additional attribute `job_title`. Override the `greet()` method to include the job title in the greeting.

Example:

```
class Employee(Person):
    def __init__(self, name, age, job_title):
        super().__init__(name, age)  # Call the parent class's constructor
        self.job_title = job_title

    def greet(self):
        print(f"Hello, my name is {self.name}, I am {self.age} years old, and I work as a {self.job_title}.")

employee = Employee("Bob", 28, "Software Developer")
employee.greet()  # Output: Hello, my name is Bob, I am 28 years old, and I work as a Software Developer.
```

Exercise 3:

- Create a class `Shape` with a method `area()` that returns 0. Create two subclasses `Rectangle` and `Circle` that override the `area()` method to calculate the area for each shape.

Example:

```
import math

class Shape:
    def area(self):
        return 0
```

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

    def area(self):
        return self.width * self.height

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius ** 2

rectangle = Rectangle(5, 10)
circle = Circle(7)

print(f"Rectangle area: {rectangle.area()}")  # Output: Rectangle area: 50
print(f"Circle area: {circle.area()}")  # Output: Circle area: 153.93804002589985
```

Chapter 8: Working with Files

8.1 Opening and Closing Files

In Python, you use the `open()` function to open a file. Once you are done working with the file, it's important to close it using the `close()` method to free up system resources.

Syntax for Opening a File:

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

- **`mode`**: Specifies the mode in which the file is opened. Common modes are:
- `"r"`: Read mode (default). Opens the file for reading.
- `"w"`: Write mode. Opens the file for writing (overwrites the file if it exists).
- `"a"`: Append mode. Opens the file for appending data at the end.
- `"b"`: Binary mode. Used for working with binary files.

Example:

```
# Opening and closing a file

file = open("example.txt", "r")

content = file.read()

print(content)

file.close()
```

8.2 Using the **`with`** Statement

The `with` statement simplifies file handling by automatically closing the file after the block of code is executed, even if an error occurs.

Example:

```
# Using the with statement to open a file

with open("example.txt", "r") as file:

    content = file.read()
    print(content)

# No need to manually close the file
```

8.3 Reading Files

There are different methods to read the content of a file:

- `read()`: Reads the entire file as a string.
- `readline()`: Reads one line at a time.
- `readlines()`: Reads all lines and returns them as a list.

Example:

```
# Reading the entire file

with open("example.txt", "r") as file:

    content = file.read()

    print(content)


# Reading line by line

with open("example.txt", "r") as file:

    line = file.readline()

    while line:

        print(line.strip())  # Strip removes extra newline characters

        line = file.readline()


# Reading all lines into a list

with open("example.txt", "r") as file:

    lines = file.readlines()

    for line in lines:

        print(line.strip())
```

8.4 Writing to Files

You can write to a file using the `write()` method. If the file doesn't exist, Python will create it. If the file exists, it will overwrite the content (if opened in `"w"` mode).

Example:

```
# Writing to a file

with open("output.txt", "w") as file:

    file.write("This is a new file.\n")

    file.write("It contains multiple lines of text.\n")
```

Appending to a File:

If you want to add content to an existing file without overwriting it, you can open the file in **append mode** (`"a"`).

Example:

```
# Appending to a file

with open("output.txt", "a") as file:

    file.write("This line is appended to the file.\n")
```

8.5 Working with Binary Files

In addition to text files, Python can handle binary files. Binary files, such as images and audio files, contain data in a format that is not human-readable.

To work with binary files, you open the file in **binary mode** by adding `"b"` to the mode.

Example:

```
# Reading a binary file (e.g., an image)

with open("image.jpg", "rb") as file:

    binary_content = file.read()

    print(binary_content[:10])  # Print the first 10 bytes of the file


# Writing to a binary file
```

```
with open("copy_image.jpg", "wb") as file:
    file.write(binary_content)
```

8.6 Handling File Exceptions

File operations can raise exceptions, such as `FileNotFoundError` if the file doesn't exist. It's important to handle such errors gracefully.

Example:

```
try:
    with open("non_existent_file.txt", "r") as file:
        content = file.read()
except FileNotFoundError:
    print("The file was not found.")
```

Hands-on Exercises:

Exercise 1:

- Write a program that reads a file line by line and prints each line to the console.

Example:

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

Exercise 2:

- Create a program that writes a list of numbers (1 to 10) into a file called `numbers.txt`. Then, write another program to read the file and print the numbers.

Writing to the file:

```
numbers = [str(i) for i in range(1, 11)]
with open("numbers.txt", "w") as file:
    file.write("\n".join(numbers))
```

Reading from the file:

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

Exercise 3:

- Write a program that opens an image file in binary mode and creates a copy of it.

Example:

```
with open("image.jpg", "rb") as file:
    image_data = file.read()


with open("copy_image.jpg", "wb") as file:
    file.write(image_data)
```

Chapter 9: Project: Building a Simple Python Program

9.1 Project Overview

We'll build a **Simple To-Do List Program** that allows users to:

- Add tasks to a to-do list.
- View tasks in the list.
- Mark tasks as completed.
- Save the list to a file and load it when the program starts.

9.2 Project Requirements

The program should:

- Allow the user to add tasks.
- Allow the user to view tasks.
- Allow the user to mark tasks as completed.
- Save tasks to a file and load them when the program starts.

9.3 Step-by-Step Implementation

Step 1: Displaying the Menu

We need a simple menu system that allows the user to choose from a list of options: add a task, view tasks, mark a task as completed, or quit the program.

Example:

```
def display_menu():

    print("\nTo-Do List Menu")

    print("1. Add a task")

    print("2. View tasks")

    print("3. Mark a task as completed")

    print("4. Quit")


# Sample usage of the menu

while True:
    display_menu()

    choice = input("Enter your choice: ")

    if choice == "4":

        break
```

Step 2: Adding Tasks

Next, we will create a function that allows the user to add a task to the to-do list. The tasks will be stored in a list.

Example:

```
def add_task(tasks):

    task = input("Enter a task: ")

    tasks.append({"task": task, "completed": False})

    print(f"Task '{task}' added.")
```

Step 3: Viewing Tasks

We will create a function to display the current tasks in the to-do list. It will show both completed and incomplete tasks.

Example:

```
def view_tasks(tasks):

    if not tasks:

        print("No tasks in the to-do list.")

        return



    print("\nTo-Do List:")

    for index, task in enumerate(tasks, start=1):

        status = "Completed" if task["completed"] else "Incomplete"

        print(f"{index}. {task['task']} - {status}")
```

Step 4: Marking Tasks as Completed

We need a function that allows the user to mark a specific task as completed. The user will input the task number to mark it as done.

Example:

```
def mark_task_completed(tasks):

    view_tasks(tasks)

    try:

        task_num = int(input("Enter the number of the task to mark as completed: "))

        tasks[task_num - 1]["completed"] = True

        print(f"Task '{tasks[task_num - 1]['task']}' marked as completed.")

    except (ValueError, IndexError):

        print("Invalid task number.")
```

Step 5: Saving and Loading Tasks from a File

The program will save the tasks to a file so they are preserved between sessions. When the program starts, it will load the saved tasks from the file.

Saving to a File:

```
import json



def save_tasks(tasks, filename="tasks.json"):

    with open(filename, "w") as file:

        json.dump(tasks, file)

    print("Tasks saved to file.")
```

Loading from a File:

```
def load_tasks(filename="tasks.json"):
    try:
        with open(filename, "r") as file:
            return json.load(file)
    except FileNotFoundError:
        return []
```

9.4 Full Program

Here’s the complete program that integrates all the steps above:

```
import json


# Functions for managing tasks
def display_menu():
    print("\nTo-Do List Menu")
    print("1. Add a task")
    print("2. View tasks")
    print("3. Mark a task as completed")
    print("4. Quit")


def add_task(tasks):
    task = input("Enter a task: ")
    tasks.append({"task": task, "completed": False})
    print(f"Task '{task}' added.")


def view_tasks(tasks):
    if not tasks:
        print("No tasks in the to-do list.")
        return

    print("\nTo-Do List:")
    for index, task in enumerate(tasks, start=1):
        status = "Completed" if task["completed"] else "Incomplete"
        print(f"{index}. {task['task']} - {status}")

def mark_task_completed(tasks):
    view_tasks(tasks)
    try:
        task_num = int(input("Enter the number of the task to mark as completed: "))
        tasks[task_num - 1]["completed"] = True
        print(f"Task '{tasks[task_num - 1]['task']}' marked as completed.")
    except (ValueError, IndexError):
        print("Invalid task number.")

def save_tasks(tasks, filename="tasks.json"):
    with open(filename, "w") as file:
        json.dump(tasks, file)
    print("Tasks saved to file.")

def load_tasks(filename="tasks.json"):
    try:
        with open(filename, "r") as file:
            return json.load(file)
    except FileNotFoundError:
        return []


# Main program
def main():
    tasks = load_tasks()


    while True:
        display_menu()
        choice = input("Enter your choice: ")


        if choice == "1":
            add_task(tasks)
        elif choice == "2":
            view_tasks(tasks)
        elif choice == "3":
            mark_task_completed(tasks)
        elif choice == "4":
            save_tasks(tasks)
            print("Goodbye!")
            break
        else:
            print("Invalid choice. Please try again.")


if __name__ == "__main__":
    main()
```

9.5 Extending the Program (Optional)

You can extend the project by adding additional features such as:

- Deleting tasks.
- Editing tasks.
- Sorting tasks by status or alphabetically.

Hands-on Exercise:

Exercise 1:

- Build this simple to-do list program by following the steps above. Try running it and adding some tasks, viewing them, marking them as completed, and saving the list to a file.

Exercise 2 (Optional Extension):

- Modify the program to allow users to delete tasks from the list.

Chapter 10: Conclusion and Next Steps

10.1 Congratulations!

You've reached the end of the book! By now, you should have a solid understanding of Python programming fundamentals. You've learned about:

- Variables and data types.
- Control flow with if statements and loops.
- Functions and modules.
- Data structures such as lists, tuples, sets, and dictionaries.
- Error handling with try-except blocks.
- Object-Oriented Programming (OOP).
- Working with files.
- How to build a simple Python program from scratch.

You've not only grasped the theory but also applied it with hands-on exercises and a project, giving you the practical experience necessary to confidently write Python programs.

10.2 Where to Go from Here?

Now that you have a strong foundation in Python, there are many exciting directions you can take. Depending on your interests, here are some ideas for what to learn next:

1. Data Science and Machine Learning

Python is one of the most popular languages for data analysis and machine learning. If you're interested in this field, explore:

- **NumPy**: For numerical computing.
- **Pandas**: For data manipulation and analysis.
- **Matplotlib** and **Seaborn**: For data visualization.
- **Scikit-learn**: For machine learning algorithms.

2. Web Development

Python is widely used for building web applications. If you're interested in creating websites or web-based applications, explore:

- **Flask**: A lightweight web framework.
- **Django**: A more full-featured web framework for larger applications.

3. Automation and Scripting

Python is great for automating repetitive tasks, such as file management, web scraping, and sending emails. Look into:

- **Selenium**: For automating web browser interactions.

- **BeautifulSoup**: For web scraping.
- **Requests**: For making HTTP requests.

4. Game Development

Python can also be used for creating simple games or graphical applications. Check out:

- **Pygame**: A set of Python modules designed for writing video games.

10.3 Practice, Practice, Practice!

To truly master Python, the key is to keep practicing. Here are some ways to stay sharp and keep improving your skills:

- **Personal Projects**: Start working on your own projects, no matter how small. This will help you apply what you've learned in new and creative ways.
- **Coding Challenges**: Websites like **LeetCode**, **HackerRank**, and **Codewars** provide coding challenges to help you practice and improve your problem-solving skills.
- **Contribute to Open Source**: Contributing to open-source projects is a great way to work on real-world projects and collaborate with other developers. Check out projects on **GitHub** to get started.

10.4 Final Thoughts

Learning to program is a journey, and you've taken the first big step. As you continue to explore Python and other programming concepts, remember that every challenge you encounter is an opportunity to learn and grow.

Keep experimenting, stay curious, and don't be afraid to make mistakes—they are a crucial part of the learning process. With the foundation you've built, you are well on your way to becoming a proficient Python programmer!

proficient Python programmer!

Thank you for reading!
Happy Coding!
