Introduction to Python [1]

International Islamia School Otsuka [2]

Abdulmalek Salem Shefat[3]

July 15, 2024

[1] https://www.python.org
[2] https://www.iiso-edu.jp
[3] malekshefat@gmail.com

Contents

Preface

Welcome to the world of Python programming! Whether you are a seasoned developer or a complete beginner, this book aims to provide you with a comprehensive understanding of Python, a versatile and powerful programming language that has become one of the most popular in the world.

Python was created in the late 1980s by Guido van Rossum and has since grown into a language that powers some of the world's most complex and fascinating systems. From web development and data analysis to artificial intelligence and scientific computing, Python's simplicity and readability make it an ideal choice for a wide range of applications.

Why Python?

Python stands out for several reasons:

- **Readability and Simplicity**: Python's syntax is clear and straightforward, which makes it easy to learn and use. This emphasis on readability reduces the cost of program maintenance and allows developers to collaborate more effectively.
- **Versatility**: Python can be used for web development, automation, scientific computing, data analysis, artificial intelligence, and more. Its extensive standard library and vibrant ecosystem of third-party packages provide tools for virtually any task.
- **Community and Support**: Python boasts a large and active community. This means that you have access to a wealth of resources, including documentation, tutorials, forums, and conferences, which can help you solve problems and stay updated with the latest developments.
- **Integration and Scalability**: Python can easily integrate with other languages and technologies, making it suitable for building scalable and complex applications. It is often used as a "glue" language to connect different systems and components.

Who Is This Book For?

This book is designed for anyone interested in learning Python, including:

- **Beginners**: If you are new to programming, Python is an excellent starting point. This book will guide you through the basics and help you build a solid foundation.
- **Intermediate Programmers**: If you have some programming experience but are new to Python, this book will help you transition smoothly and deepen your understanding of the language.
- **Advanced Users**: Even experienced Python programmers can benefit from this book by exploring advanced topics and best practices.

What Will You Learn?

Throughout this book, you will:

- Understand the core principles of Python programming.
- Learn to write clean and efficient code.
- Explore Python's built-in data structures and libraries.
- Develop real-world applications through hands-on projects.
- Discover advanced concepts and techniques to optimize your code.

How to Use This Book

Each chapter builds on the previous one, so it is recommended to read them in order. However, if you are already familiar with certain topics, feel free to skip ahead to the sections that interest you the most.

The book includes numerous examples and exercises to reinforce your learning. I encourage you to try them out and experiment with the code. Programming is best learned by doing, so don't hesitate to get your hands dirty!

Acknowledgements

Writing this book has been a collaborative effort, and I would like to thank everyone who contributed their time, knowledge, and support. Special thanks to the Python community for their invaluable resources and to my family and friends for their encouragement and patience.

Final Thoughts

Python is more than just a programming language; it is a gateway to endless possibilities. I hope this book inspires you to explore, create, and innovate with Python. Happy coding!

Structure of book

Each unit will focus on a specific problem. While trying to solve that problem, you will start recognizing the patterns of python and Object Oriented Programming (OOP). At the end of each chapter, there will be some exercises to test your understanding of the topics, general code structure and problem-solving skills.

Abdulmalek Salem Shefat
`https://www.linkedin.com/in/mrintj`

1

Introduction to Python

"The joy of coding Python should be in seeing short, concise, readable classes that express a lot of action in a small amount of clear code – not in reams of trivial code that bores the reader to death."- Guido van Rossum

1.1 Introduction

Python is a high-level, interpreted programming language known for its simplicity and readability. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming. This chapter will introduce you to Python's basic concepts and data types, setting the foundation for more advanced topics.

1.2 Variables

In Python, variables are used to store data that can be referenced and manipulated later. Variables do not need explicit declaration to reserve memory space. The declaration happens automatically when a value is assigned to a variable.

1.3 Data Types

Python supports various data types. In this section, we will cover the fundamental ones: integers, floats, booleans, strings, lists, tuples, and dictionaries.

1.3.1 Integers

Integers are whole numbers, positive or negative, without decimals. In Python, you can assign an integer to a variable as follows:

```
1 x = 10
2 y = -5
```

1.3.2 Floats

Floats, or floating-point numbers, are numbers with decimals. Here is how you assign a float to a variable:

```
pi = 3.14
gravity = 9.81
```

1.3.3 Booleans

Booleans represent one of two values: `True` or `False`. They are often used in conditional statements. Assign a boolean value to a variable like this:

```
is_sunny = True
is_raining = False
```

1.3.4 Strings

Strings are sequences of characters, enclosed in single, double, or triple quotes. You can assign a string to a variable as follows:

```
greeting = 'Hello, World!'
name = "Alice"
multiline = '''This is a
multi-line string.'''
```

1.3.5 Lists

Lists are ordered collections of items, which can be of different types. Lists are mutable, meaning their content can be changed after creation. Define a list like this:

```
fruits = ['apple', 'banana', 'cherry']
numbers = [1, 2, 3, 4, 5]
mixed = [1, 'two', 3.0, True]
```

You can access list items by their index, starting from 0:

```
first_fruit = fruits[0]  # 'apple'
second_number = numbers[1]  # 2
```

1.3.6 Tuples

Tuples are similar to lists but are immutable, meaning their content cannot be changed after creation. Tuples are defined using parentheses:

```
coordinates = (10.0, 20.0)
rgb_color = (255, 0, 0)
```

You can access tuple items by their index as well:

```
x_coord = coordinates[0]  # 10.0
red = rgb_color[0]  # 255
```

1.3.7 Dictionaries

Dictionaries are collections of key-value pairs. They are unordered, mutable, and indexed by keys, which can be of any immutable type. Define a dictionary like this:

```
student = {
    'name': 'Alice',
    'age': 21,
    'is_enrolled': True
}
```

You can access dictionary values by their keys:

```
student_name = student['name']  # 'Alice'
student_age = student['age']  # 21
```

1.4 Exercises

1. Write a Python program that initializes an integer variable and prints its value.
2. Create a float variable representing the value of pi (3.14159) and print it.
3. Define a boolean variable `is_sunny` and set it to `True`. Print its value.
4. Create a string variable `name` and assign your name to it. Print the string.
5. Initialize a list called `fruits` with three fruit names. Print the list.
6. Create a tuple `coordinates` with latitude and longitude values. Print the tuple.
7. Define a dictionary `student_info` with keys `'name'`, `'age'`, and `'grade'`. Print the dictionary.
8. Write a program that checks if a given variable is of type integer.
9. Create a list of numbers and print the length of the list.

2

Object-Oriented Programming in Python

2.1 Variables

Variables in programming languages are used to store data that can be referenced and manipulated. In Python, variables are dynamically typed, meaning their type is inferred at runtime.

2.1.1 Example with Lists and Tuples

In Python, lists are mutable (can change) while tuples are immutable (cannot change).

```
# List variable (mutable)
fruits = ["apple", "banana", "cherry"]
print(f"Fruits list: {fruits}")  # Print initial list

# Change value in list
fruits[0] = "orange"
print(f"Fruits list after change: {fruits}")  # Print list after change

# Tuple variable (immutable)
coordinates = (50.0, 100.5)
print(f"Coordinates tuple: {coordinates}")  # Print initial tuple

# Attempt to change value in tuple (results in error)
# coordinates[0] = 55.0  # Uncomment to see TypeError
```

In this example:

1. The initial state of the ‘fruits‘ list is ‘["apple", "banana", "cherry"]‘, which is printed as "Initial fruits list".
2. After changing the first element of the ‘fruits‘ list to ‘"orange"‘, it is printed again as "Fruits list after change".
3. The initial state of the ‘coordinates‘ tuple is ‘(50.0, 100.5)‘, printed as "Initial coordinates tuple".
4. An attempt to change its value (commented out for clarity) would result in a ‘TypeError‘, indicating that tuples are immutable.

2.2 Functions

Functions are blocks of code that perform a specific task. They are defined using the `def` keyword in Python.

2.2.1 Examples of Functions

```
# Function without parameters
def greet():
    """Prints a greeting message."""
    print("1. Hello, welcome!")

greet()  # Call the greet function

# Function with parameters and return value
def add_numbers(a, b):
    """Adds two numbers and returns the result."""
    return a + b


print(f"2. Result of addition: {result}")

# Function with default parameter
def greet_person(name="Guest"):
    """Greets a person by name."""
    print(f"3. Hello, {name}!")

greet_person("Alice") # Call greet_person with argument
greet_person() # Call greet_person without argument (uses
↪  default)

# Function with variable-length arguments
```

```python
def multiply(*args):
    """Multiplies any number of arguments together."""
    result = 1
    for num in args:
        result *= num
    return result

product = multiply(2, 3, 4)  # Call multiply function with multiple arguments
print(f"4. Result of multiplication: {product}")
```

In this example:

1. The ‘greet‘ function prints a greeting message when called.
2. The ‘add_numbers‘ function adds two numbers and returns the result, which is then printed.
3. The ‘greet_person‘ function greets a person by name. It can be called with or without an argument.
4. The ‘multiply‘ function accepts any number of arguments and multiplies them together. The result is printed after calling the function.

2.3 Classes

Classes are blueprints for creating objects. They encapsulate data (attributes) and behaviors (methods). The `__init__()` method initializes the object's attributes.

2.3.1 Examples of Classes

```python
# Define a Book class
class Book:
    def __init__(self, title, author):
        """Initialize the Book object with title and author."""
        self.title = title  # Initialize title attribute
        self.author = author  # Initialize author attribute

    def display_info(self):
        """Display information about the book."""
        print(f"1. Title: {self.title}")
        print(f"   Author: {self.author}")
```

```
12
13 # Create Book objects
14 book1 = Book("Python Programming", "John Smith")  # Create
   ↪  first Book object
15 book2 = Book("Data Structures in Python", "Alice Brown")  #
   ↪  Create second Book object
16
17 # Use object methods to display book information
18 book1.display_info()  # Display information for book1
19 book2.display_info()  # Display information for book2
```

In this example:

1. The ‘Book‘ class is defined with an ‘___init___‘ method that initializes the ‘title‘ and ‘author‘ attributes when a ‘Book‘ object is created.

2. The ‘display_info‘ method prints the title and author of a ‘Book‘ object.

3. Two ‘Book‘ objects (‘book1‘ and ‘book2‘) are created with different titles and authors.

4. The ‘display_info‘ method is called on each ‘Book‘ object to print its information.

2.4 Objects and OOP

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around objects and data, rather than actions and logic. It provides several benefits, including:

- **Modularity**: Encapsulation allows objects to be independently developed and tested.

- **Reuse**: Objects can be reused in different parts of the program or in different programs.

- **Flexibility**: OOP supports dynamic and polymorphic behavior, enabling code to be more adaptable to changing requirements.

2.4.1 Example: Book, Library, and Member

Let’s illustrate OOP concepts with an example involving a Book, Library, and Member:

The ‘Book‘ class represents a book with attributes ‘title‘, ‘author‘, and ‘borrowed_by‘. It has methods ‘display_info‘ to print its information, ‘borrow‘ to allow a member to borrow it, and ‘return_book‘ to return it to the library.

```
# Define a Book class
class Book:
    def __init__(self, title, author):
        """Initialize the Book object with title and
        ↪  author."""
        self.title = title  # Initialize title attribute
        self.author = author  # Initialize author attribute


    def display_info(self):
        """Display information about the book."""
        print(f"1. Title: {self.title}")
        print(f"   Author: {self.author}")
        if self.borrowed_by:
            print(f"   Borrowed by:
            ↪  {self.borrowed_by.name}")

    def borrow(self, member):
        """Allow a member to borrow the book."""
        if not self.borrowed_by:
            self.borrowed_by = member
            print(f"2. '{self.title}' has been borrowed by
            ↪  {member.name}.")
        else:
            print(f"2. '{self.title}' is already borrowed by
            ↪  {self.borrowed_by.name}.")

    def return_book(self):
        """Return the book to the library."""
        if self.borrowed_by:
            print(f"3. '{self.title}' has been returned by
            ↪  {self.borrowed_by.name}.")
            self.borrowed_by = None
        else:
            print(f"3. '{self.title}' is not currently
            ↪  borrowed.")
```

The ‘Library‘ class manages a collection of books (‘books‘ list) with methods ‘add_book‘ to add books and ‘display_books‘ to display informa-

tion about all books.

```
# Define a Library class
class Library:
    def __init__(self):
        """Initialize the Library object with an empty list
        ↪ of books."""
        self.books = []  # Initialize an empty list of books

    def add_book(self, book):
        """Add a Book object to the library."""
        self.books.append(book)

    def display_books(self):
        """Display information about all books in the
        ↪ library."""
        print("4. Books in the Library:")
        for book in self.books:
            book.display_info()
```

The ‘Member‘ class represents a library member with attributes ‘name‘ and ‘borrowed_books‘. It has methods ‘borrow_book‘ to borrow a book from the library and ‘return_book‘ to return a borrowed book.

```
# Define a Member class
class Member:
    def __init__(self, name):
        """Initialize the Member object with a name."""
        self.name = name  # Initialize name attribute
        self.borrowed_books = []  # Initialize an empty list
        ↪ of borrowed books

    def borrow_book(self, library, book_title):
        """Borrow a book from the library."""
        for book in library.books:
            if book.title == book_title:
                book.borrow(self)
                self.borrowed_books.append(book)
                return
        print(f"5. {self.name} could not find '{book_title}'
        ↪ in the library.")

    def return_book(self, library, book_title):
        """Return a borrowed book to the library."""
```

```
            for book in self.borrowed_books:
                if book.title == book_title:
                    book.return_book()
                    self.borrowed_books.remove(book)
                    return
            print(f"6. {self.name} did not borrow
            ↪  '{book_title}'.")
```

The '___main___' block demonstrates the usage of these classes, creating instances of 'Book', 'Library', and 'Member', and performing actions such as adding books to the library, borrowing a book, and returning it, with output statements numbered to indicate the sequence of actions.

```
# Usage example:
if __name__ == "__main__":
    # Create Library object
    library = Library()

    # Create Book objects and add to library
    book1 = Book("Python Programming", "John Smith")
    book2 = Book("Data Structures in Python", "Alice Brown")
    library.add_book(book1)
    library.add_book(book2)

    # Display all books in the library
    library.display_books()

    # Create Member object
    member1 = Member("Alice")

    # Member borrows a book
    member1.borrow_book(library, "Python Programming")

    # Display updated library
    library.display_books()

    # Member returns the borrowed book
    member1.return_book(library, "Python Programming")

    # Display updated library
    library.display_books()
```

Exercises: Object-Oriented Programming (OOP)

1. **Bank Account Management System**
   - Design a ‘BankAccount‘ class that models a bank account with attributes for account number, account holder name, and balance.
   - Implement methods to deposit money, withdraw money (with validation for sufficient balance), and display account details.
   - Consider edge cases such as minimum balance requirements or overdraft protection.
2. **Online Bookstore System**
   - Create classes for ‘Book‘, ‘Author‘, and ‘Customer‘ in an online bookstore system.
   - The ‘Book‘ class should have attributes like title, author(s), price, and availability.
   - Implement methods in the ‘Customer‘ class to add books to a shopping cart, calculate total price, and place orders.
   - Use inheritance or composition to model relationships between classes (e.g., an author can have multiple books).
3. **Inventory Management System for a Store**
   - Define classes for ‘Product‘, ‘Inventory‘, and ‘Supplier‘ to manage inventory for a store.
   - The ‘Product‘ class should have attributes like name, price, quantity, and supplier details.
   - Implement methods in the ‘Inventory‘ class to add/remove products, update quantities, and generate reports (e.g., low stock alerts).
   - Use composition to handle relationships between products and suppliers.
4. **Social Media Platform**
   - Design classes for ‘User‘, ‘Post‘, and ‘Comment‘ in a social media platform.
   - The ‘User‘ class should have attributes like username, email, and posts/comments history.
   - Implement methods in the ‘Post‘ class to create posts with content and manage comments.
   - Use inheritance or composition to model interactions and relationships between users, posts, and comments.

5. **Movie Ticket Booking System**

   - Create classes for ‘Movie‘, ‘Theater‘, ‘Ticket‘, and ‘Customer‘ to manage a movie ticket booking system.
   - The ‘Movie‘ class should have attributes like title, genre, duration, and showtimes.
   - Implement methods in the ‘Theater‘ class to display available movies, book tickets, and manage seating arrangements.
   - Use composition to model relationships between movies, theaters, and ticket bookings.

2.5 Python Modules and Import

The following example demonstrates how to import and use the `Tello` class to control a Tello drone using the `djitellopy` library:

```
'''
in the terminal run:
pip install djitellopy
to install the djitellopy library
'''
from djitellopy import Tello  # Import Tello class from djitellopy library
import time  # Import time module for sleep function

# Initialize Tello object
tello = Tello()

# Connect to Tello drone
tello.connect()
print(f"Connected to Tello: {tello.get_battery()}% battery")  # Print battery status

# Take off
tello.takeoff()
print("Drone is taking off...")

# Let the drone hover for 5 seconds
time.sleep(5)

# Move up by 50 cm
tello.move_up(50)
print("Drone moving up by 50 cm...")
```

```
26
27  # Wait for 2 seconds
28  time.sleep(2)
29
30  # Move left by 30 cm
31  tello.move_left(30)
32  print("Drone moving left by 30 cm...")
33
34  # Wait for 2 seconds
35  time.sleep(2)
36
37  # Rotate clockwise by 90 degrees
38  tello.rotate_clockwise(90)
39  print("Drone rotating clockwise by 90 degrees...")
40
41  # Wait for 2 seconds
42  time.sleep(2)
43
44  # Land the drone
45  tello.land()
46  print("Drone is landing...")
47
48  # Disconnect from Tello drone
49  tello.end()
50  print("Disconnected from Tello.")
```

In this example:

- The `Tello` class encapsulates operations for controlling a Tello drone using the `djitellopy` library.
- Methods such as `connect()`, `takeoff()`, `land()`, `move_up(distance)`, etc., are defined to perform specific actions with the drone.
- The script demonstrates how to instantiate the `Tello` class, connect to the drone, perform flight maneuvers, and disconnect.

2.6 Homework

Creating a Class (Easy)

Task: Create a class called `Car` with attributes `make`, `model`, and `year`. Create an instance of the `Car` class and print out its attributes.

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

my_car = Car(..., ..., ...)
print(...)
```

Adding Methods to a Class (Easy-Intermediate)

Task: Extend the `Car` class to include a method called `display_info` that prints out the car's details. Create an instance and call this method.

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

    def display_info(self):
        # print the car info in a nice and readable way
        ...

my_car = Car(..., ..., ...)
my_car.display_info()
```

Class with Constructor and List Attribute (Intermediate)

Task: Create a class `Garage` that has an attribute `cars` which is a list of `Car` objects. Add methods to add cars to the garage and to list all cars. Create a few `Car` objects and add them to the `Garage`, then list them.

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

    def display_info(self):
        ...

class Garage:
    def __init__(self):
        self.cars = []
```

```
    def add_car(self, car):
        ...

    def list_cars(self):
        # print all cars in a nice and readable way
        ...

my_garage = Garage()
car1 = Car(..., ..., ...)
car2 = Car(..., ..., ...)
my_garage.add_car(car1)
my_garage.add_car(car2)
my_garage.list_cars()
```

Encapsulation (Intermediate-Advanced)

Task: Modify the `Car` class to include a private attribute `_mileage` and provide methods to set and get the mileage, ensuring it cannot be set to a negative value. Update the `Garage` class to handle mileage. Create instances and test these methods.

```
class Car:
    def __init__(self, maker, model, year):
        self._mileage = 0
        ...

    def display_info(self):
        ...

    def set_mileage(self, mileage):
        ...

    def get_mileage(self):
        ...

class Garage:
    def __init__(self):
        self.cars = []

    def add_car(self, car):
        ...

    def list_cars(self):
```

```
        ...

my_garage = Garage()
car1 = Car('Toyota', 'Corolla', 2020)
car1.set_mileage(15000)
car2 = Car('Honda', 'Civic', 2019)
car2.set_mileage(20000)
my_garage.add_car(car1)
my_garage.add_car(car2)
my_garage.list_cars()
```

Simple Project (Advanced)

Task: Create a class `GarageOwner` with attributes `name` and a `Garage`. Add methods to add cars to the garage and display all cars owned by the garage owner. Create instances of `GarageOwner`, `Garage`, and `Car`, and demonstrate the complete functionality.

```
class Car:
    def __init__(self, make, model, year):
        self._mileage = 0
        ...

    def display_info(self):
        ...

    def set_mileage(self, mileage):
        ...

    def get_mileage(self):
        ...

class Garage:
    def __init__(self):
        self.cars = []

    def add_car(self, car):
        ...

    def list_cars(self):
        ...

class GarageOwner:
    def __init__(self, name):
```

```
        self.name = name
        self.garage = Garage()

    def add_car_to_garage(self, car):
        ...

    def display_garage(self):
        ...

owner = GarageOwner('Alex')
car1 = Car('Toyota', 'Corolla', 2020)
car1.set_mileage(15000)
car2 = Car('Honda', 'Civic', 2019)
car2.set_mileage(20000)
owner.add_car_to_garage(car1)
owner.add_car_to_garage(car2)
owner.display_garage()
```