Book Preview

This is a preview version containing only the first 2 chapters of the book.

Full Book Details:

- **Title:** Hands-On Python Programming
- **Subtitle:** Learn by Building Real-World Applications
- **Author:** Kindson Munonye
- **Edition:** 1st Edition
- **Publisher:** Alkademy Books

To access the complete book with all chapters, additional content, and full features, please purchase the full version.

Thank you for your interest!

Hands-On Python Programming

Learn by Building Real-World Applications

Kindson Munonye

1st Edition

Alkademy Books

December 2025

PREFACE

As a passionate Python developer and educator, I wrote "Hands-On Python Programming" to bridge the gap between theory and practice in learning this powerful language. Over the years, I have witnessed countless learners struggle with the transition from understanding Python's syntax to applying it in real-world scenarios. This book is my response to that challenge, aiming to provide a comprehensive guide that not only teaches Python's core and advanced concepts but also demonstrates their application in building practical projects.

This book is designed for a wide audience: from beginners who have a basic understanding of programming to more experienced developers looking to deepen their Python skills and explore advanced techniques. Whether you are a student, a professional seeking to pivot your career, or an enthusiast eager to expand your programming toolkit, this book is crafted to meet your needs.

The structure of "Hands-On Python Programming" is straightforward yet thorough. We start with the fundamentals—data types, control structures, functions, and object-oriented programming—laying a solid foundation. As we progress, we delve into advanced topics such as decorators, generators, and context managers, revealing the versatility and power of Python. Each chapter is designed as a step-by-step tutorial, guiding you through practical projects in web development, data analysis, and automation, among others.

My approach is deeply hands-on, emphasizing learning by doing. Throughout the book, you'll find problem-solving strategies and best practices that will not only enhance your coding skills but

also improve your ability to write clean, efficient, and maintainable code. My hope is that as you turn these pages, you will find not just a guide to Python, but a companion in your programming journey.

To all the lifelong learners, who never stop exploring, questioning, and growing.

To those who dare to teach themselves, and then teach others.

This book is for you.

Contents

CHAPTER 1

INTRODUCTION TO CORE PYTHON CONCEPTS

1.1 Introduction to Core Python Concepts

Python is a versatile and widely-used programming language known for its simplicity and readability, making it an excellent choice for both beginners and experienced developers. This chapter introduces the fundamental concepts of Python programming, including data types, control structures, functions, and an introduction to object-oriented programming. These core concepts form the foundation upon which more advanced topics will be built in subsequent chapters.

1.1.1 Data Types in Python

Python provides a variety of data types that allow programmers to store and manipulate data in different ways. Understanding these data types is crucial for effective programming.

Numeric Types

Python supports several numeric types, including **integers** and **floating-point numbers**.

- **Integer**: Represents whole numbers without a fractional component. In Python, integers have arbitrary precision.
- **Float**: Represents real numbers and is used for floating-point calculations.

```
# Example of integers
x = 10
y = -5

# Example of floating-point numbers
pi = 3.14159
negative_float = -0.5
```

String Type

Strings in Python are sequences of characters enclosed in single, double, or triple quotes. They are used to handle textual data.

```
# Example of strings
single_quote_string = 'Hello, World!'
double_quote_string = "Python Programming"
triple_quote_string = """This is a
multi-line string."""
```

Boolean Type

The Boolean type in Python has two values: **True** and **False**. It is used for logical operations and comparisons.

```
# Example of boolean values
is_python_fun = True
```

```
is_sky_green = False
```

Collection Types

Python provides several types of collections to store multiple items in a single variable.

- **List**: An ordered, mutable collection of items.
- **Tuple**: An ordered, immutable collection of items.
- **Dictionary**: A collection of key-value pairs, where keys are unique.
- **Set**: An unordered collection of unique items.

```
# Examples of collection types
my_list = [1, 2, 3, 4]
my_tuple = (1, 2, 3, 4)
my_dict = {'name': 'Alice', 'age': 25}
my_set = {1, 2, 3, 4}
```

1.1.2 Control Structures

Control structures allow programmers to dictate the flow of execution in a program. Python offers several control structures, including conditional statements and loops.

Conditional Statements

Python uses the **if**, **elif**, and **else** statements to execute code based on certain conditions.

```
# Example of conditional statements
age = 18

```

```
if age < 18:
    print("You are a minor.")
elif age == 18:
    print("You just became an adult.")
else:
    print("You are an adult.")
```

Loops

Loops are used to execute a block of code repeatedly. Python supports **for** and **while** loops.

- **For Loop**: Iterates over a sequence (e.g., list, tuple) and executes the loop body for each item.
- **While Loop**: Repeatedly executes the loop body as long as a given condition is true.

```
# Example of a for loop
for i in range(5):
    print("Iteration:", i)

# Example of a while loop
count = 0
while count < 5:
    print("Count:", count)
    count += 1
```

1.1.3 Functions

Functions in Python are defined using the **def** keyword, followed by the function name and parameters. Functions allow for code reusability and organization.

```python
# Example of a function
def greet(name):
    """This function greets the person passed in as a parameter."""
    print("Hello, " + name + "!")

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

Functions can return values using the **return** statement.

```python
# Function with a return value
def add(a, b):
    """This function returns the sum of two numbers."""
    return a + b

# Using the function
result = add(5, 3)
print("Sum:", result)
```

1.1.4 Introduction to Object-Oriented Programming

Object-oriented programming (OOP) is a paradigm that uses "objects" to model real-world entities. Python supports OOP and enables developers to create classes and objects.

Classes and Objects

A **class** is a blueprint for creating objects, and an **object** is an instance of a class. Classes encapsulate attributes and methods that define the behavior of their objects.

```python
# Example of a class
class Dog:
    # Class attribute
    species = "Canis familiaris"

    # Initializer / Instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def description(self):
        return f"{self.name} is {self.age} years old."

# Creating objects
dog1 = Dog("Buddy", 5)
dog2 = Dog("Molly", 3)

# Accessing attributes and methods
print(dog1.description())
print(dog2.description())
```

Inheritance

Inheritance allows one class (child class) to inherit attributes and methods from another class (parent class), promoting code reuse.

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

```

```
    def speak(self):
        return "Some sound"

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

# Using inheritance
cat = Cat("Whiskers")
print(cat.name)
print(cat.speak())
```

1.2 Conclusion

This chapter introduced the core Python concepts necessary for programming in Python, including data types, control structures, functions, and a brief overview of object-oriented programming. These foundational elements are essential for writing efficient and effective Python code, and they serve as the groundwork for exploring more advanced topics in the subsequent chapters. As you continue your Python journey, these concepts will become tools you can leverage to tackle a variety of programming challenges.

CHAPTER 2

ADVANCED PYTHON TECHNIQUES

2.1 Introduction to Advanced Python Techniques

In this chapter, we explore some of the more advanced features of Python that set it apart as a powerful and versatile programming language. Python, with its simple syntax and readability, also offers advanced constructs like **decorators**, **generators**, and **context managers**. These features not only help in writing clean and efficient code but also enhance your ability to tackle complex programming challenges.

2.2 Decorators

2.2.1 What are Decorators?

Decorators are a powerful feature in Python that allow the modification of functions or methods using other functions. They are often used to add functionality to existing code in a clean and manageable way. Essentially, decorators are functions that return other functions.

2.2.2 Creating a Simple Decorator

To understand decorators, let's start with a basic example:

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

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

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

In this example, `my_decorator` is a decorator function that takes another function `func` as an argument, wraps it in another function `wrapper`, and returns the `wrapper`. When `say_hello` is called, the output is:

```
Something is happening before the function is called.
Hello!
Something is happening after the function is called.
```

2.2.3 Using the @ Syntax

Python provides a cleaner syntax for decorators using the `@` symbol. Here's how you can achieve the same functionality using the decorator syntax:

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

```

say_hello()
```

This syntax is not only cleaner but also makes it clear that `say_hello` is being modified by `my_decorator`.

2.2.4 Decorators with Arguments

Decorators can also take arguments themselves. Let's see how this can be achieved:

```
def repeat(num_times):
    def decorator_repeat(func):
        def wrapper(*args, **kwargs):
            for _ in range(num_times):
                func(*args, **kwargs)
        return wrapper
    return decorator_repeat

@repeat(num_times=3)
def greet(name):
    print(f"Hello, {name}")

greet("Alice")
```

In this example, `repeat` is a decorator factory that takes an argument `num_times`. It returns a decorator `decorator_repeat` that repeats the execution of the decorated function `greet` three times.

2.3 Generators

2.3.1 Introduction to Generators

Generators are a special class of functions that simplify the task of writing iterators. They allow you to declare a function that behaves like an iterator, i.e., it can be used in a `for` loop. Generators are written just like regular functions but use the `yield` statement whenever they want to return data.

2.3.2 Creating a Generator

Here's a simple generator example that yields a sequence of numbers:

```
def count_up_to(max):
    count = 1
    while count <= max:
        yield count
        count += 1

counter = count_up_to(5)
for number in counter:
    print(number)
```

The output of this code will be:

1
2
3
4
5

2.3.3 Generator Expressions

Generator expressions provide an additional way to create generators in a more concise manner. They are similar to list comprehensions but use parentheses instead of square brackets.

```
squared_numbers = (x * x for x in range(10))
for number in squared_numbers:
    print(number)
```

This code will print the squares of numbers from 0 to 9.

2.3.4 Advantages of Generators

Generators have several advantages:

- **Memory Efficiency**: Generators do not store the entire sequence in memory; they generate items on-the-fly.
- **Lazy Evaluation**: They compute values only when needed, which can lead to performance improvements for large datasets.
- **Composable Functions**: Generators can be composed together, allowing for complex data processing pipelines.

2.4 Context Managers

2.4.1 Understanding Context Managers

Context managers are a way to allocate and release resources precisely when you need to. The most common way to use context managers is via the `with` statement. In Python, context managers are implemented using classes or functions with `__enter__` and `__exit__` methods.

2.4.2 Using Context Managers

A typical use case for a context manager is managing a file:

```
with open('file.txt', 'w') as file:
    file.write('Hello, world!')
```

This ensures that the file is properly closed after its suite finishes, even if an exception is raised.

2.4.3 Creating a Custom Context Manager

Let's create a simple context manager that measures the execution time of a block of code:

```
import time

class Timer:
    def __enter__(self):
        self.start = time.time()
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        self.end = time.time()
        print(f"Elapsed time: {self.end - self.start:.2f} seconds")

with Timer():
    # Simulate a long-running operation
    time.sleep(2)
```

When the `with` block is executed, the `__enter__` method starts the timer, and the `__exit__` method stops the timer and prints the elapsed time.

2.4.4 The `contextlib` Module

Python's `contextlib` module provides utilities for creating and working with context managers. One such utility is the `contextmanager` decorator, which makes it easier to create context

managers from generator functions.

```
from contextlib import contextmanager

@contextmanager
def open_file(name, mode):
    f = open(name, mode)
    try:
        yield f
    finally:
        f.close()

with open_file('file.txt', 'w') as f:
    f.write('Hello, world!')
```

In this example, `open_file` is a generator function that uses `yield` to return the file object. The `contextmanager` decorator transforms it into a context manager.

2.5 Conclusion

This chapter covered some of the advanced features of Python, including decorators, generators, and context managers. These constructs are vital for writing efficient and clean Python code. Decorators allow you to add functionality to existing functions, generators enable efficient iteration over sequences, and context managers provide a clean way to manage resources. By mastering these techniques, you can greatly enhance your Python programming skills and tackle more complex problems with ease.