Master Python: From Basics to Advanced

Learn, Build, and Deploy Real-World Python Applications

By Pranav Jha

August 20, 2025

Contents

1 Introduction to the Course

1.1 Course Overview

Welcome to **Master Python: From Basics to Advanced**! This course is designed to take you on a complete journey through Python programming, starting from fundamental concepts and gradually progressing to advanced topics and real-world applications.

Python is one of the most popular and versatile programming languages in the world. It is widely used in web development, data science, machine learning, artificial intelligence, automation, and more. Its simple syntax and readability make it an excellent choice for beginners, while its powerful libraries and frameworks make it suitable for advanced users.

1.2 Learning Objectives

By the end of this course, you will be able to:

- Understand and write Python code using variables, data types, and operators.
- Create and use functions, loops, and conditional statements effectively.
- Work with Python data structures such as lists, tuples, sets, and dictionaries.
- Apply object-oriented programming concepts, including classes, inheritance, and polymorphism.
- Handle file operations and manage data efficiently.
- Use Python for real-world applications, including web development, data analysis, and automation.
- Explore advanced topics like error handling, decorators, generators, and Python libraries.

1.3 Who This Course is For

This course is ideal for:

- Beginners who want to start programming with Python.
- Intermediate programmers looking to strengthen their Python skills.
- Professionals aiming to apply Python in data science, AI, or web development.

1.4 Expected Outcomes

By completing this course, you will:

- Gain a strong foundation in Python programming.
- Be capable of building small to medium-scale Python applications.
- Be ready to explore specialized domains like AI, machine learning, or data analytics using Python.
- Develop problem-solving and logical thinking skills essential for programming.

2 Setting up Python Environment in VS Code

Hello everyone, and welcome to the Python series! In this session, we will begin by setting up our Python environment in **VS Code**.

2.1 Why VS Code?

I find VS Code an amazing IDE because it supports:

- Running both `.py` and Jupyter Notebook (`.ipynb`) files.
- Easy creation and management of environments.
- Extensions, code assistance, and many more productivity tools.

2.2 Choosing the Python Version

For this series, I will use the newest Python version available at the time: **Python 3.12**. We will explore features introduced from Python 3.10 to 3.12.

Note

Always create a separate environment for each new project. This prevents dependency conflicts and allows different projects to use different package versions.

2.3 Creating a New Environment with Conda

We will use the `conda` command to create our virtual environment.

```
conda create -p venv python==3.12
```

Listing 2.1: Creating a Python 3.12 environment in Conda

Here:

- `venv` is the environment name.
- `python==3.12` specifies the version.

When prompted, press `Y` to proceed with installation. Basic libraries will be installed automatically.

2.4 Activating the Environment

To activate the environment:

```
conda activate venv
```

Once activated, you can run Python files directly:

```
python app.py
```

2.5 Working with Jupyter Notebooks in VS Code

1. Create a new folder (e.g., `Python_Basics`). 2. Inside, create a Jupyter Notebook file (`test.ipynb`). 3. Select the kernel: choose the `venv` environment (Python 3.12). 4. Create **Code** cells and **Markdown** cells as needed.

Note

Execute a cell by pressing `Shift+Enter`.

2.5.1 Installing Required Packages

To run notebooks, you need `ipykernel`:

```
pip install ipykernel
```

You can maintain a `requirements.txt` file to keep track of all packages:

```
ipykernel
pandas
numpy
# Add more as needed
```

2.6 Conclusion

In this session, we learned:

- How to create and activate a conda environment.
- How to run Python scripts and notebooks in VS Code.

- How to install necessary packages.

In the next chapter, we will start exploring Python basics.

3 Creating Python Environments for Your Projects

In this chapter, we will explore **three different ways** to create Python virtual environments for your projects. Creating a separate environment for each project is important because:

- Different projects often require different versions of dependencies.
- It helps avoid package version conflicts.
- It keeps your global Python installation clean.

3.1 Method 1: Using `python -m venv`

This method requires only Python to be installed (no Anaconda).

Note

If you want a specific Python version, install that version first from python.org.

Steps

1. Open a terminal or command prompt.
2. Run:

```
python -m venv myenv
```

Listing 3.1: Creating a virtual environment using venv

3. Activate the environment:

   - On Windows:

   ```
   myenv\Scripts\activate
   ```

   - On macOS/Linux:

   ```
   source myenv/bin/activate
   ```

4. Install packages:

   ```
   pip install pandas numpy
   ```

5. Deactivate when done:

   ```
   deactivate
   ```

3.2 Method 2: Using `virtualenv`

This method works across Windows, macOS, and Linux.

Steps

1. Install `virtualenv`:

   ```
   pip install virtualenv
   ```

2. Create the environment:

   ```
   virtualenv -p python3.10 venv_name
   ```

3. Activate it (similar to Method 1):

   - Windows:

     ```
     venv_name\Scripts\activate
     ```

   - macOS/Linux:

     ```
     source venv_name/bin/activate
     ```

3.3 Method 3: Using `conda create` (Recommended)

If you have **Anaconda** or **Miniconda** installed, this is often the most efficient method.

Steps

1. Create the environment with a specific Python version:

   ```
   conda create -p venv_name python==3.10 -y
   ```

2. Activate:

```
conda activate venv_name
```

3. Install packages:

```
conda install pandas numpy
```

Note

Conda environments allow you to easily view installed packages and manage dependencies.

3.4 Conclusion

We covered:

- `python -m venv` for a minimal setup.
- `virtualenv` for cross-platform flexibility.
- `conda create` for robust environment management.

Choose the method that best fits your workflow.

4 Python Syntax and Semantics

Welcome to the next session of our Python series. In this chapter, we will cover:

- Single-line and multi-line comments
- Definitions of **syntax** and **semantics**
- Basic syntax rules in Python
- Variable assignment, type inference, and dynamic typing
- Indentation rules and common syntax errors
- Practical code examples

4.1 Comments in Python

Python supports both single-line and multi-line comments.

4.1.1 Single-line Comments

Single-line comments begin with a hash symbol `#`:

```python
# This is a single-line comment
print("Hello World") # Inline comment
```

Listing 4.1: Single-line comment

4.1.2 Multi-line Comments

Multi-line comments use triple quotes (`""" ... """`) in Python scripts:

```python
"""
Welcome to the Python course.
This is a multi-line comment.
"""
```

Listing 4.2: Multi-line comment

Note: Multi-line comments may not work in Jupyter notebooks; single-line comments are always safe.

4.2 Syntax vs Semantics

4.2.1 Syntax

Syntax refers to the rules that define the structure of valid Python programs. It is about the correct arrangement of symbols and keywords.

4.2.2 Semantics

Semantics is the meaning or interpretation of code: what the program does when executed. Syntax is *how* you write the code, semantics is *what* it does.

4.3 Basic Syntax Rules

4.3.1 Case Sensitivity

Python is case-sensitive. Variables with different capitalization are treated as distinct:

```python
name = "Chris"
Name = "Nick"

print(name) # Outputs: Chris
print(Name) # Outputs: Nick
```

Listing 4.3: Case sensitivity example

4.3.2 Indentation

Python uses indentation instead of braces to define code blocks. Consistent use of spaces (commonly four) or tabs is required:

```python
age = 32
if age > 30:
    print("Age is greater than 30") # Indented block
print("This prints regardless of the if condition") # Outside block
```

Listing 4.4: Indentation example

4.3.3 Line Continuation

Use a backslash (\) to continue a long statement on the next line:

```python
total = 1 + 2 + 3 + \
       4 + 5 + 6
print(total) # Outputs: 21
```

Listing 4.5: Line continuation example

4.3.4 Multiple Statements in a Single Line

Separate statements with a semicolon (;):

```python
x = 5; y = 10; z = x + y
print(z) # Outputs: 15
```

Listing 4.6: Multiple statements in one line

4.4 Variables and Type Inference

Python infers the type of a variable dynamically at runtime:

```python
age = 32        # integer
name = "Chris"  # string

print(type(age)) # <class 'int'>
print(type(name)) # <class 'str'>

variable = 10
print(type(variable)) # int
variable = "Chris"
print(type(variable)) # str
```

Listing 4.7: Type inference example

4.5 Common Syntax Errors

4.5.1 Name Errors

Using an undefined variable:

```python
a = B # NameError: name 'B' is not defined
```

Listing 4.8: Name error example

4.5.2 Indentation Errors

Incorrect indentation:

```python
age = 32
if age > 30:
print(age) # IndentationError
```

Listing 4.9: Indentation error example

4.6 Practical Example: Nested Indentation

```python
if True:
    print("Inside True block")
    if False:
        print("Inside nested False block")
print("Outside all blocks")
```

Listing 4.10: Nested indentation example

This prints twice: once for the True block and once for the statement outside all blocks.

4.7 Summary

In this chapter, we covered:

- Single-line and multi-line comments
- Difference between syntax and semantics
- Case sensitivity, indentation, line continuation, and multiple statements
- Variables, type inference, and dynamic typing
- Common syntax and indentation errors

In the next chapter, we will explore variables, data types, and operators in Python.

5 Introduction to Variables in Python

5.1 Overview

Variables are fundamental elements in programming used to store data that can be referenced and manipulated in a program. In Python, variables are created when you assign a value to them; explicit declaration is not required.

5.2 What is a Variable?

Variables are fundamental elements in programming used to store data that can be referenced and manipulated in a program. In Python, variables are created when you assign a value to them. There is no need for explicit declaration or memory reservation; Python handles this automatically.

```python
a = 100
```

This statement creates a variable `a` and assigns it the value `100`.

5.3 Declaring and Assigning Variables

Variables are declared and assigned using the assignment operator `=`. You can assign values of any type:

```python
age = 32
height = 6.1
name = "John"
is_student = True
```

You can print variables using the `print()` function:

```python
print("Age:", age)
print("Height:", height)
print("Name:", name)
print("Is Student:", is_student)
```

5.4 Naming Conventions

Proper naming conventions help make code readable and maintainable. Important rules:

- Variable names should be descriptive.
- Must start with a letter or underscore (_).
- Can contain letters, numbers, and underscores.
- Cannot start with a number.
- Variable names are case sensitive.

Valid variable names:

```python
first_name = "John"
last_name = "Nick"
score1 = 100
_is_valid = True
```

Invalid variable names:

```python
1age = 30         # Invalid: starts with a number
first-name = "John" # Invalid: dash not allowed
@name = "John"    # Invalid: special character not allowed
```

Case Sensitivity:

```python
name = "Chris"
Name = "Naic"
print(name == Name) # False, different variables
```

5.5 Variable Types and Dynamic Typing

Python is dynamically typed, meaning the type of a variable is determined at runtime. Common types include `int`, `float`, `str`, and `bool`.

```python
age = 25        # int
height = 6.1    # float
name = "John"  # str
is_student = True # bool
```

You can check the type using `type()`:

```python
print(type(age))    # <class 'int'>
print(type(height)) # <class 'float'>
print(type(name))   # <class 'str'>
print(type(is_student)) # <class 'bool'>
```

5.6 Type Checking and Conversion

Type checking helps you understand what type of data a variable holds. Type conversion allows you to change the type of a variable.

Convert int to str:

```python
age = 25
age_str = str(age)
print(age_str, type(age_str)) # '25' <class 'str'>
```

Convert str to int:

```python
age = "25"
age_int = int(age)
print(age_int, type(age_int)) # 25 <class 'int'>
```

Convert float to int and vice versa:

```python
height = 5.11
height_int = int(height)  # 5
height_float = float(height_int) # 5.0
print(height_int, type(height_int))
print(height_float, type(height_float))
```

Invalid conversions:

```python
name = "John"
int(name) # Raises ValueError
```

5.7 Dynamic Typing in Python

Python allows the type of a variable to change as the program executes:

```python
var = 10
print(var, type(var))  # 10 <class 'int'>
var = "Hello"
print(var, type(var))  # Hello <class 'str'>
var = 3.14
print(var, type(var))  # 3.14 <class 'float'>
```

This flexibility is called dynamic typing.

5.8 User Input and Typecasting

Use the `input()` function to get user input. Input is always received as a string, so type conversion may be needed.

```python
age = input("Enter your age: ")
print(age, type(age)) # age is str

age_int = int(age)
print(age_int, type(age_int)) # age_int is int
```

5.9 Practical Example: Simple Calculator

A simple calculator using variables and input:

```python
num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))

print("Sum:", num1 + num2)
print("Difference:", num1 - num2)
print("Product:", num1 * num2)
print("Quotient:", num1 / num2)
```

5.10 Common Errors

- Using invalid variable names (e.g., starting with a number or using special characters).
- Type conversion errors (e.g., converting a non-numeric string to int).
- Forgetting that input() returns a string and not converting to int or float.
- Case sensitivity mistakes (e.g., using `name` and `Name` interchangeably).

5.11 Summary

- Variables store data and are dynamically typed in Python.
- Naming conventions help avoid errors and improve readability.
- Type checking and conversion are straightforward.
- Input from users is always a string and may need conversion.
- Python allows variable types to change during execution.
- Practical examples help reinforce concepts and highlight common errors.

6 Python Data Types

6.1 Overview and Outline

In this chapter, we will cover:

- Introduction to data types
- Importance of data types in programming
- Basic data types: integer, float, string, boolean
- Common errors and type conversion
- Preview of advanced data types (lists, tuples, sets, dictionaries)
- Practical examples

6.2 Introduction to Data Types

Data types are a classification of data which tells the compiler or interpreter how the programmer intends to use the data. They determine:

- The type of operations that can be performed
- The values the data can take
- The amount of memory needed to store the data

Different data types (e.g., integer, float) require different amounts of memory. Proper use of data types ensures efficient storage, correct operations, and helps prevent errors and bugs.

6.3 Basic Data Types in Python

6.3.1 Integer

Stores whole numbers.

```python
age = 35
print(type(age)) # <class 'int'>
```

6.3.2 Float

Stores decimal numbers.

```python
height = 5.11
print(height)      # 5.11
print(type(height)) # <class 'float'>
```

6.3.3 String

Stores sequences of characters.

```python
name = "John"
print(name)        # John
print(type(name))  # <class 'str'>
```

6.3.4 Boolean

Stores True or False values.

```python
is_true = True
print(type(is_true)) # <class 'bool'>

a = 10
b = 10
print(a == b)      # True
print(type(a == b)) # <class 'bool'>
```

6.4 Common Errors and Type Conversion

6.4.1 Type Error Example

Trying to add a string and an integer:

```python
result = "hello" + 5 # TypeError: can only concatenate str (not "int") to
    str
```

Fix: Use typecasting.

```python
result = "hello" + str(5)
print(result)      # hello5
```

6.4.2 Type Conversion Examples

```python
# String to int
num_str = "25"
num_int = int(num_str)
print(num_int, type(num_int)) # 25 <class 'int'>

# Float to int
height = 5.11
height_int = int(height)
print(height_int, type(height_int)) # 5 <class 'int'>

# Int to float
score = 100
score_float = float(score)
print(score_float, type(score_float)) # 100.0 <class 'float'>
```

6.4.3 Invalid Conversion Example

```python
name = "John"
int(name) # ValueError: invalid literal for int() with base 10: 'John'
```

6.5 String Methods Preview

Strings have many built-in methods:

```python
text = "hello"
print(text.upper())   # HELLO
print(text.isalpha()) # True
print(text.center(10)) # ' hello '
```

We will explore more string methods and operations in later chapters.

6.6 Advanced Data Types (Preview)

Python also supports advanced data types:

- Lists
- Tuples
- Sets
- Dictionaries

These will be covered in detail in future chapters.

6.7 Summary

- Data types classify and manage data efficiently in Python.
- Basic types: int, float, str, bool.
- Type conversion and type errors are common; typecasting helps fix them.
- Strings have many useful methods.
- Advanced data types will be covered later.

7 Python Operators

7.1 Overview and Outline

In this chapter, we will cover:

- Introduction to operators
- Arithmetic operators: addition, subtraction, multiplication, division, floor division, modulus, exponentiation
- Comparison operators: equal to, not equal to, greater than, less than, greater than or equal to, less than or equal to
- Logical operators: and, or, not
- Practical examples and common errors
- Simple calculator project

7.2 Introduction to Operators

Operators are symbols that perform mathematical, logical, or comparison operations on values and variables. They are essential for building logic and performing calculations in Python programs.

7.3 Arithmetic Operators

```python
a = 10
b = 5

add_result = a + b       # Addition
sub_result = a - b       # Subtraction
mul_result = a * b       # Multiplication
div_result = a / b       # Division (float result)
floor_div_result = a // b # Floor Division (integer result)
mod_result = a % b       # Modulus (remainder)
exp_result = a ** b      # Exponentiation (a to the power of b)

print("Addition:", add_result)
print("Subtraction:", sub_result)
print("Multiplication:", mul_result)
print("Division:", div_result)
print("Floor Division:", floor_div_result)
print("Modulus:", mod_result)
print("Exponentiation:", exp_result)
```

Notes:

- Division (/) returns a float.
- Floor division (//) returns the largest integer less than or equal to the result.
- Modulus (%) returns the remainder.
- Exponentiation (**) raises the first operand to the power of the second.

7.4 Comparison Operators

```python
a = 10
b = 15

print(a == b)  # Equal to
print(a != b)  # Not equal to
print(a > b)   # Greater than
print(a < b)   # Less than
print(a >= b)  # Greater than or equal to
print(a <= b)  # Less than or equal to

# String comparison (case sensitive)
str1 = "John"
str2 = "John"
print(str1 == str2) # True

str3 = "john"
print(str1 == str3) # False
```

7.5 Logical Operators

```python
x = True
y = False

print(x and y) # AND: True only if both are True
print(x or y)  # OR: True if at least one is True
print(not x)   # NOT: Inverts the boolean value
```

Truth Table:

- `and`: True only if both operands are True.
- `or`: True if at least one operand is True.
- `not`: Inverts the operand.

7.6 Practical Example: Simple Calculator

```python
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))

print("Addition:", num1 + num2)
print("Subtraction:", num1 - num2)
print("Multiplication:", num1 * num2)
print("Division:", num1 / num2)
print("Floor Division:", num1 // num2)
print("Modulus:", num1 % num2)
print("Exponentiation:", num1 ** num2)
```

7.7 Summary

- Operators are used for arithmetic, comparison, and logical operations in Python.
- Arithmetic operators include: `+`, `-`, `*`, `/`, `//`, `%`, `**`.
- Comparison operators include: `==`, `!=`, `>`, `<`, `>=`, `<=`.
- Logical operators include: `and`, `or`, `not`.
- Practical examples help reinforce concepts and highlight common errors.

8 Python Control Flow: Conditional Statements

8.1 Overview and Outline

In this chapter, we will cover:

- Introduction to conditional statements
- The `if` statement
- The `else` statement
- The `elif` statement
- Nested conditional statements
- Practical examples and real-world use cases
- Common errors and best practices
- Assignments for practice

8.2 Introduction to Conditional Statements

Conditional statements allow you to control the flow of your program based on conditions. They help you execute certain blocks of code only when specific conditions are met.

8.3 The `if` Statement

The `if` statement evaluates a condition and executes the block of code within it if the condition is true.

```python
age = 20
if age >= 18:
    print("You are allowed to vote in the elections.")
```

8.4 The `else` Statement

The `else` statement executes a block of code if the condition in the `if` statement is false.

```python
age = 16
if age >= 18:
    print("You are eligible for voting.")
else:
    print("You are a minor.")
```

8.5 The `elif` Statement

The `elif` (else if) statement allows you to check multiple conditions.

```python
age = 20
if age < 13:
    print("You are a child.")
elif age < 18:
    print("You are a teenager.")
else:
    print("You are an adult.")
```

8.6 Nested Conditional Statements

You can place one or more `if`, `elif`, or `else` statements inside another conditional block.

```python
number = int(input("Enter the number: "))
if number > 0:
    print("The number is positive.")
    if number % 2 == 0:
        print("The number is even.")
    else:
        print("The number is odd.")
else:
    print("The number is zero or negative.")
```

8.7 Practical Example: Leap Year Checker

A year is a leap year if it is divisible by 4, but not by 100 unless it is also divisible by 400.

```python
year = int(input("Enter the year: "))
if year % 4 == 0:
    if year % 100 == 0:
        if year % 400 == 0:
            print(f"{year} is a leap year.")
        else:
            print(f"{year} is not a leap year.")
    else:
        print(f"{year} is a leap year.")
else:
    print(f"{year} is not a leap year.")
```

8.8 Common Errors and Best Practices

- Always use a colon (:) after `if`, `elif`, and `else`.
- Maintain proper indentation for code blocks.
- Use descriptive variable names and clear conditions.
- Test your code with different inputs to ensure all branches work.

8.9 Assignment: Simple Calculator

Write a program that takes two numbers and an operation (+, -, *, /,

```
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))
operation = input("Enter operation (+, -, *, /, %, **): ")

if operation == "+":
    result = num1 + num2
elif operation == "-":
    result = num1 - num2
elif operation == "*":
    result = num1 * num2
elif operation == "/":
    result = num1 / num2
elif operation == "%":
    result = num1 % num2
elif operation == "**":
    result = num1 ** num2
else:
    result = "Invalid operation"

print("Result:", result)
```

8.10 Assignment: Ticket Price Based on Age

Write a program to determine ticket price based on age and student status.

```python
age = int(input("Enter your age: "))
is_student = input("Are you a student? (yes/no): ").lower()

if age < 5:
    price = 0
elif age < 12:
    price = 10
elif age < 17:
    if is_student == "yes":
        price = 12
    else:
        price = 15
else:
    price = 20

print(f"Your ticket price is: ${price}")
```

8.11 Summary

- Conditional statements control the flow of your program.
- Use `if`, `elif`, and `else` for decision making.
- Nested conditionals allow complex logic.
- Practice with real-world examples to master control flow.

9 Python Loops

9.1 Overview and Outline

In this chapter, we will cover:

- Introduction to loops
- For loop and the `range()` function
- While loop
- Loop control statements: `break`, `continue`, `pass`
- Nested loops
- Practical examples
- Common errors and best practices

9.2 Introduction to Loops

Loops allow you to execute a block of code multiple times. Python supports two main types of loops: `for` loops and `while` loops.

9.3 For Loop and Range Function

The `for` loop is used to iterate over a sequence (such as a list, string, or range of numbers).

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

`range(start, stop, step)` generates a sequence of numbers:

```python
for i in range(1, 6):
    print(i)
# Output: 1 2 3 4 5

for i in range(1, 10, 2):
    print(i)
```

```python
# Output: 1 3 5 7 9

for i in range(10, 0, -1):
    print(i)
# Output: 10 9 8 7 6 5 4 3 2 1
```

9.4 For Loop with Strings

You can iterate over each character in a string:

```python
text = "John Nayak"
for char in text:
    print(char)
```

9.5 While Loop

The `while` loop continues to execute as long as the condition is true.

```python
count = 0
while count < 5:
    print(count)
    count += 1
# Output: 0 1 2 3 4
```

9.6 Loop Control Statements

9.6.1 Break

Exits the loop prematurely.

```python
for i in range(10):
    if i == 5:
        break
    print(i)
# Output: 0 1 2 3 4
```

9.6.2 Continue

Skips the current iteration and continues with the next.

```python
for i in range(10):
    if i % 2 == 0:
        continue
    print(i)
# Output: 1 3 5 7 9
```

9.6.3 Pass

Does nothing; acts as a placeholder.

```python
for i in range(5):
    if i == 3:
        pass
    print(f"Number is {i}")
```

9.7 Nested Loops

A loop inside another loop.

```python
for i in range(3):
    for j in range(2):
        print(f"i = {i}, j = {j}")
```

9.8 Practical Examples

9.8.1 Sum of First n Natural Numbers

Using a while loop:

```python
n = 10
sum = 0
count = 1
while count <= n:
    sum += count
    count += 1
print("Sum of first", n, "natural numbers:", sum)
```

Using a for loop:

```python
n = 10
sum = 0
for i in range(1, n+1):
    sum += i
print("Sum of first", n, "natural numbers:", sum)
```

9.8.2 Prime Numbers Between 1 and 100

```python
for num in range(2, 101):
    for i in range(2, num):
        if num % i == 0:
            break
    else:
        print(num)
```

9.9 Common Errors and Best Practices

- Always update loop variables to avoid infinite loops.
- Use proper indentation for nested loops.
- Use `break`, `continue`, and `pass` appropriately.
- Test your code with different inputs.

9.10 Summary

Loops are powerful tools in Python that allow you to execute code multiple times. Understanding for and while loops, along with loop control statements, helps you solve a wide range of programming tasks efficiently.

10 Python Lists

10.1 Overview and Outline

In this chapter, we will cover:

- Definition of lists
- Creating a list
- Accessing list items
- Modifying list items
- Common list methods
- Slicing lists
- Iterating over lists
- List comprehension
- Nested lists
- Common errors

10.2 Definition of Lists

Lists are ordered, mutable collections of items. They can contain items of different data types.

10.3 Creating a List

```python
# Empty list
my_list = []
print(type(my_list)) # <class 'list'>

# List of strings
names = ["John", "Jack", "Jacob"]
print(names)
```

```python

# Mixed data types
mixed_list = [1, "hello", 3.14, True]
print(mixed_list)
```

10.4 Accessing List Items

```python
fruits = ["apple", "banana", "cherry", "kiwi", "guava"]

print(fruits[0])   # apple
print(fruits[2])   # cherry
print(fruits[-1])  # guava (last element)
print(fruits[1:4]) # ['banana', 'cherry', 'kiwi']
print(fruits[1:])  # ['banana', 'cherry', 'kiwi', 'guava']
```

10.5 Modifying List Items

```python
fruits[1] = "watermelon"
print(fruits) # banana replaced by watermelon
```

10.6 List Methods

```python
fruits.append("orange")     # Add to end
fruits.insert(1, "banana")  # Insert at index 1
fruits.remove("banana")     # Remove first occurrence
last_fruit = fruits.pop()   # Remove and return last item
index_cherry = fruits.index("cherry") # Get index of 'cherry'
count_banana = fruits.count("banana") # Count occurrences
fruits.sort()               # Sort list
fruits.reverse()            # Reverse list
fruits.clear()              # Remove all items
```

10.7 Slicing Lists

```python
numbers = [1,2,3,4,5,6,7,8,9,10]
print(numbers[2:5])   # [3, 4, 5]
print(numbers[5:])    # [6, 7, 8, 9, 10]
print(numbers[::2])   # [1, 3, 5, 7, 9]
print(numbers[::-1]) # [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

10.8 Iterating Over Lists

```python
for number in numbers:
    print(number)

# Iterating with index
for index, number in enumerate(numbers):
    print(f"Index {index}: {number}")
```

10.9 List Comprehension

10.9.1 Basic Syntax

```python
# [expression for item in iterable]
squares = [x**2 for x in range(10)]
print(squares)
```

10.9.2 With Condition

```python
# [expression for item in iterable if condition]
evens = [i for i in range(10) if i % 2 == 0]
print(evens)
```

10.10 Nested Lists

```python
nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(nested_list[0])  # [1, 2, 3]
print(nested_list[1][2]) # 6
```

10.11 Common Errors

- IndexError: Accessing an index that does not exist.
- TypeError: Using wrong data type for list operations.
- Modifying a list slice with a non-list value.

10.12 Advanced List Comprehension

10.12.1 List Comprehension with Condition

```python
# Traditional way
even_numbers = []
for i in range(10):
    if i % 2 == 0:
        even_numbers.append(i)
print(even_numbers)

# List comprehension way
even_numbers = [num for num in range(10) if num % 2 == 0]
print(even_numbers)
```

10.12.2 Nested List Comprehension

You can use nested loops inside a list comprehension to create combinations or pairs.

```python
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c', 'd']

pairs = [(i, j) for i in list1 for j in list2]
print(pairs)
# Output: [(1, 'a'), (1, 'b'), ..., (3, 'd')]
```

10.12.3 List Comprehension with Function Calls

You can use function calls inside list comprehensions.

```python
words = ["hello", "world", "Python", "list", "comprehension"]
lengths = [len(word) for word in words]
print(lengths)
# Output: [5, 5, 6, 4, 13]
```

10.12.4 List Comprehension with Else Block (Assignment)

Try to create a list comprehension that uses an else block for conditional logic.

```python
# Example (for practice):
result = [x if x % 2 == 0 else -x for x in range(10)]
print(result)
# Output: [0, -1, 2, -3, 4, -5, 6, -7, 8, -9]
```

10.13 Conclusion

- Lists are ordered, mutable, and can hold mixed data types.
- Indexing and slicing allow you to access and modify items efficiently.
- Built-in list methods simplify common operations.
- List comprehensions provide a concise way to create and transform lists.
- Nested lists enable multi-dimensional data structures.

List comprehensions are a powerful and compact feature in Python, allowing you to replace verbose loops with cleaner code. Understanding and practicing list comprehensions will help you write more efficient and readable Python programs.

Practice with more examples to master lists and list comprehensions!

11 Python Tuples

11.1 Overview and Outline

In this chapter, we will cover:

- Definition of tuples
- Creating tuples
- Accessing tuple elements
- Tuple operations
- Immutable nature of tuples
- Common tuple methods
- Packing and unpacking tuples
- Nested tuples
- Practical examples
- Conclusion

11.2 Definition of Tuples

Tuples are ordered collections of items that are **immutable**. They are similar to lists, but their elements cannot be changed once assigned.

11.3 Creating Tuples

```python
# Empty tuple
empty_tuple = ()
print(empty_tuple)
print(type(empty_tuple)) # <class 'tuple'>

# Tuple from a list
numbers = tuple([1, 2, 3, 4, 5, 6])
```

```python
print(numbers)

# Mixed data types
mixed_tuple = (1, "hello", 3.14, True)
print(mixed_tuple)

# Convert tuple to list
as_list = list(numbers)
print(as_list)
```

11.4 Accessing Tuple Elements

```python
print(numbers[0])   # First element
print(numbers[-1])  # Last element
print(numbers[0:4]) # Slicing
print(numbers[::-1]) # Reverse
```

11.5 Tuple Operations

```python
# Concatenation
concat_tuple = numbers + mixed_tuple
print(concat_tuple)

# Repetition
repeat_tuple = mixed_tuple * 3
print(repeat_tuple)
```

11.6 Immutable Nature of Tuples

Tuples are immutable; you cannot change their elements after creation.

```python
# Lists are mutable
my_list = [1, 2, 3, 4, 5]
my_list[0] = "John"
print(my_list)

# Tuples are immutable
try:
    numbers[1] = "John"
except TypeError as e:
    print(e) # 'tuple' object does not support item assignment
```

11.7 Tuple Methods

```python
# Count occurrences
print(numbers.count(1)) # 1

# Index of value
print(numbers.index(3)) # 2
```

11.8 Packing and Unpacking Tuples

```python
# Packing
packed_tuple = (1, "hello", 3.14)
print(packed_tuple)

# Unpacking
a, b, c = packed_tuple
print(a, b, c)

# Unpacking with star
numbers = (1, 2, 3, 4, 5, 6)
first, *middle, last = numbers
print(first)  # 1
print(middle) # [2, 3, 4, 5]
print(last)   # 6
```

11.9 Nested Tuples

```python
nested_tuple = ((1, 2, 3), ("a", "b", "c"), (True, False))
print(nested_tuple[0])    # (1, 2, 3)
print(nested_tuple[1][2]) # 'c'

# Iterating over nested tuples
for sub_tuple in nested_tuple:
    for item in sub_tuple:
        print(item, end=" ")
    print()
```

11.10 Conclusion

Tuples are versatile and useful in many real-world scenarios where an immutable and ordered collection is required. They are commonly used in data structures, function arguments, return values, and as dictionary keys. Understanding tuples can improve the efficiency and readability of your Python code.

12 Python Dictionaries

12.1 Overview and Outline

In this chapter, we will cover:

- Introduction to dictionaries
- Creating dictionaries
- Accessing dictionary elements
- Modifying dictionary elements
- Common dictionary methods
- Iterating over dictionaries
- Nested dictionaries
- Dictionary comprehension
- Common errors

12.2 Introduction to Dictionaries

Dictionaries are unordered collections of items stored as key-value pairs. Keys must be unique and immutable; values can be of any type.

12.3 Creating Dictionaries

```python
# Empty dictionary
empty_dict = {}
print(type(empty_dict)) # <class 'dict'>

# Another way
empty_dict2 = dict()
print(empty_dict2)

# Dictionary with key-value pairs
```

```python
student = {
    "name": "John",
    "age": 32,
    "grade": "A"
}
print(student)
```

12.4 Accessing Dictionary Elements

```python
# Access by key
print(student["grade"]) # A
print(student["age"]) # 32

# Using get method
print(student.get("grade"))    # A
print(student.get("last_name")) # None
print(student.get("last_name", "Not available")) # Not available
```

12.5 Modifying Dictionary Elements

```python
# Update value
student["age"] = 33

# Add new key-value pair
student["address"] = "India"

# Delete a key-value pair
del student["grade"]

print(student)
```

12.6 Common Dictionary Methods

```python
# Get all keys
print(student.keys())

# Get all values
print(student.values())

# Get all key-value pairs
print(student.items())

```

```python
# Shallow copy
student_copy = student.copy()
student["name"] = "John2"
print(student_copy["name"]) # John
print(student["name"])   # John2
```

12.7 Iterating Over Dictionaries

```python
# Iterate over keys
for key in student.keys():
    print(key)

# Iterate over values
for value in student.values():
    print(value)

# Iterate over key-value pairs
for key, value in student.items():
    print(f"{key}: {value}")
```

12.8 Nested Dictionaries

```python
students = {
    "student1": {"name": "John", "age": 32},
    "student2": {"name": "Peter", "age": 35}
}
print(students["student2"]["name"]) # Peter

# Iterating over nested dictionaries
for student_id, student_info in students.items():
    print(f"{student_id}: {student_info}")
    for key, value in student_info.items():
        print(f"{key}: {value}")
```

12.9 Dictionary Comprehension

```python
# Create a dictionary of squares
squares = {x: x**2 for x in range(5)}
print(squares)
# Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

12.10 Common Errors

- KeyError: Accessing a key that does not exist.
- Duplicate keys: Only the last value for a key is kept.
- Shallow copy: Use `copy()` to avoid reference issues.

12.11 Conditional Dictionary Comprehension

You can add conditions to dictionary comprehensions.

```python
# Squares of only even numbers
even_squares = {x: x**2 for x in range(10) if x % 2 == 0}
print(even_squares)
# Output: {0: 0, 2: 4, 4: 16, 6: 36, 8: 64}
```

12.12 Practical Examples

12.12.1 Counting Frequency of Elements in a List

```python
numbers = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
frequency = {}
for number in numbers:
    if number in frequency:
        frequency[number] += 1
    else:
        frequency[number] = 1
print(frequency)
# Output: {1: 1, 2: 2, 3: 3, 4: 4}
```

12.12.2 Merging Two Dictionaries

```python
dict1 = {"a": 1, "b": 2}
dict2 = {"b": 3, "c": 4}
merged_dict = {**dict1, **dict2}
print(merged_dict)
# Output: {'a': 1, 'b': 3, 'c': 4}
```

12.13 Summary

Dictionaries are powerful data structures for storing and manipulating key-value pairs in Python. They are mutable, support many useful methods, and can be nested or created efficiently using dictionary comprehensions.

Dictionary comprehensions allow you to build dictionaries with conditions and concise syntax. Dictionaries are commonly used for counting, merging, and organizing data. **Practice with more examples to master dictionary operations in Python!**

13 Real World Examples Using Lists

13.1 Overview

Lists are widely used in real-world applications and are a fundamental data structure in Python. This chapter demonstrates practical examples of how lists can be used in various scenarios.

13.2 Managing a To-Do List

```python
# Create a to-do list
to_do_list = ["Buy groceries", "Clean the house", "Pay bills"]

# Add new tasks
to_do_list.append("Schedule meeting")
to_do_list.append("Go for a run")

# Remove a completed task
to_do_list.remove("Clean the house")

# Check if a task is in the list
if "Pay bills" in to_do_list:
    print("Don't forget to pay the utility bills!")

# Print remaining tasks
print("To-do list remaining:")
for task in to_do_list:
    print(f"- {task}")
```

13.3 Organizing Student Grades

```python
grades = [85, 92, 78, 90, 88]

# Add a new grade
grades.append(95)

# Calculate average grade
```

```python
average = sum(grades) / len(grades)

# Find highest and lowest grades
highest = max(grades)
lowest = min(grades)

print(f"Average grade: {average}")
print(f"Highest grade: {highest}")
print(f"Lowest grade: {lowest}")
```

13.4 Managing an Inventory

```python
inventory = ["apples", "bananas", "oranges", "grapes"]

# Add a new item
inventory.append("strawberries")

# Remove an item after delivery
inventory.remove("bananas")

# Check if an item is in stock
item = "oranges"
if item in inventory:
    print(f"{item} are in stock.")
else:
    print(f"{item} are out of stock.")

# Print current inventory
print("Inventory list:", inventory)
```

13.5 Collecting User Feedback

```python
feedback = [
    "Great service",
    "Very satisfied",
    "Could be better",
    "Excellent experience"
]

# Add a new feedback
feedback.append("Not happy with the service")

# Count positive feedback
positive_count = sum(
    1 for comment in feedback if "great" in comment.lower() or "excellent"
        in comment.lower()
)

print(f"Positive feedback count: {positive_count}")
print("User feedback comments:")
for comment in feedback:
    print(f"- {comment}")
```

13.6 Summary

Lists are a versatile and commonly used data structure in Python. They help manage tasks, organize data, and analyze information in many real-world applications. Practice with these examples to strengthen your understanding of lists in Python.

14 Python Functions

14.1 Overview and Outline

In this chapter, we will cover:

- What are functions?
- Why use functions?
- Defining and calling functions
- Function parameters and default parameters
- Variable length arguments: positional and keyword
- Return statement
- Practical examples

14.2 Introduction to Functions

A function is a block of code that performs a specific task. Functions help organize code, enable code reuse, and improve readability.

14.3 Defining and Calling Functions

```python
# Basic function syntax
def function_name(parameters):
    """Docstring: describes the function."""
    # Function body
    return value

# Example: Check even or odd
def even_or_odd(num):
    """Finds whether a number is even or odd."""
    if num % 2 == 0:
        print("The number is even")
    else:
        print("The number is odd")
```

```python

even_or_odd(24)
```

14.4 Why Use Functions?

Functions allow you to reuse code and keep your programs organized. Instead of repeating code, you can call a function whenever needed.

14.5 Function with Multiple Parameters

```python
def add(a, b):
    """Returns the sum of two numbers."""
    return a + b

result = add(2, 4)
print(result) # Output: 6
```

14.6 Default Parameters

```python
def greet(name="Guest"):
    print(f"Hello {name}, welcome to the Paradise!")

greet("John") # Output: Hello John, welcome to the Paradise!
greet()       # Output: Hello Guest, welcome to the Paradise!
```

14.7 Variable Length Arguments

14.7.1 Positional Arguments (*args)

```python
def print_numbers(*args):
    for number in args:
        print(number)

print_numbers(1, 2, 3, 4, 5, "John")
```

14.7.2 Keyword Arguments (**kwargs)

```python
def print_details(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_details(name="John", age=32, country="India")
```

14.7.3 Combining Positional and Keyword Arguments

```python
def print_all(*args, **kwargs):
    print("Positional arguments:")
    for val in args:
        print(val)
    print("Keyword arguments:")
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_all(1, 2, 3, name="John", age=32)
```

14.8 Return Statement

Functions can return values, including multiple values.

```python
def multiply(a, b):
    return a * b

result = multiply(2, 3)
print(result) # Output: 6

# Returning multiple values
def multi_return(a, b):
    return a * b, a, b

x, y, z = multi_return(2, 3)
print(x, y, z) # Output: 6 2 3
```

14.9 Summary

Functions are essential for organizing, reusing, and maintaining code in Python. They support parameters, default values, variable length arguments, and can return single or multiple values. Practice writing and using functions to master this important concept!

14.10 Information Alert

Schedule learning time: Learning a little each day adds up. Research shows that students who make learning a habit are more likely to reach their goals. Set time aside to learn and get reminders using your learning scheduler.

15 Functions in Python: Practical Examples

In this chapter, we will explore several practical examples of Python functions. These examples will reinforce key concepts such as parameters, conditionals, recursion, string manipulation, dictionaries, and file operations.

15.1 Temperature Conversion

We begin with a simple but useful example: converting temperatures between Celsius and Fahrenheit.

```python
def convert_temperature(temperature, unit):
    """
    Converts temperature between Celsius and Fahrenheit.
    unit = 'C' means convert Celsius to Fahrenheit
    unit = 'F' means convert Fahrenheit to Celsius
    """
    if unit == 'C':
        return (temperature * 9/5) + 32
    elif unit == 'F':
        return (temperature - 32) * 5/9
    else:
        return None

# Example usage
print(convert_temperature(25, 'C')) # 77.0
print(convert_temperature(77, 'F')) # 25.0
```

Listing 15.1: Temperature conversion function

This demonstrates how conditionals help handle different cases based on the unit provided.

15.2 Password Strength Checker

Next, we create a function to check whether a password is strong. A strong password should:

- Be at least 8 characters long

- Contain at least one digit
- Contain at least one uppercase letter
- Contain at least one lowercase letter
- Contain at least one special character

```python
def is_strong_password(password):
    """
    Checks if the given password is strong.
    """
    if len(password) < 8:
        return False
    if not any(char.isdigit() for char in password):
        return False
    if not any(char.islower() for char in password):
        return False
    if not any(char.isupper() for char in password):
        return False
    special_chars = "!@#$%^&*()-_=+[{]}|;:'\",<.>/?`~"
    if not any(char in special_chars for char in password):
        return False
    return True

# Example usage
print(is_strong_password("WeakPwd")) # False
print(is_strong_password("Str0ngPwd!")) # True
```

Listing 15.2: Password strength checker

15.3 Shopping Cart: Total Cost

A practical use case of functions and dictionaries is computing the total cost of items in a shopping cart.

```python
def calculate_total_cost(cart):
    total_cost = 0
    for item in cart:
        total_cost += item["price"] * item["quantity"]
    return total_cost

# Example cart
cart = [
    {"name": "Apple", "price": 0.5, "quantity": 4},
    {"name": "Banana", "price": 0.3, "quantity": 6},
    {"name": "Orange", "price": 0.7, "quantity": 3}
]

```

```python
print(calculate_total_cost(cart)) # 5.9
```

Listing 15.3: Calculating total cost of shopping cart

15.4 Palindrome Checker

A palindrome is a string that reads the same forwards and backwards.

```python
def is_palindrome(s):
    s = s.lower().replace(" ", "")
    return s == s[::-1]

# Example usage
print(is_palindrome("aba"))                # True
print(is_palindrome("A man a plan a canal Panama")) # True
print(is_palindrome("hello"))              # False
```

Listing 15.4: Palindrome check

15.5 Factorial Using Recursion

Recursion is when a function calls itself. A classic example is computing factorials.

```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

# Example usage
print(factorial(5)) # 120
```

Listing 15.5: Factorial with recursion

15.6 Word Frequency in a File

We can read a text file and count the frequency of each word using dictionaries.

```python
def count_word_frequency(file_path):
    word_count = {}
    with open(file_path, "r") as file:
        for line in file:
            words = line.split()
            for word in words:
                word = word.lower().strip(".,!?;:\"'()[]{}")
                word_count[word] = word_count.get(word, 0) + 1
    return word_count

# Example usage
print(count_word_frequency("sample.txt"))
```

Listing 15.6: Word frequency counter

15.7 Email Validation with Regular Expressions

Finally, we can validate email addresses using regular expressions.

```python
import re

def is_valid_email(email):
    pattern = r'^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$'
    return re.match(pattern, email) is not None

# Example usage
print(is_valid_email("user@gmail.com")) # True
print(is_valid_email("invalid-email")) # False
```

Listing 15.7: Email validation

—

These examples demonstrate how Python functions can be applied in real-world scenarios such as data validation, text analysis, and simple mathematical computations.

16 Lambda Functions in Python

16.1 Introduction

In this chapter, we will discuss **lambda functions** in Python. Lambda functions are small, anonymous functions defined using the `lambda` keyword.

- An **anonymous function** is simply a function without a name.
- Lambda functions can take **any number of arguments**, but they contain only **one expression**.
- They are often used for short operations or as arguments to higher-order functions such as `map()` and `filter()`.

16.2 Syntax

The general syntax of a lambda function is:

```
lambda arguments : expression
```

16.3 Basic Example

Consider a simple function that adds two numbers:

```python
def addition(a, b):
    return a + b

print(addition(2, 3)) # 5
```

Listing 16.1: Standard function for addition

This function contains only a single expression. Instead of writing a full function definition, we can use a lambda function:

```python
addition = lambda a, b: a + b
print(addition(5, 6)) # 11
```

Listing 16.2: Addition using lambda

Here, `addition` is a variable holding a lambda function.

16.4 Checking Even Numbers

Let us now write a function to check whether a number is even:

```python
def is_even(num):
    return num % 2 == 0

print(is_even(24)) # True
```

Listing 16.3: Even check using a normal function

Using a lambda function:

```python
is_even = lambda num: num % 2 == 0
print(is_even(12)) # True
```

Listing 16.4: Even check using lambda

16.5 Lambda with Multiple Parameters

Lambda functions can also accept multiple parameters.

```python
addition = lambda x, y, z: x + y + z
print(addition(12, 13, 14)) # 39
```

Listing 16.5: Addition with three parameters

16.6 Using Lambda with Map

Lambda functions are commonly used with higher-order functions like `map()`.

16.6.1 Without Map

To find the square of a number:

```python
def square(num):
    return num ** 2

print(square(2)) # 4
```

Listing 16.6: Square function

If we want to apply this to a list of numbers, we would normally write a loop.

16.6.2 With Map and Lambda

Instead, we can use `map()` with `lambda`:

```python
numbers = [1, 2, 3, 4, 5, 6]
squared_numbers = list(map(lambda x: x**2, numbers))
print(squared_numbers) # [1, 4, 9, 16, 25, 36]
```

Listing 16.7: Using map with lambda

Here:

- The first argument of `map()` is a function (in this case, a lambda).
- The second argument is an iterable (the list `numbers`).
- The result is a `map` object, which we convert to a list.

16.7 Conclusion

Lambda functions allow us to write small, concise functions in a single line. They are particularly powerful when combined with functions like `map()` and `filter()`.

In the next chapter, we will discuss `map()` and `filter()` functions in detail.

17 The map() Function in Python

17.1 Introduction

In this chapter, we continue our discussion on Python by learning about the `map()` function.

The `map()` function:

- Applies a given function to all the items in an input iterable (such as a list or tuple).
- Returns a `map` object, which is an iterator.
- Is particularly useful for transforming data without writing explicit loops.

17.2 Basic Example

Let us first define a simple function that computes the square of a number:

```python
def square(n):
    return n * n

print(square(4)) # 16
print(square(10)) # 100
```

Listing 17.1: Square function

If we want to apply this function to a list of numbers, we can use `map()` instead of writing a loop.

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8]
result = map(square, numbers)
print(list(result))
# Output: [1, 4, 9, 16, 25, 36, 49, 64]
```

Listing 17.2: Using map with a defined function

Here:

- The first argument to `map()` is the function name (`square`).
- The second argument is the iterable (`numbers`).
- The result is a `map` object, which we convert into a list.

17.3 Using Lambda Functions with Map

Instead of defining a separate function, we can use **lambda functions** with `map()`.

```
numbers = [1, 2, 3, 4, 5, 6]
result = list(map(lambda x: x * x, numbers))
print(result)
# Output: [1, 4, 9, 16, 25, 36]
```

Listing 17.3: Using map with a lambda function

Lambda functions make the code shorter and more readable.

17.4 Mapping Multiple Iterables

The `map()` function can also take multiple iterables. In such cases, the function should accept the same number of arguments as the iterables.

```
numbers1 = [1, 2, 3]
numbers2 = [4, 5, 6]

added_numbers = list(map(lambda x, y: x + y, numbers1, numbers2))
print(added_numbers)
# Output: [5, 7, 9]
```

Listing 17.4: Adding two lists using map

17.5 Using Built-in Functions with Map

We can also apply built-in functions directly with `map()`.

17.5.1 Type Conversion

```
string_numbers = ["1", "2", "3", "4"]
int_numbers = list(map(int, string_numbers))
print(int_numbers)
# Output: [1, 2, 3, 4]
```

Listing 17.5: Converting strings to integers

17.5.2 String Manipulation

```
words = ["apple", "banana", "cherry"]
upper_words = list(map(str.upper, words))
print(upper_words)
```

```python
# Output: ['APPLE', 'BANANA', 'CHERRY']
```

Listing 17.6: Converting words to uppercase

17.6 Map with a List of Dictionaries

We can even use `map()` on complex structures like a list of dictionaries.

```python
def get_name(person):
    return person["name"]

people = [
    {"name": "John", "age": 32},
    {"name": "Jack", "age": 33}
]

names = list(map(get_name, people))
print(names)
# Output: ['John', 'Jack']
```

Listing 17.7: Extracting names from a list of dictionaries

17.7 Conclusion

The `map()` function is a powerful tool for applying transformations to iterable data structures.

- It can be used with regular functions, lambda functions, and even multiple iterables.
- Built-in functions can also be applied directly using `map()`.
- It improves code readability and efficiency by eliminating explicit loops.

By mastering `map()`, you can write concise, efficient, and more Pythonic code.

18 The `filter()` Function in Python

18.1 Introduction

The `filter()` function in Python constructs an iterator from elements of an iterable for which a given function returns `True`. It is primarily used to **filter out items from a list (or any iterable) based on a condition**.

Formally:

```
filter(function, iterable)
```

- **function**: A function that returns `True` or `False`. - **iterable**: The sequence (list, tuple, dictionary, etc.) to filter.

The result is a `filter` object (an iterator), which can be converted to a list, tuple, or other data structures.

18.2 Basic Example: Filtering Even Numbers

Let us define a function that checks whether a number is even:

```python
def is_even(num):
    return num % 2 == 0

print(is_even(24)) # True
print(is_even(13)) # False
```

Listing 18.1: Checking even numbers

Now, we can apply this function to filter only even numbers from a list:

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

result = filter(is_even, numbers)
print(list(result))
# Output: [2, 4, 6, 8, 10, 12]
```

Listing 18.2: Filtering even numbers from a list

Here:

- `is_even` is the filtering function.

- Only those elements for which the function returns `True` are included in the result.

18.3 Using Lambda Functions with Filter

We can avoid writing a separate function by using a **lambda function**.

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

result = filter(lambda x: x > 5, numbers)
print(list(result))
# Output: [6, 7, 8, 9]
```

Listing 18.3: Filtering numbers greater than 5 using lambda

This provides a concise way to apply filtering logic inline.

18.4 Filter with Multiple Conditions

We can also filter elements using multiple conditions combined with logical operators.

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

result = filter(lambda x: x > 5 and x % 2 == 0, numbers)
print(list(result))
# Output: [6, 8]
```

Listing 18.4: Filtering even numbers greater than 5

In this example:

- The number must be greater than 5.
- The number must also be even.

18.5 Filter with Dictionaries

The `filter()` function can also be applied to a list of dictionaries.

```python
people = [
    {"name": "John", "age": 32},
    {"name": "Jack", "age": 33},
    {"name": "John", "age": 25}
]

def age_above_25(person):
    return person["age"] > 25

result = filter(age_above_25, people)
print(list(result))
```

```python
# Output: [{'name': 'John', 'age': 32}, {'name': 'Jack', 'age': 33}]
```

Listing 18.5: Filtering people with age > 25

Here, only people whose age is greater than 25 are returned.

18.6 Conclusion

The `filter()` function is a powerful tool for data selection:

- It constructs iterators that return only elements satisfying a condition.
- It can be used with both named functions and lambda expressions.
- It supports filtering on complex data structures such as lists of dictionaries.

By mastering `filter()`, you can write concise, efficient, and Pythonic code for data processing and manipulation.

19 Modules and Packages in Python

19.1 Introduction

Python is an open-source programming language with a rich ecosystem of modules and packages. Modules and packages allow you to:

- Reuse code efficiently
- Organize code logically
- Access built-in or third-party functionality without rewriting it from scratch

19.2 Importing Built-in Modules

Python provides many built-in modules such as `math`. To use a module, you can import it:

```python
import math

print(math.sqrt(16)) # Output: 4.0
print(math.pi)     # Output: 3.141592653589793
```

Listing 19.1: Basic import

You can also import specific functions from a module:

```python
from math import sqrt, pi

print(sqrt(25)) # Output: 5.0
print(pi)      # Output: 3.141592653589793
```

Or import all functions using the wildcard `*`:

```python
from math import *

print(sqrt(16)) # Output: 4.0
print(pi)      # Output: 3.141592653589793
```

19.3 Third-party Packages

Third-party packages like `NumPy` are not included by default. You can install them using `pip`:

```
pip install numpy
# or via requirements.txt:
pip install -r requirements.txt
```

After installation, you can import the package and optionally use an alias:

```python
import numpy as np

arr = np.array([1, 2, 3, 4])
print(arr) # Output: [1 2 3 4]
```

19.4 Creating Custom Modules and Packages

19.4.1 Module

A module is a Python file containing functions or classes. Example: `maths.py`:

```python
# maths.py
def addition(a, b):
    return a + b

def subtraction(a, b):
    return a - b
```

19.4.2 Package

A package is a folder containing one or more modules and a special file `__init__.py`. Example folder structure:

```
package/
    __init__.py
    maths.py
```

Import a function from a custom package:

```python
from package.maths import addition, subtraction

print(addition(2, 3))   # Output: 5
print(subtraction(4, 3)) # Output: 1
```

Alternatively:

```python
from package import maths

print(maths.addition(2, 3))  # Output: 5
print(maths.subtraction(4, 3)) # Output: 1
```

19.4.3 Sub-packages

Packages can contain sub-packages. Example folder structure:

```
package/
    __init__.py
    maths.py
    subpackage/
        __init__.py
        mult.py
```

`mult.py`:

```python
def multiply(a, b):
    return a * b
```

Import from a sub-package:

```python
from package.subpackage.mult import multiply

print(multiply(4, 5)) # Output: 20
```

19.5 Key Takeaways

- Modules and packages help organize and reuse code efficiently.
- Built-in modules (`math`) provide ready-made functionality.
- Third-party packages (`NumPy`) can be installed via `pip`.
- Custom packages require a folder with an `__init__.py` file.
- Sub-packages allow hierarchical organization of complex codebases.

20 Python Standard Library Overview

Python provides a vast standard library that offers modules and packages to handle common programming tasks. These libraries are open source and can be used to simplify development and reduce code complexity. In this chapter, we discuss some commonly used modules and their usage.

20.1 Array Module

The `array` module allows the creation of arrays with items restricted by type code.

```python
import array

arr = array.array('i', [1, 2, 3, 4]) # 'i' indicates integer type code
print(arr)
```

20.2 Math Module

The `math` module provides functions and constants for mathematical operations.

```python
import math

print(math.sqrt(16))
print(math.pi)
```

20.3 Random Module

The `random` module is used to generate random numbers and select random items from a collection.

```python
import random

# Generate a random integer between 1 and 10
print(random.randint(1, 10))

# Select a random item from a list
choices = ['apple', 'banana', 'cherry']
```

```python
print(random.choice(choices))
```

20.4 OS Module

The `os` module provides functions to interact with the operating system, including file and directory operations.

```python
import os

# Get current working directory
print(os.getcwd())

# Create a new directory
os.mkdir('test_dir')
```

20.5 Shutil Module

The `shutil` module performs high-level file operations such as copying files.

```python
import shutil

# Copy source.txt to destination.txt
shutil.copy('source.txt', 'destination.txt')
```

20.6 JSON Module

The `json` module enables serialization (converting a dictionary to JSON) and deserialization (converting JSON to a dictionary).

```python
import json

data = {'name': 'John', 'age': 25}

# Serialize dictionary to JSON string
json_str = json.dumps(data)
print(json_str, type(json_str))

# Deserialize JSON string back to dictionary
parsed_data = json.loads(json_str)
print(parsed_data, type(parsed_data))
```

20.7 CSV Module

The `csv` module allows reading from and writing to CSV files.

```python
import csv

# Writing to CSV
with open('example.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(['name', 'age'])
    writer.writerow(['John', 32])

# Reading from CSV
with open('example.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

20.8 Datetime Module

The `datetime` module provides classes to manipulate dates and times.

```python
from datetime import datetime, timedelta

now = datetime.now()
print("Current time:", now)

# Calculate time one day ago
yesterday = now - timedelta(days=1)
print("Yesterday:", yesterday)
```

20.9 Time Module

The `time` module allows controlling execution flow, such as pausing the program.

```python
import time

print(time.time())
time.sleep(2) # Pause execution for 2 seconds
print(time.time())
```

20.10 Regular Expression Module

The `re` module provides tools for pattern matching in strings.

```python
import re

pattern = r'\d+'
text = "There are 123 apples"

match = re.search(pattern, text)
print(match.group()) # Output: 123
```

20.11 Summary

The Python standard library contains numerous modules for a variety of tasks. Commonly used modules include:

- `array`
- `math`
- `random`
- `os`
- `shutil`
- `json`
- `csv`
- `datetime`
- `time`
- `re`

These modules facilitate common programming operations, reduce development effort, and enhance code readability. Exploring the standard library enables efficient implementation of complex functionalities without relying on external libraries.

21 File Operations in Python

Python provides robust support for handling both text and binary files. This chapter discusses various file operations including reading, writing, appending, and working with file cursors.

21.1 Reading a File

Python provides the `open` function to access files. By default, files are opened in read mode (`'r'`). The `with` statement ensures proper closure of files.

21.1.1 Reading the Entire File

```python
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)
```

Explanation: The `read()` method reads the entire content of the file. If the file does not exist, Python will raise a `FileNotFoundError`.

21.1.2 Reading Line by Line

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

Explanation: Iterating over the file object reads one line at a time. The `strip()` method removes newline characters for cleaner output.

21.2 Writing to a File

Files can be opened in write mode (`'w'`) to overwrite existing content, or append mode (`'a'`) to add content at the end.

21.2.1 Overwriting a File

```python
with open('example.txt', 'w') as file:
    file.write("Hello World\n")
    file.write("This is a new line\n")
```

Explanation: Opening a file in write mode (`'w'`) erases existing content. Each `write()` call adds content to the file.

21.2.2 Appending to a File

```python
with open('example.txt', 'a') as file:
    file.write("Appending this line\n")
```

Explanation: Append mode (`'a'`) adds new content at the end without deleting existing data. The newline character `'\n'` ensures proper line separation.

21.2.3 Writing Multiple Lines

```python
lines = ["First line\n", "Second line\n", "Third line\n"]
with open('example.txt', 'a') as file:
    file.writelines(lines)
```

Explanation: `writelines()` writes a list of strings to the file sequentially.

21.3 Binary File Operations

Binary files store data in bytes. Python provides modes `'wb'` and `'rb'` for writing and reading binary files.

21.3.1 Writing to a Binary File

```python
with open('example.bin', 'wb') as file:
    file.write(b"Hello World in bytes")
```

21.3.2 Reading from a Binary File

```python
with open('example.bin', 'rb') as file:
    content = file.read()
    print(content)
```

21.4 Copying File Content

It is possible to read from a source file and write to a destination file.

```python
with open('example.txt', 'r') as src, open('destination.txt', 'w') as dst:
    content = src.read()
    dst.write(content)
```

21.5 Reading, Writing, and Using File Cursor

Python provides the `w+` mode to simultaneously read and write a file. The `seek()` method resets the file cursor for reading after writing.

```python
with open('example.txt', 'w+') as file:
    file.write("Hello World\n")
    file.write("This is a new line\n")

    # Move cursor to the beginning
    file.seek(0)

    # Read content after writing
    content = file.read()
    print(content)
```

Explanation: When writing to a file in `w+` mode, the cursor moves to the end. Using `seek(0)` resets it to the beginning, allowing the content to be read.

21.6 Practical Assignments

1. Read a text file and count the number of lines, words, and characters.
2. Copy content from a source text file to a destination text file.
3. Write and read a file multiple times using `w+` mode and `seek()` to manipulate the file cursor.

21.7 Summary

Python provides powerful and flexible file handling mechanisms for both text and binary files. Key operations include:

- `open()` with modes: `r, w, a, r+, w+, rb, wb`
- Reading entire file or line by line

- Writing, appending, and writing multiple lines
- Using `seek()` to manipulate file cursor
- Handling binary files using byte operations

Mastering these operations is essential for file manipulation tasks in Python programming.

22 Working with Directories and File Paths in Python

Python's `os` module provides functionality to interact with the operating system, allowing users to create directories, list files, manipulate paths, and check file or directory existence. This chapter discusses these features with practical examples.

22.1 Creating a New Directory

A new directory can be created using the `os.mkdir()` function.

```python
import os

new_directory = 'package'
os.mkdir(new_directory)
print(f"Directory {new_directory} created")
```

Explanation: The above code creates a folder named `package` in the current working directory and confirms creation with a message.

22.2 Listing Files and Directories

To list all files and directories within a folder, use the `os.listdir()` method.

```python
items = os.listdir('.')
print(items)
```

Explanation: The dot (.) refers to the current working directory. This function returns all files and subdirectories in the specified folder.

22.3 Joining Paths

Hardcoding paths can create compatibility issues across different operating systems. Python provides `os.path.join()` to build system-independent paths.

```python
directory_name = 'folder'
file_name = 'file.txt'
```

```python

full_path = os.path.join(directory_name, file_name)
print(full_path)

absolute_path = os.path.join(os.getcwd(), directory_name, file_name)
print(absolute_path)
```

Explanation: `os.path.join()` constructs a relative or absolute path using the current working directory (`os.getcwd()`) and folder/file names.

22.4 Checking Path Existence

Before performing file operations, it is essential to verify if a file or directory exists.

```python
path = 'example1.txt'

if os.path.exists(path):
    print(f"The path {path} exists")
else:
    print(f"The path {path} does not exist")
```

Explanation: `os.path.exists()` returns `True` if the specified path exists; otherwise, it returns `False`.

22.5 Checking File or Directory Type

Python provides methods to determine whether a path is a file or a directory.

```python
path = 'example.txt'

if os.path.isfile(path):
    print(f"The path {path} is a file")
elif os.path.isdir(path):
    print(f"The path {path} is a directory")
else:
    print(f"{path} is neither a file nor a directory")
```

Explanation: Use `os.path.isfile()` and `os.path.isdir()` to differentiate between files and folders.

22.6 Relative vs. Absolute Paths

A relative path specifies a location relative to the current working directory, whereas an absolute path provides the full path from the root directory.

```python
relative_path = 'folder/example.txt'
absolute_path = os.path.abspath(relative_path)
```

```
print(absolute_path)
```

Explanation: `os.path.abspath()` converts a relative path to an absolute path, which can be used reliably in file operations across different systems.

22.7 Practical Notes

- Always check whether a file or directory exists before performing operations to avoid errors.
- Use `os.path.join()` to maintain compatibility across operating systems.
- `os` module functions like `mkdir()`, `listdir()`, `path.exists()`, `isfile()`, and `isdir()` are essential for end-to-end projects involving multiple files and directories.

22.8 Summary

The `os` module provides essential utilities to manage directories and file paths in Python. Key operations include:

- Creating directories with `os.mkdir()`
- Listing files and directories with `os.listdir()`
- Joining paths using `os.path.join()`
- Checking existence with `os.path.exists()`
- Identifying file or directory type with `os.path.isfile()` and `os.path.isdir()`
- Converting relative paths to absolute paths using `os.path.abspath()`

Mastering these operations is crucial for robust file and directory management in Python-based projects.

23 Exception Handling in Python

Exception handling in Python allows programmers to manage errors gracefully and take corrective actions without stopping the execution of a program. This chapter introduces the key concepts of exceptions and demonstrates how to handle them using `try`, `except`, `else`, and `finally` blocks.

23.1 Introduction to Exceptions

Exceptions are events that disrupt the normal flow of a program. They occur when an error is encountered during execution. Unlike syntax errors, exceptions are detected while the program is running. Common examples include:

- `ZeroDivisionError` — division by zero.
- `FileNotFoundError` — file does not exist.
- `ValueError` — invalid value for an operation.
- `TypeError` — incorrect data type.
- `NameError` — variable is not defined.

23.2 Basic `try-except` Block

The `try` block contains code that may raise an exception. If an exception occurs, it is caught by the `except` block.

```python
try:
    a = b
except NameError:
    print("The variable has not been assigned.")
```

Explanation: Here, `b` is not defined. The `NameError` exception is caught, and a user-friendly message is displayed instead of stopping the program.

23.3 Handling Exception Objects

You can capture the exception object to display more details:

```python
try:
    a = b
except NameError as e:
    print(f"Exception occurred: {e}")
```

This prints the actual exception message: `NameError: name 'b' is not defined`.

23.4 Handling Multiple Exceptions

Different exceptions can be handled using multiple `except` blocks:

```python
try:
    result = 10 / num
except ValueError:
    print("Not a valid number.")
except ZeroDivisionError:
    print("Enter a denominator greater than zero.")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
```

Explanation:

- `ValueError` handles invalid input values.
- `ZeroDivisionError` handles division by zero.
- `Exception` is the base class for all exceptions and catches any other errors.

23.5 Using the `else` and `finally` Blocks

Python's exception handling allows graceful management of runtime errors. In addition to `try` and `except` blocks, Python provides `else` and `finally` for more control.

23.5.1 The `else` Block

The `else` block executes only if no exception occurs in the `try` block. It is useful for code that should run after successful execution.

```python
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ValueError:
    print("Not a valid number.")
except ZeroDivisionError:
    print("Enter a denominator greater than zero.")
else:
    print(f"The result is {result}")
```

Explanation:

- Executes only if the user inputs a valid number.
- Skipped if any exception occurs.

23.5.2 The `finally` Block

The `finally` block executes regardless of whether an exception occurs. It is commonly used to release resources such as files or database connections.

```python
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("You can't divide by zero.")
except ValueError:
    print("Not a valid number.")
else:
    print(f"The result is {result}")
finally:
    print("Execution complete.")
```

Explanation:

- `else` runs only if no exception occurs.
- `finally` runs regardless of exceptions, making it ideal for cleanup tasks.

23.5.3 Practical Example: File Handling

```python
try:
    file = open("example_one.txt", "r")
    content = file.read()
    print(content)
except FileNotFoundError:
    print("The file does not exist.")
except Exception as e:
    print(f"An error occurred: {e}")
finally:
    if 'file' in locals() and not file.closed:
        file.close()
        print("File closed.")
```

Explanation:

- Handles missing files gracefully with `FileNotFoundError`.
- Catches unexpected errors using the base `Exception` class.
- Ensures the file is always closed with `finally`.

23.6 Key Takeaways

- Use `except` to catch specific exceptions.
- Use `else` for code that should run if no exceptions occur.
- Use `finally` for cleanup tasks that must execute regardless of exceptions.
- Combine `try`, `except`, `else`, and `finally` for robust, user-friendly code.

Conclusion: Mastering `try-except-else-finally` blocks ensures that Python programs handle errors gracefully, maintain resource integrity, and provide clear, user-friendly feedback.

24 Object-Oriented Programming in Python: Classes and Objects

24.1 Introduction to Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm that uses objects to design applications and computer programs. Python allows us to model real-world scenarios using classes and objects, which increases code reusability and efficiency. By understanding OOP concepts, we can develop real-world use cases effectively.

24.2 Classes and Objects

A **class** is a blueprint for creating objects. It contains attributes (properties) and methods (functions) that define the behavior and characteristics of the object.

24.2.1 Basic Example: Car Class

We can define a class in Python using the `class` keyword:

```python
class Car:
    pass
```

Here, `Car` is a class. We can now create objects (instances) from this class:

```python
# Creating objects
audi = Car()
bmw = Car()

print(type(audi)) # Output: <class '__main__.Car'>
print(type(bmw)) # Output: <class '__main__.Car'>
```

Each object represents a specific instance of the class. For example, `audi` and `bmw` are different cars, but they share the same blueprint provided by the `Car` class.

24.3 Attributes and Methods

24.3.1 Instance Variables

Attributes represent the properties of a class. We can define them for objects as follows:

```python
audi.windows = 4
tata = Car()
tata.doors = 4
```

However, this approach is not recommended because attributes may not be consistent across objects. Instead, we use a constructor method to initialize attributes.

24.3.2 Constructor: __init__ Method

The `__init__` method is called a constructor and is used to initialize attributes when an object is created. It uses the `self` keyword to refer to the current instance.

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

`name` and `age` are instance variables initialized during object creation:

```python
dog1 = Dog("Buddy", 3)
dog2 = Dog("Lucy", 4)

print(dog1.name, dog1.age) # Output: Buddy 3
print(dog2.name, dog2.age) # Output: Lucy 4
```

24.3.3 Instance Methods

Methods define the behavior of objects. An instance method can access instance variables using `self`:

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

   def bark(self):
      print(f"{self.name} says Woof")
```

We can now call methods on specific objects:

```python
dog1.bark() # Output: Buddy says Woof
dog2.bark() # Output: Lucy says Woof
```

Each object uses the same method but behaves according to its own attributes.

24.4 Example: Bank Account

Let's create a practical example of modeling a bank account using classes.

24.4.1 Class Definition

```python
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        print(f"{amount} is deposited. New balance: {self.balance}")

    def withdraw(self, amount):
        if amount > self.balance:
            print("Insufficient funds")
        else:
            self.balance -= amount
            print(f"{amount} is withdrawn. New balance: {self.balance}")

    def get_balance(self):
        return self.balance
```

24.4.2 Creating Objects and Using Methods

```python
# Creating a bank account for Crush
account = BankAccount("Crush", 5000)

# Depositing money
account.deposit(100) # Output: 100 is deposited. New balance: 5100

# Withdrawing money
account.withdraw(300) # Output: 300 is withdrawn. New balance: 4800

# Checking balance
print(account.get_balance()) # Output: 4800
```

24.5 Conclusion

Object-Oriented Programming in Python allows us to model real-world scenarios using classes and objects. In this chapter, you have learned how to:

- Create classes and objects

- Define and use instance variables (attributes)
- Define and use instance methods
- Apply these concepts in a real-world example of a bank account

Understanding OOP is essential for writing reusable, maintainable, and scalable Python applications.

25 Inheritance in Python: Single and Multiple Inheritance

25.1 Introduction

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class to inherit attributes and methods from another class. This promotes code reuse and models real-world scenarios, such as inheriting property from parents.

In this chapter, we will discuss how to implement **single inheritance** and **multiple inheritance** in Python, with practical examples.

25.2 Single Inheritance

25.2.1 Parent Class Example: Car

We define a parent class `Car` with attributes `windows`, `doors`, and `engine_type`, along with an instance method `drive`.

```python
class Car:
    def __init__(self, windows, doors, engine_type):
        self.windows = windows
        self.doors = doors
        self.engine_type = engine_type

    def drive(self):
        print(f"The person will drive the {self.engine_type} car")

# Create an object of Car
car1 = Car(4, 5, "petrol")
car1.drive()
```

25.2.2 Child Class Example: Tesla

The `Tesla` class inherits from `Car` and adds a new attribute `is_self_driving` along with a method to show this feature.

```python
class Tesla(Car):
    def __init__(self, windows, doors, engine_type, is_self_driving):
        super().__init__(windows, doors, engine_type)
        self.is_self_driving = is_self_driving

    def self_driving_info(self):
        print(f"Tesla supports self-driving: {self.is_self_driving}")

# Create an object of Tesla
tesla1 = Tesla(4, 5, "electric", True)
tesla1.drive() # inherited from Car
tesla1.self_driving_info() # Tesla specific method
```

25.3 Multiple Inheritance

Multiple inheritance occurs when a class inherits from more than one base class.

25.3.1 Base Classes Example: Animal and Pet

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

    def speak(self):
        print("Subclasses must implement this method")

class Pet:
    def __init__(self, owner):
        self.owner = owner
```

25.3.2 Derived Class Example: Dog

```python
class Dog(Animal, Pet):
    def __init__(self, name, owner):
        Animal.__init__(self, name)
        Pet.__init__(self, owner)

    def speak(self):
        return f"{self.name} says woof"

# Create object of Dog
dog = Dog("Buddy", "Crush")
print(dog.speak()) # Output: Buddy says woof
print(dog.owner)  # Output: Crush
```

25.4 Conclusion

- **Single inheritance** allows a child class to inherit attributes and methods from a single parent class.
- **Multiple inheritance** allows a child class to inherit from multiple parent classes.
- Use of the `super()` function or direct parent class calls helps initialize inherited attributes.
- Inheritance is useful for code reuse and modeling real-world relationships in Python.

26 Polymorphism in Python

Polymorphism is a fundamental concept in object-oriented programming that allows objects of different classes to be treated as objects of a common superclass. It provides a mechanism to perform a single action in multiple forms, enhancing code flexibility and maintainability. The term ”polymorphism” literally means ”many forms.”

26.1 Method Overriding

Method overriding enables a child class to provide a specific implementation of a method that is already defined in its parent class. This allows derived classes to modify or extend the behavior of the base class.

26.1.1 Example: Animal Sounds

```python
class Animal:
    def speak(self):
        return "Sound of the animal"

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

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

dog = Dog()
cat = Cat()

print(dog.speak()) # Output: Woof
print(cat.speak()) # Output: Meow
```

Listing 26.1: Method Overriding with Animals

In this example, the method `speak` is overridden in each child class, allowing `Dog` and `Cat` to produce distinct outputs.

26.2 Polymorphism with Functions

Polymorphism can also be demonstrated using functions that operate on objects of different types in a uniform manner. This approach allows the same function to process various object types, each implementing the required interface or method.

26.2.1 Example: Shapes and Areas

```python
class Shape:
    def area(self):
        pass

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 3.14 * self.radius * self.radius

def print_area(shape):
    print(f"The area is: {shape.area()}")

rect = Rectangle(4, 5)
circle = Circle(3)

print_area(rect) # Output: The area is: 20
print_area(circle) # Output: The area is: 28.26
```

Listing 26.2: Polymorphism with Shape Classes

The function `print_area` illustrates polymorphism by invoking the `area` method on objects of different classes.

26.3 Polymorphism with Abstract Base Classes

In Python, interfaces are implemented through *abstract base classes* (ABCs). An abstract base class defines a common interface for a group of related objects and enforces that derived classes implement the required methods. This ensures consistency across multiple implementations.

26.3.1 Example: Vehicles

```python
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def start_engine(self):
        pass

class Car(Vehicle):
    def start_engine(self):
        return "Car engine started"

class Motorcycle(Vehicle):
    def start_engine(self):
        return "Motorcycle engine started"

car = Car()
motorcycle = Motorcycle()

print(car.start_engine())    # Output: Car engine started
print(motorcycle.start_engine()) # Output: Motorcycle engine started
```

Listing 26.3: Polymorphism using Abstract Base Classes

This example demonstrates how abstract base classes enforce method implementation, while still supporting polymorphic behavior across multiple derived classes.

26.4 Conclusion

Polymorphism is an essential feature of object-oriented programming. It allows a single function or method to handle objects from different classes, each implementing the required method in its own way. By leveraging polymorphism, developers can write flexible, extensible, and maintainable code that accommodates future changes and extensions without requiring modifications to existing logic.

27 Encapsulation in Python

Encapsulation is a core concept in object-oriented programming that helps in designing maintainable and reusable code. It refers to the practice of bundling data (variables) and methods that operate on that data into a single unit (a class) and restricting direct access to some components of an object. This prevents accidental modification or misuse of the data.

27.1 Access Modifiers in Python

Python supports three main types of access modifiers:

- **Public:** Variables and methods accessible from anywhere in the program.
- **Protected:** Variables and methods prefixed with a single underscore (‘_‘) that should only be accessed within the class and its subclasses.
- **Private:** Variables and methods prefixed with double underscores (‘__‘) that are intended to be hidden from outside the class.

27.2 Public Variables

Public instance variables can be accessed directly using the object.

```python
class Person:
    def __init__(self, name, age):
        self.name = name # Public variable
        self.age = age  # Public variable

person = Person("Chris", 34)
print(person.name) # Output: Chris
print(person.age) # Output: 34
```

Listing 27.1: Public Variables Example

27.3 Private Variables

Private variables are not accessible directly outside the class. They are defined with double underscores (‘__‘) and are intended to be hidden.

```python
class Person:
    def __init__(self, name, age):
        self.__name = name # Private variable
        self.__age = age # Private variable

person = Person("Chris", 34)
# print(person.__name) # This would raise an AttributeError
```
Listing 27.2: Private Variables Example

27.4 Protected Variables

Protected variables, prefixed with a single underscore (‘_‘), can be accessed in subclasses but are conventionally not accessed directly from outside the class.

```python
class Person:
    def __init__(self, name):
        self._name = name # Protected variable

class Employee(Person):
    def display_name(self):
        return self._name

employee = Employee("Chris")
print(employee.display_name()) # Output: Chris
```
Listing 27.3: Protected Variables Example

27.5 Getter and Setter Methods

To access or modify private variables safely, Python uses getter and setter methods. These allow controlled access to private attributes and can enforce constraints.

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

    # Getter methods
    def get_name(self):
        return self.__name

    def get_age(self):
        return self.__age

    # Setter methods
    def set_name(self, name):
        self.__name = name

    def set_age(self, age):
        if age > 0:
            self.__age = age
        else:
            print("Age cannot be negative")

# Example usage
person = Person("Chris", 34)
print(person.get_name()) # Output: Chris
print(person.get_age()) # Output: 34

person.set_age(35)
print(person.get_age()) # Output: 35

person.set_age(-5)     # Output: Age cannot be negative
```

Listing 27.4: Encapsulation Using Getter and Setter

27.6 Conclusion

Encapsulation in Python is achieved using access modifiers (public, protected, private) and getter/setter methods.

- **Public variables:** Accessible from anywhere.
- **Protected variables:** Accessible within the class and subclasses.
- **Private variables:** Restricted to the class itself, accessed only through getter/setter methods.

This approach hides internal implementation details while exposing only the necessary interface, leading to safer and more maintainable code. Abstraction, which will be discussed in the next chapter, builds upon encapsulation to further simplify complex systems.

28 Abstraction in Python

Abstraction is a fundamental concept in object-oriented programming (OOP). It involves hiding complex implementation details while exposing only the essential features of an object. This reduces programming complexity and enhances maintainability.

28.1 Real-World Examples of Abstraction

- **Washing Machine:** Users interact with buttons to start, stop, or set a timer, while the internal washing process remains hidden.
- **Mobile Phones or Laptops:** Users see icons, buttons, or menus for operations like shutting down or opening applications, but the underlying implementation is concealed.
- **AC Remote:** The remote exposes essential controls, while the internal circuitry and logic remain hidden.

28.2 Abstract Classes in Python

Python provides the `abc` module to implement abstraction. An abstract class serves as a blueprint for derived classes and can contain both normal and abstract methods. Abstract methods define a required interface but contain no implementation.

```python
from abc import ABC, abstractmethod

# Abstract base class
class Vehicle(ABC):

    def drive(self):
        print("The vehicle is used for driving") # Normal method

    @abstractmethod
    def start_engine(self):
        pass # Abstract method to be implemented by subclasses
```

Listing 28.1: Abstract Class and Method Example

28.3 Implementing Abstract Methods in Child Classes

Derived classes must implement all abstract methods from the abstract base class. This ensures the necessary features are exposed, while complex implementation details are handled internally.

```python
class Car(Vehicle):

    def start_engine(self):
        print("Car engine started") # Implementation of abstract method

# Example usage
def operate_vehicle(vehicle):
    vehicle.start_engine()

car = Car()
operate_vehicle(car) # Output: Car engine started
car.drive()          # Output: The vehicle is used for driving
```

Listing 28.2: Derived Class Implementing Abstract Method

28.4 Key Points

- Abstraction hides the internal workings of an object and exposes only the required interface.
- Abstract classes are defined by inheriting from `ABC`.
- Abstract methods must be implemented by derived classes, allowing flexibility in how functionality is realized.
- Normal methods in abstract classes can still provide shared functionality accessible by all derived classes.

28.5 Conclusion

Abstraction simplifies interaction with objects by presenting only the necessary features and hiding the internal complexities. It improves code maintainability, reusability, and reduces cognitive load for developers. Python supports abstraction through abstract base classes and abstract methods.

29 Magic Methods in Python

In our previous discussions, we have covered many concepts in object-oriented programming (OOP), including function and method overriding, encapsulation, and abstraction. In this chapter, we will explore **magic methods** in Python.

29.1 Introduction to Magic Methods

Magic methods, also known as *dunder methods* (double underscore methods), are special methods that begin and end with double underscores. They allow you to define the behavior of objects for built-in operations such as arithmetic, comparison, and string representation.

Some common magic methods include:

- `__init__` – Constructor, initializes a new instance of a class.
- `__str__` – Returns a human-readable string representation of the object.
- `__repr__` – Returns an official string representation of the object, useful for debugging.

29.2 Basic Example of Magic Methods

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

Listing 29.1: Defining a Class with Magic Methods

Creating an object of this class:

```python
person = Person("John", 34)
print(person) # Default output: <__main__.Person object at 0x...>
```

By default, printing the object displays its memory address. To customize this, we override the `__str__` method.

29.3 Overriding __str__ Method

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

    def __str__(self):
        return f"{self.name}, {self.age} years old"

person = Person("John", 34)
print(person) # Output: John, 34 years old
```

29.4 Overriding __repr__ Method

The `__repr__` method provides an *official* string representation of an object, often used for debugging:

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

    def __repr__(self):
        return f"Person(name={self.name}, age={self.age})"

person = Person("John", 34)
print(repr(person)) # Output: Person(name=John, age=34)
```

29.5 Key Points

- Magic methods define how objects behave with built-in operations.
- Overriding magic methods allows you to customize object behavior.
- Common magic methods include `__init__`, `__str__`, `__repr__`, `__add__`, `__eq__`, and many more.
- You can explore additional magic methods to implement operator overloading or other custom behaviors.

29.6 Conclusion

Magic methods provide a powerful way to define object behavior for built-in operations. By overriding these methods, you can make your objects more intuitive, readable, and functional. In the next chapter, we will explore **operator overloading**, which builds upon magic methods to customize arithmetic and comparison operations.

30 Operator Overloading in Python

In our previous discussion, we explored **magic methods**. In this chapter, we will learn how to use these methods to perform **operator overloading**.

30.1 Introduction

Operator overloading allows you to define custom behavior for Python's built-in operators (such as `+`, `-`, `*`, `==`, etc.) for your own objects. This is done by overriding specific magic methods.

- `__add__` for addition (`+`)
- `__sub__` for subtraction (`-`)
- `__mul__` for multiplication (`*`)
- `__truediv__` for division (`/`)
- `__eq__` for equality comparison (`==`)
- `__lt__` for less than (`<`)
- `__gt__` for greater than (`>`)

30.2 Example: Vector Class

We can overload operators for a simple 2D vector class.

```python
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # Addition
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    # Subtraction
    def __sub__(self, other):
        return Vector(self.x - other.x, self.y - other.y)

    # Multiplication (by scalar)
    def __mul__(self, other):
        return Vector(self.x * other, self.y * other)

    # Equality check
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    # String representation
    def __repr__(self):
        return f"Vector(x={self.x}, y={self.y})"

# Create vector objects
v1 = Vector(2, 3)
v2 = Vector(4, 5)

# Operator overloading in action
print(v1 + v2) # Vector(x=6, y=8)
print(v1 - v2) # Vector(x=-2, y=-2)
print(v1 * 3) # Vector(x=6, y=9)
print(v1 == v2) # False
```

Listing 30.1: Vector Class with Operator Overloading

30.3 Explanation

- `v1 + v2` calls `v1.__add__(v2)` and returns a new vector with summed coordinates.
- `v1 - v2` calls `v1.__sub__(v2)`.
- `v1 * 3` calls `v1.__mul__(3)` for scalar multiplication.
- `v1 == v2` calls `v1.__eq__(v2)` and returns a boolean.
- `__repr__` provides a human-readable representation for debugging and printing.

30.4 Key Points

- Operator overloading is achieved by overriding magic methods.
- You can overload arithmetic, comparison, and other operators for custom classes.
- This is particularly useful for mathematical objects like vectors, matrices, and complex numbers.
- Proper implementation of these methods ensures your objects behave intuitively.

30.5 Conclusion

Magic methods and operator overloading are powerful features in Python. They allow you to define custom behaviors for standard operators, making your objects more expressive and easier to use in application development. You can apply this to vectors, matrices, complex numbers, or any custom data structures.

31 Creating Web Applications with Streamlit

In this chapter, we will learn how to create interactive web applications using **Streamlit**, an open-source framework for machine learning and data science projects. Streamlit allows you to build beautiful web apps using only Python scripts, without needing HTML or CSS knowledge.

31.1 Introduction to Streamlit

Streamlit is widely used in data science and AI projects, including creating interactive dashboards and generative AI chatbots. Its key features include:

- Easy Python integration
- Pre-built widgets (text input, sliders, select boxes, etc.)
- Charts and tables support
- Rapid prototyping for ML and data projects

31.2 Getting Started

31.2.1 Installation

Install the required packages using:

```
pip install -r requirements.txt
```

31.2.2 Basic App Setup

Create a Python file, e.g., `app.py`, and import Streamlit:

```
import streamlit as st
import pandas as pd
import numpy as np

# Title of the app
```

```python
st.title("Hello Streamlit")

# Display a simple text
st.write("This is a simple text.")

# Display a DataFrame
df = pd.DataFrame({
    "Column 1": [1, 2, 3],
    "Column 2": [4, 5, 6]
})
st.write("Here is the DataFrame:")
st.write(df)

# Create a line chart
chart_data = pd.DataFrame(
    np.random.randn(20, 3),
    columns=['A', 'B', 'C']
)
st.line_chart(chart_data)
```

Run the app with:

```
streamlit run app.py
```

The app will open in your browser at `localhost:8501`.

31.3 Interactive Widgets

Streamlit provides several widgets for user interaction. Examples:

31.3.1 Text Input

```python
name = st.text_input("Enter your name:")

if name:
    st.write(f"Hello {name}!")
```

31.3.2 Slider

```python
age = st.slider("Select your age:", 0, 100, 25)
st.write(f"Your age is: {age}")
```

31.3.3 Select Box

```python
options = ["Python", "Java", "C", "C++", "JavaScript"]
choice = st.selectbox("Choose your favorite programming language:", options
    )
st.write(f"Your choice is: {choice}")
```

31.3.4 File Uploader

```python
uploaded_file = st.file_uploader("Choose a CSV file", type="csv")

if uploaded_file is not None:
    df = pd.read_csv(uploaded_file)
    st.write(df)
```

31.4 Conclusion

Streamlit allows rapid development of interactive web apps for data science and machine learning projects. Key points:

- No need for HTML/CSS knowledge
- Easy to display data, charts, and widgets
- Ideal for POCs and interactive dashboards
- Supports machine learning project deployment

In the next chapter, we will demonstrate an end-to-end machine learning application built using Streamlit, showcasing how to make ML models interactive and accessible via a web app.

32 Building a Machine Learning Web App with Streamlit

In this chapter, we demonstrate how to create an interactive machine learning application using **Streamlit**. This example uses the classic Iris dataset and a Random Forest classifier to showcase predictions in real-time.

32.1 Introduction

Streamlit allows developers to create interactive web apps for data science and machine learning without needing HTML or CSS. By combining sliders, input fields, and charts, users can interact with models directly from the browser.

32.2 Project Setup

32.2.1 Installing Dependencies

Install the required packages using:

```
pip install -r requirements.txt
```

Ensure the environment includes `streamlit`, `pandas`, `numpy`, and `scikit-learn`.

32.2.2 Importing Libraries

```
import streamlit as st
import pandas as pd
import numpy as np
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
```

32.3 Loading and Caching the Dataset

We define a function to load the Iris dataset and convert it into a pandas DataFrame. Using the `@st.cache_data` decorator ensures that the dataset is cached and not reloaded every time the app runs.

```python
@st.cache_data
def load_data():
    iris = load_iris()
    df = pd.DataFrame(iris.data, columns=iris.feature_names)
    df['species'] = iris.target
    return df, iris.target_names

df, target_names = load_data()
```

32.4 Training the Random Forest Model

```python
model = RandomForestClassifier()
X = df.iloc[:, :-1] # Independent features
y = df['species'] # Dependent feature
model.fit(X, y)
```

32.5 Creating Interactive Sliders for Input

We use Streamlit sliders to allow users to input sepal length, sepal width, petal length, and petal width interactively:

```python
sepal_length = st.slider("Sepal Length", float(df.iloc[:,0].min()), float(
    df.iloc[:,0].max()))
sepal_width = st.slider("Sepal Width", float(df.iloc[:,1].min()), float(df.
    iloc[:,1].max()))
petal_length = st.slider("Petal Length", float(df.iloc[:,2].min()), float(
    df.iloc[:,2].max()))
petal_width = st.slider("Petal Width", float(df.iloc[:,3].min()), float(df.
    iloc[:,3].max()))
```

32.6 Making Predictions

The app collects the slider inputs into a list and predicts the Iris species using the trained model:

```python
input_data = [[sepal_length, sepal_width, petal_length, petal_width]]
prediction = model.predict(input_data)
predicted_species = target_names[prediction[0]]
st.write(f"Predicted Iris Species: {predicted_species}")
```

32.7 Running the App

Run the Streamlit app using:

```
streamlit run classification.py
```

The app opens in the browser, displaying sliders for feature input and the predicted Iris species in real-time.

32.8 Conclusion

This example demonstrates the power of Streamlit for creating interactive machine learning applications:

- Users can interactively adjust input features using sliders.
- Predictions from the machine learning model are updated instantly.
- Minimal code is required for front-end display; no HTML or CSS is necessary.
- Streamlit caching ensures efficient loading of datasets.

In the next chapter, we will explore more advanced interactive components and integrate additional visualizations into a machine learning web app.