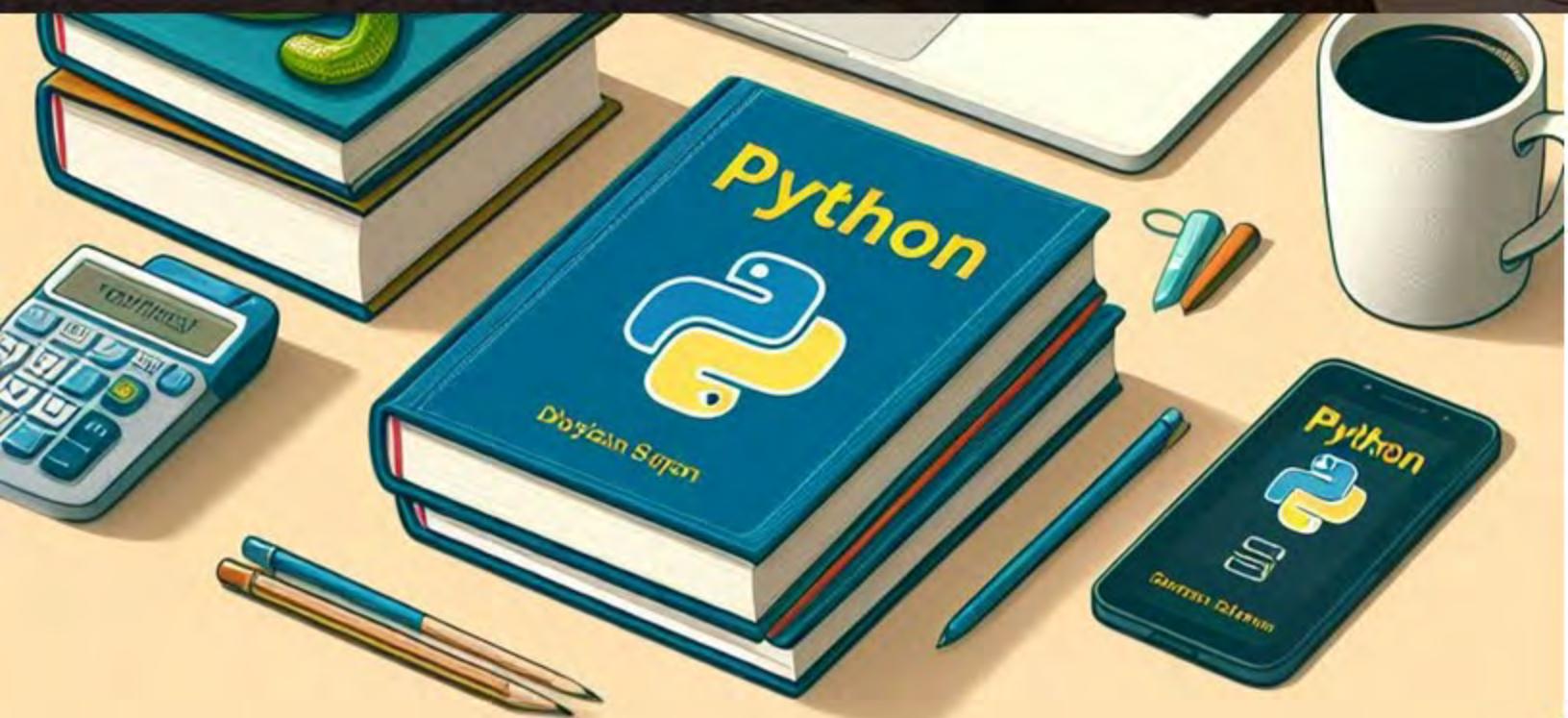


Learning Python by Example



Learning Python by Example

Learner's Guide

© 2024 Aptech Limited

All rights reserved.

No part of this book may be reproduced or copied in any form or by any means – graphic, electronic or mechanical, including photocopying, recording, taping, or storing in information retrieval system or sent or transferred without the prior written permission of copyright owner Aptech Limited.

All trademarks acknowledged.

APTECH LIMITED

Contact E-mail: ov-support@onlinevarsity.com

First Edition - 2024



Onlinevarsity



**LEARN
ANYWHERE
ANYTIME**

Preface

In the dynamic realm of programming education, mastering Python serves as a foundational skill for aspiring developers. "Python-Learning by Example" is a comprehensive guide designed to take beginners through the intricacies of Python programming with practical examples. It begins with a gentle introduction to Python, laying the foundation with a thorough exploration of data types, operators, and flow control statements. As readers progress, they delve into functions, parameters, and essential data structures like lists, tuples, dictionaries, and sets, fostering a solid understanding of Python's core concepts.

Moving beyond the basics, the book navigates through iterators, generators, decorators, and Object-Oriented Programming (OOP) principles. It further delves into advanced OOP techniques and exception handling strategies, equipping learners with the skills to write robust and error-tolerant code. Additionally, the book explores serialization, file handling, threading, networking, and database interactions, empowering readers to apply Python in a variety of real-world scenarios and harness its full potential as a versatile programming language.

This book is the result of a concentrated effort of the Design Team, which is continuously striving to bring you the best and the latest in Information Technology. The process of design has been a part of the ISO 9001 certification for Aptech-IT Division, Education Support Services. As part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends.

We will be glad to receive your suggestions.

Design Team



**MANY
COURSES
ONE
PLATFORM**



Onlinevarsity App for Android devices

Download from **Google Play Store**

Table of Contents

Sessions

1. Introduction to Python
2. Data Types and Operators in Python
3. Flow Control Statements in Python
4. Functions and Function Parameters in Python
5. List and Tuples in Python
6. Dictionaries and Sets in Python
7. Iterators, Generators, and Decorators
8. Object Oriented Programming (OOPS)
9. Advanced OOPS and Exception Handling
10. Serialization and Deserialization
11. Files and Threads
12. Networking and Database Handling

Appendix



SESSION 01

INTRODUCTION TO PYTHON

Learning Objectives

In this session, students will learn to:

- ◆ Explain the concept of Python Programming
- ◆ Explain briefly the history of Python
- ◆ List the features, usage, and benefits of Python
- ◆ Outline the process to create a Python Program
- ◆ Explain about identifiers
- ◆ Identify and list the keywords in Python

1.1 Introduction to Python Programming

Python, known for its simplicity and readability, is a highly popular high-level programming language widely used in diverse domains such as Web development, data analysis, scientific computing, and Artificial Intelligence. The online Python community actively contributes to its continuous improvement since it is an open-source language with freely available source code.

Python's design is believed to highlight the code readability through significant indentation. Python accommodates a variety of programming approaches, such as structured, object-oriented, and functional programming. It also features dynamic typing and automatic memory management. Due to its interpreted nature and high-level built-in data structures, Python is well-suited for Rapid Application Development (RAD) and serves as a scripting or glue language to connect existing components.

Python's easy-to-learn syntax enhances productivity, reducing the cost of program maintenance. Python does not require separate compilation stage, it is compiled and interpreted in the same stage.

The absence of a compilation step enables an exceptionally fast edit-test-debug cycle. When errors occur, Python raises exceptions instead of causing segmentation faults, and the interpreter (is a computer program that converts each high-level program statement into machine code). This provides a stack trace if exceptions are not caught.

Inspecting variables, evaluating expressions, setting breakpoints, and stepping through the code one by one are allowed by the language's source-level debugger. This ability showcases how the language can look into its own workings. Additionally, programmers often use print statements for quick debugging, taking advantage of Python's swift edit-test-debug cycle.

1.2 History of Python

In the late 1980s, the foundations for the history of Python were being laid, with the commencement of work on the programming language. It was during this time that its development was initiated, led by Guido Van Rossum at Centrum Wiskunde and Informatica (CWI) in the Netherlands. The application-based work on Python began in December 1989.

On February 20, 1991, the first version of Python was released. Interestingly, the name of the Python programming language was inspired by an old BBC television

comedy sketch series called Monty Python's Flying Circus. It was rather than the large snake commonly associated with the name.

The maintenance of Python is currently done by the Python Software Foundation, which is a non-profit membership organization with a dedicated community. The continuous improvement, expansion, and popularization of the Python language and its environment are the foundation's main focus.

The Python Software Foundation is responsible for the maintenance of Python. It is a non-profit membership organization and has a dedicated community. The foundation is devoted to continuous improvement, expansion, and popularization of the Python language and its environment.

1.3 Features and Benefits of Python

Python, with its simplicity, readability, and flexibility, is a versatile and widely-used programming language.

Python has attained popularity among programmers, data scientists, and engineers due to its extensive libraries, ease of use, and strong community support.

1.3.1 Features of Python

Python provides several features, some of which are as follows:

Easy to Learn and Read

Python's syntax is designed to be straightforward and easily understood, making it an excellent choice for beginners and experienced programmers alike.

Open Source

Python is an open-source language, which means its source code is freely available for anyone to use, modify, and distribute.

Cross-Platform

Python programs can be run on various operating systems, including Windows, macOS, Linux, and others, without the necessity for any modifications.

Large Standard Library

A vast standard library is provided by Python, which offers ready-to-use modules and functions for a wide range of tasks, such as string processing, file I/O, networking, and more.

Third-Party Libraries

Python's potential is greatly enhanced across multiple domains. This includes data analysis, Machine Learning, Web development, and scientific computing, by an extensive collection of third-party libraries and packages such as NumPy, Pandas, TensorFlow, Django, Flask, and others.

Dynamic Typing

Python's dynamic typing feature allows the type of a variable to be determined dynamically at runtime. This eliminates the necessity of an explicit declaration of variable types, thereby contributing its ease of use and adaptability.

Object-Oriented

Python supports Object-Oriented Programming (OOP) principles, allowing creation and working with classes and objects.

High-level Language

Python abstracts many low-level details, making the code more human-readable and reducing its complexity.

1.3.2 Benefits of Python

Benefits of Python include:

Productivity and Speed: Python's concise syntax boosts productivity by enabling efficient, readable code with fewer lines, which increases productivity.

Community and Support: Developers are provided ample documentation, tutorials, and support by Python's big and active community.

Flexibility: Python's versatility makes it a powerful language for diverse applications, as it can be used in a wide variety of domains.

Integration: Python easily integrates with other languages such as C, C++, and Java, enabling developers to leverage existing codebases.

Scalability: Python's performance can be further enhanced by utilizing libraries such as NumPy or Cython for computationally intensive tasks.

Career Opportunities: Python's widespread adoption has created high demand for programmers, making it a top choice for technical career opportunities.

1.3.3 Usage of Python

Web Development: Extensive Web Development with scalable and secure applications is enabled.

Data Science (DS) and Machine Learning (ML): Python dominance in this field is due to various libraries.

Game Development: Python creates simple games and prototypes using libraries.

Automation and Scripting: Cross-platform and ease of use makes it easy for automation and writing scripts.

Scientific Computing: Python is used in various scientific fields.

Desktop Applications: Python is used to build desktop applications using frameworks.

1.4 Recommended Integrated Development Environment (IDE) for Python

Several popular IDEs for Python developments are widely used and highly regarded in the Python community, each offering its own set of features and capabilities.

PyCharm is a powerful IDE developed by JetBrains, is offered in both free (Community Edition) and paid (Professional Edition) versions. It provides code completion, intelligent code analysis, debugging, version control integration, and more.

Visual Studio Code (VS Code) is a lightweight yet potent code editor with an extensive extension library. When equipped with the right extensions, it is favored by programmers for its speed and customizability, offering a comprehensive Python development experience.

Jupyter Notebooks and JupyterLab are interactive computing environments used for combining code, documentation, and visualizations in a single document. They find wide applications in data analysis, data visualization, and machine learning tasks.

Spyder, explicitly designed for scientific computing with Python, offers a MATLAB-like interface, a variable explorer, an IPython console, and a powerful code editor.

Sublime Text, while not Python-specific, remains popular among Python programmers as a highly customizable and lightweight code editor.

Atom, similar to Sublime Text, can be adapted to support Python development through various packages and plugins, making it a versatile and extensible code editor.

Python's basic IDE, called Integrated Development and Learning Environment (IDLE), provides a simple and straightforward interface but lacks some advanced features found in other IDEs.

Ultimately, the best IDE for Python depends on the programmer's individual requirements, comfort level with the interface, and specific project requirements.

1.5 Python Installation

1.5.1 Python Installation for Windows

The official Python Website (<https://www.python.org>) should be accessed using a Web browser.

In the 'Downloads' section, the '**Latest Python X.XX**' button (replace 'X.XX' with the latest version number) can be clicked.

The appropriate Windows installer for the system, usually '**Windows installer (64-bit)**' for most modern PCs, is selected at the bottom of the page.

The downloaded installer is then executed.

During the installation process, ensure that the box '**Add Python X.X to PATH**' is checked.

The *installation wizard* will guide the user and Python will be installed on the Windows machine.

1.5.2 Python Installation for macOS

For macOS, Python is usually pre-installed. The version can be checked by opening the Terminal and typing `python3 --version`. If a specific version must be installed, a package manager such as **Homebrew** can be used.

To install, use **Terminal** – It serves as a command-line interface, allowing users to interact with their computer using text-based commands. It can execute various command-line applications, including text-based shells, all within a convenient multi-tabbed window. By default, it provides built-in support for running

Command Prompt, PowerShell, and Bash on Windows Subsystem for Linux, offering users a seamless experience across different operating systems and shell environments.

If Homebrew is not already installed, following command can be pasted into the Terminal to install it:

```
/bin/bash -c "$(curl -fsSL  
https://raw.githubusercontent.com/Homebrew/install/HEAD/  
install.sh)"
```

After Homebrew is installed, Python can be installed using this command:

```
brew install python
```

1.5.3 PyCharm Installation for Windows and macOS

The JetBrains Website (<https://www.jetbrains.com/pycharm/>) should be accessed using a Web browser.

The free Community edition or the Professional edition (if there is a license) of PyCharm can be downloaded, depending on the user's requirements.

After the download is complete, the installer should be run and the on-screen instructions should be followed.

Running PyCharm:

After PyCharm is installed, it can be run from the Start menu on Windows or from the Applications folder on macOS.

Configuring Python Interpreter in PyCharm:

When PyCharm is opened for the first time, the user will be asked to configure the Python interpreter. The Python version installed earlier (example: Python X.X) should be chosen as the interpreter in PyCharm.

Once PyCharm is opened,

File → Settings (or **Preferences** on macOS) → **Project** → **Python Interpreter** is accessed.

The gear icon is clicked and '**Add...**' is selected.

System Interpreter is chosen and the Python executable installed earlier is browsed and selected.

Clicking **OK** will save the interpreter configuration.

Now, Python and PyCharm are set up on the Windows or macOS system, and Python coding can be done using the PyCharm IDE.

1.6 Basics of Python Programming

1.6.1 Code Blocks, Indentation, and Comments

In Python, blocks of code are defined using indentation instead of curly braces, as is done in many other programming languages. Consistent indentation is considered essential for Python code to work correctly. Typically, four spaces are used for indentation, though tabs can also be used (although mixing spaces and tabs is not recommended).

Comments can be added in Python to provide explanations or clarifications. Comments start with the # symbol and extend to the end of the line.

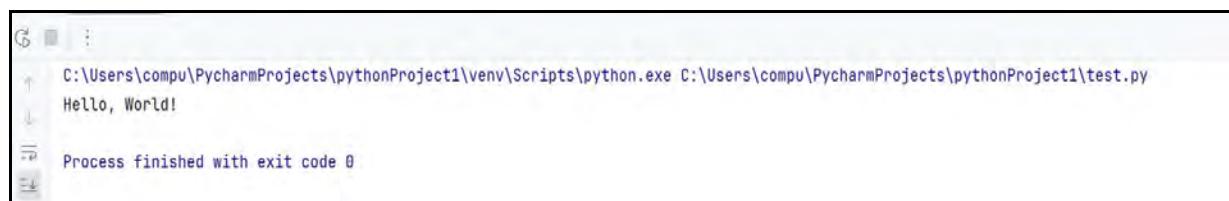
Code Snippet 1 shows how to add a comment.

Code Snippet 1:

```
# This is a comment
print("Hello, World!") # This is another comment
```

In the Code Snippet 1, Python print statement is used to output the string Hello, World! to the console.

Figure 1.1 shows the output of Code Snippet 1.



The screenshot shows the PyCharm terminal window. The command entered is "python test.py". The output displayed is "Hello, World!". Below the output, it says "Process finished with exit code 0".

Figure 1.1: Output of Code Snippet 1

1.6.2 Variables

Data and values are stored in Python using variables, which are assigned values through the utilization of the assignment operator (=).

Code Snippet 2 shows an example with variables of different types.

Code Snippet 2:

```
x = 10          # integer
y = 3.14        # floating-point number
is_student = True # Boolean
```

In Code Snippet 2, the code initializes three variables in Python with different data types. Variable `x` holds an integer value of 10, `y` stores a floating-point number 3.14, and `is_student` is set to the Boolean value True. Comments alongside each variable indicate their respective data types for clarity and documentation.

1.6.3 Data Types

Various data types are supported by Python, encompassing integers, floats, strings, booleans, lists, tuples, dictionaries, and more. The data type does not necessarily require to be declared explicitly; it is dynamically determined by Python based on the value assigned to the variable.

Code Snippet 3 shows how to write data types through several examples.

Code Snippet 3:

```
# Examples of different data types
number = 42
pi = 3.14
name = "Alice"
is_student = True
```

In Code Snippet 3, the code defines variables `number`, `pi`, `name`, and `is_student`, assigning them values of 42 (an integer), 3.14 (a floating-point number), "Alice" (a string), and True (a boolean), respectively. These variables store different data types—integer, floating-point, string, and boolean—representing numerical values, text, and logical states.

1.6.4 Operators

Python includes various operators to perform operations on data, such as arithmetic, comparison, logical, assignment, and more.

Code Snippet 4 shows how to write operators.

Code Snippet 4:

```
# Examples of arithmetic and comparison operators
a = 10
b = 5
addition_result = a + b
is_greater = a > b #True
```

In Code Snippet 4, the code uses arithmetic and comparison operators. Initially, two variables, `a` and `b`, are assigned values of 10 and 5, respectively. The addition operator `+` combines these values to yield 15, stored in `addition_result`. The comparison operator `>` evaluates whether `a` is greater than `b`, resulting in `is_greater` being assigned `True`. This concise example illustrates the fundamental concepts of arithmetic operations and value comparisons in Python.

1.6.5 Conditional Statements

Conditional statements allow you to control the flow of execution based on conditions. The most common ones are `if`, `elif`, and `else`.

Code Snippet 5 shows an example that uses conditional statements.

Code Snippet 5:

```
if age < 18:
    print("You are a minor.")
elif age >= 18 and age < 55:
    print("You are an adult.")
else:
    print("You are a senior citizen.")
```

In Code Snippet 5, the code checks the value of the variable `age`. If it's less than 18, it prints `You are a minor`. If it's between 18 and 54 (inclusive), it prints `You are an adult`. Otherwise, it prints `You are a senior citizen`.

1.6.6 Loops

A block of code is repeatedly executed through the utilization of loops. Python supports `for` and `while` loops.

Code Snippet 6 shows how to write `for` and `while` statements.

Code Snippet 6:

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

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

In Code Snippet 6, the code consists of two parts. The first part utilizes a for loop to iterate over the range from 0 to 4 inclusive, printing each number in the sequence. The second part demonstrates a similar iteration using a while loop. It initializes a count variable to 0, and within the loop, it prints the current count and increments it until the count reaches 5.

Figure 1.2 shows the output of Code Snippet 6.



The screenshot shows the PyCharm terminal window. The command entered is `C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:/Users/compu/PycharmProjects/pythonProject1/test.py`. The output displayed is:
0
1
2
3
4
0
1
2
3
4
Process finished with exit code 0

Figure 1.2: Output of Code Snippet 6

1.6.7 Functions

Blocks of reusable code that perform specific tasks are represented by functions. They aid in code organization and the facilitation of reusability.

Code Snippet 7 shows an example of a function.

Code Snippet 7:

```
def add(a, b):
    return a + b
```

```
result = add(5, 7)
print(result)
# Output: 12
```

In Code Snippet 7, the code defines a function named `add` that takes two parameters and returns their sum. Then, it calls this function with arguments 5 and 7, storing the result in the variable `result`, which is subsequently printed, resulting in the output 12.

1.6.8 Lists, Tuples, and Dictionaries

Lists, Tuples, and Dictionaries are some of the essential data structures in Python.

Code Snippet 8 shows an example of using List, Tuple, and Dictionary.

Code Snippet 8:

```
# List - a collection of items in a specific order
#(mutable)
fruits = ["apple", "banana", "orange"]
# Tuple - a collection of items in a specific order
#(immutable)
coordinates = (10, 20)
# Dictionary - a collection of key-value pairs
person = {"name": "John", "age": 30, "is_student": True}
```

In Code Snippet 8, the code illustrates three essential data structures in Python: lists, tuples, and dictionaries. Lists, denoted by square brackets, are mutable collections of items; tuples, represented by parentheses, are immutable ordered collections; dictionaries, enclosed in curly braces, are key-value pairs allowing for efficient retrieval. These structures offer versatility for organizing and managing data in Python programs.

1.6.9 Input and Output

Python allows you to take user input and display output using the `input()` and `print()` functions, respectively.

Code Snippet 9 shows an example for input and output.

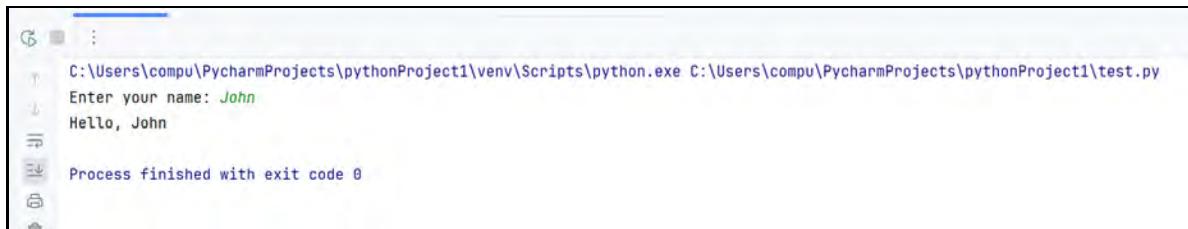
Code Snippet 9:

```
name = input("Enter your name: ")
```

```
print("Hello, ", name)
```

In Code Snippet 8, the code prompts the user to enter their name using the `input()` function and stores the input in the variable `name`. Then, it prints a greeting message using the entered name by concatenating it with the string "Hello," using the `print()` function.

Figure 1.3 shows the output of Code Snippet 9.



The screenshot shows a PyCharm terminal window. The command `C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:\Users\compu\PycharmProjects\pythonProject1\test.py` is run. The terminal prompts for the name with `Enter your name:` , followed by the user input `John`. The program then outputs `Hello, John`. Finally, it displays `Process finished with exit code 0`.

Figure 1.3: Output of Code Snippet 9

1.7 Identifiers in Python

1.7.1 Definition of Identifiers

An identifier in Python is a name employed to recognize a variable, function, class, module, or any other user-defined object. It is essentially a label assigned by programmers to various elements in their code to make them recognizable and accessible.

1.7.2 Valid Identifier

Valid identifiers in Python are names that are governed by the rules for defining identifiers in the language. The key characteristics of valid identifiers are as follows:

Letters (both uppercase and lowercase), digits, and underscores () can be included in them, while they are required to commence with a letter (a-z, A-Z) or an underscore (). Starting with a digit (0-9) is not allowed.

Identifiers are case-sensitive, meaning that names such as `myVariable` and `myvariable` are considered different identifiers.

They cannot be a Python keyword or a reserved word since, they have predefined meanings in the language.

Some valid identifiers in Python include:

```
variable  
my_var
```

```
Count
total_sum
_name
MAX_VALUE
__private_var
this_is_a_long_identifier
```

1.7.3 Invalid Identifier

Invalid identifiers in Python are names that do not adhere to the rules for defining identifiers in the language.

Here are some examples of invalid identifiers.

- Starting with a digit

```
1variable
123Identifier
```

- Containing invalid characters (example, special symbols)

```
@invalid_identifier
my-variable
this_identifier!
```

- Being a Python keyword or reserved word

```
if
for
while
def
class
```

- Having spaces in the name

```
invalid identifier
my variable
```

It is important to note that using invalid identifiers will result in syntax errors or unexpected behavior in your Python code. The rules for defining valid identifiers should always be followed to maintain code correctness and readability.

1.8 Keywords in Python

Keywords in Python are reserved words with specific meanings and purposes within the language. As part of the Python syntax, they cannot be used as variable names or identifiers. Some of the commonly used keywords in Python are mentioned in Table 1.1.

Keyword	Definition
FALSE	Representing the boolean value False.
None	Representing a null or empty value.
TRUE	Representing the boolean value True.
and	Serving as the logical AND operator.
as	Used in import statements or for creating aliases for modules and classes.
assert	Utilized for checking if a condition is true, raising an error if the condition is false.
async	Employed to define asynchronous functions.
await	Used for pausing execution within an async function until an awaited coroutine completes.
break	Used for prematurely exiting from a loop.
class	Used to define a class.
continue	Used to proceed to the next iteration of a loop.
def	Used to define a function.
del	Employed to delete variables or items from a list or dictionary.
elif	An abbreviation of 'else if', used in conditional statements.
else	Used in conditional statements as a fallback when the condition is not met.
except	Used in exception handling to define a block of code that runs when an exception occurs.
finally	Used in exception handling to define a block of code that always runs, regardless of whether an exception occurred.
for	Used to create loops.
from	Used in import statements to import specific parts of a module.
global	Used to indicate that a variable inside a function should be treated as a global variable.
if	Used to define conditional statements.

Keyword	Definition
import	Used to import modules or packages.
in	Used in for loops to iterate over elements in a sequence.
is	Used to test object identity.
lambda	Used to create anonymous functions (also known as lambda functions).
nonlocal	Used in nested functions to indicate that a variable should be treated as a non-local variable.
not	The logical NOT operator.
or	The logical OR operator.
pass	Used as a placeholder for empty blocks of code.
raise	Used to raise exceptions.
return	Used to return a value from a function.
try	Used to start exception handling blocks.
while	Used to create loops.
with	Used to simplify exception handling and resource management with context managers.
yield	Used in generators to produce a value.

Table 1.1: Keywords in Python

It is significant to avoid naming variables with any of these reserved words, as doing so will result in syntax errors.

1.9 Summary

- Python, a high-level, interpreted, and versatile programming language is renowned for its simplicity and readability.
- Python was created in the early 1990s by Guido van Rossum and has a rich history, boasting a large community of users and programmers.
- Extensive standard libraries make Python suitable for various applications, including Web development, data analysis, Artificial Intelligence, and scripting.
- The Python language offers benefits such as easy learning, fast development cycles, and an active open-source ecosystem.
- PyCharm, Visual Studio Code, and Jupyter Notebook are popular IDEs for Python, with PyCharm often being preferred by many programmers.
- A Python program can be created by understanding basic syntax, such as indentation, code blocks, data types, and comments.
- Identifiers in Python are names used for variables, functions, and so on, and must adhere to certain naming rules.
- Python keywords are reserved words with specific meanings and cannot be utilized as identifiers.

1.10 Test Your Knowledge

1. What is Python Programming Language?
 - A) A high-level programming language
 - B) A low-level programming language
 - C) A markup language
 - D) A database management system
2. Which of the following statements about Python's uses and features is correct?
 - A) Python is primarily used for Web development and cannot be used for other purposes
 - B) Python is a statically typed language, meaning data types must be declared explicitly before using variables
 - C) Python has a strong emphasis on readability and uses indentation for code blocks instead of braces or keywords
 - D) Python is not an open-source language, and its usage requires a commercial license
3. Python is widely used in the field of data analysis and scientific computing due to its extensive libraries and tools.
 - A) True
 - B) False
4. Which of the following is a popular Integrated Development Environment (IDE) for Python?
 - A) Firefox
 - B) Microsoft Excel
 - C) Sublime Text
 - D) PyCharm
5. Which of the following is the correct syntax to print 'Hello, World!' in Python?
 - A) `print("Hello, World!")`
 - B) `print("Hello, World!")`
 - C) `Print("Hello, World!")`
 - D) `PRINT("Hello, World!")`

6. Which of the following is a valid identifier in Python?

- A) 123Python
- B) _myVar
- C) @variable
- D) while

7. What are keywords in Python?

- A) Keywords are user-defined names used to identify variables, functions, or classes in Python
- B) Keywords are special words reserved by Python, representing predefined actions or meanings
- C) Keywords are used to define comments and documentation in Python code
- D) Keywords are optional features in Python that can be enabled or disabled based on user preferences

1.10.1 Answers to Test Your Knowledge

1. A high-level programming language
2. Python has a strong emphasis on readability and uses indentation for code blocks instead of braces or keywords
3. True
4. PyCharm
5. `print("Hello, World!")`
6. `_myVar`
7. Keywords are special words reserved by Python, representing predefined actions or meanings

Try It Yourself

1. Write a Python program to print any sentence.
2. Write a Python program that asks the user to enter two numbers and then, prints the sum of those two numbers.



SESSION 02

DATA TYPES AND OPERATORS IN PYTHON

Learning Objectives

In this session, students will learn to:

- ◆ Explain the data types in Python
- ◆ Distinguish between different numeric types
- ◆ Explain type conversion functions in Python
- ◆ Categorize different operators in Python
- ◆ Explain about identifiers
- ◆ Identify the keywords in Python

2.1 Introduction to Data Types in Python Programming

Data types in Python define the kinds of values variables can hold. Python is a dynamically typed language, so explicit specification of data types during variable declaration is not required. The data type is automatically assigned based on the value given to the variable.

Here is an overview of the data types in Python.

2.1.1 Numeric Types

The numeric data type in Python is used for representing data with numeric values, which can encompass integers, floating numbers, and even complex numbers.

`int`: Represents integer values. For example, 1, -5, 100.

`float`: Represents floating-point or decimal values. For example, 3.14, -0.5, 2.0.

2.1.2 Boolean Type

Boolean values are binary in nature, holding only two potential states: True or False. These values are crucial in conditional expressions and logical computations.

2.1.3 Sequence Types

Sequence types in Python are data structures that represent ordered collections of items or elements. These elements can be of different data types and are accessible by their position (index) within the sequence.

Here are some common sequence types in Python:

String Type:

Shows a collection of characters contained within single or double quotation marks. For example, 'hello', 'world'.

List:

Displays an ordered collection of items, which can be of different data types and can be modified (mutable). For example, [1, 'apple', 3.14].

Tuple:

Displays an ordered collection of items, similar to lists but immutable (cannot be modified after creation). For example, (1, 'apple', 3.14).

Set Type:

Shows an unordered collection of unique elements. Duplicates are automatically removed. For example, {1, 2, 3}.

Frozenset:

Represents an immutable set. It is similar to sets but cannot be modified after creation. For example, frozenset({1, 2, 3}).

Mapping Type/Dictionary:

Represents a collection of key-value pairs. Each key must be unique and mapped to a value. For example, {'name': 'John', 'age': 30}.

None Type:

Null value or "None" in Python represents the absence of a value. It is commonly used to indicate that a variable lacks any valid data.

2.2 Numeric Types

In Python, numeric types represent numerical values. Python supports two main numeric data types:

Integer

Integers are numeric data types that represent whole numbers, positive or negative without any fractional part.

You can assign an integer value to a variable using the standard variable assignment syntax.

Code Snippet 1 shows an example of integer representation.

Code Snippet 1:

```
age = 25  
grade = 85
```

In Code Snippet 1, the snippet defines two variables: `age` and `grade`. The variable `age` is assigned a value of 25, indicating the age of an individual, while `grade` is assigned a value of 85.

Float

Float data type represents floating-point numbers, which are numbers that have a fractional part. The numbers can have a positive, negative, or zero value. Floats are used to represent real numbers and can be used for various purposes, such as mathematical calculations and representing decimal values.

Python automatically recognizes numbers with decimal points as float type.

Code Snippet 2 shows an example of float representation.

Code Snippet 2:

```
Height = 6.1  
Weight = 50.22  
Pi = 3.1416
```

In Code Snippet 2, it assigns numerical values to three variables: `Height` is set to 6.1, `Weight` to 50.22, and `Pi` to 3.1416.

2.3 Complex, Binary, and Hexadecimal Types

In addition to `int` and `float`, Python supports several other data types that serve different purposes. The data types frequently utilized in Python include:

Complex Types

A complex type is used to represent complex numbers, which are numbers with both a real and an imaginary part. Complex numbers are denoted as `a + bj`, where `a` is the real part, `b` is the imaginary part, and `j` represents the square root of -1.

To create a complex number in Python, you can use the `complex()` function or simply express the number directly using the `j` suffix for the imaginary part.

Code Snippet 3 shows the usage of a `complex()` function.

Code Snippet 3:

```
# Create a complex number with real part 2 and
#imaginary part 3
z1 = complex(2, 3) # Output: 2 + 3j

# Create a complex number with real part 0 and
#imaginary part -4
z2 = complex(0, -4) # Output: 0 - 4j
```

In Code Snippet 3, the code creates two complex numbers using the `complex` function. `z1` is initialized with a real part of 2 and an imaginary part of 3, resulting in the complex number `2 + 3j`. `z2` is initialized with a real part of 0 and an imaginary part of -4, leading to the complex number `0 - 4j`.

Various mathematical operations can be performed with complex numbers in Python, such as with other numerical types. These operations include addition, subtraction, multiplication, division, and so on.

Code Snippet 4 shows mathematical operations using complex numbers.

Code Snippet 4:

```
z1 = 2 + 3j
z2 = 4 - 1j

# Addition
result_add = z1 + z2 #Output:6 + 2j

# Subtraction
result_sub = z1 - z2 #Output:-2 + 4j

# Multiplication
result_mul = z1 * z2 #Output:11 + 10j

# Division
result_div = z1 / z2 #Output:0.4 + 0.7j
```

In Code Snippet 4, the code performs arithmetic operations on two complex numbers, `z1` (`2 + 3j`) and `z2` (`4 - 1j`). It calculates their addition, subtraction, multiplication, and division, storing the outcomes in `result_add`(`6 + 2j`), `result_sub`(`-2 + 4j`), `result_mul` (`11 + 10j`), and `result_div` (`0.4 + 0.7j`), respectively.

Table 2.1 represents a Few Numeric Data Types.

int	float	complex
10	0.0	<code>3.14j</code>
100	15.20	<code>45.j</code>
-786	-21.9	<code>9.322e-36j</code>
080	<code>32.3+e18</code>	<code>.876j</code>
-0490	-90.	<code>-.6545+0J</code>
-0x260	<code>-32.54e100</code>	<code>3e+26J</code>
0x69	<code>70.2-E12</code>	<code>4.53e-7j</code>

Table 2.1: Representation of a Few Numeric Data Types

Binary Types

The byte data type is a built-in data type in Python that is used to work with binary data. It is often combined with other data types such as strings. It can hold values from 0 to 255 and represents a group of 8 binary digits. Binary data, such as images or audio files and low-level network protocols are commonly manipulated using bytes.

The byte array data type, also present in Python, is a mutable sequence of bytes used for handling binary data. It is commonly combined with other data types, such as strings, to manipulate binary data. Unlike bytes, byte arrays can be modified in place, providing a more flexible approach to binary data manipulation.

Hexadecimal Type

In Python, hexadecimal is a base-16 numbering system utilized for representing numbers, especially when binary data and low-level programming tasks are involved. Hexadecimal numbers utilize digits ranging from 0 to 9 and letters from

A to F. In this system, A signifies 10, B represents 11, C stands for 12, and so forth, up to F, which denotes 15.

Code Snippet 5 shows the representation of Hexadecimal data type.

Code Snippet 5:

```
hex_value = 0x1A    # Hexadecimal representation of  
#the decimal number 26  
  
print(hex_value)    # Output: 26
```

In Code Snippet 5, The code defines a variable `hex_value` with a hexadecimal number `0x1A`, which is the hexadecimal representation of the decimal number 26. It then prints the decimal equivalent of this hexadecimal value, which is 26.

Figure 2.1 depicts the output of this code when executed through PyCharm.

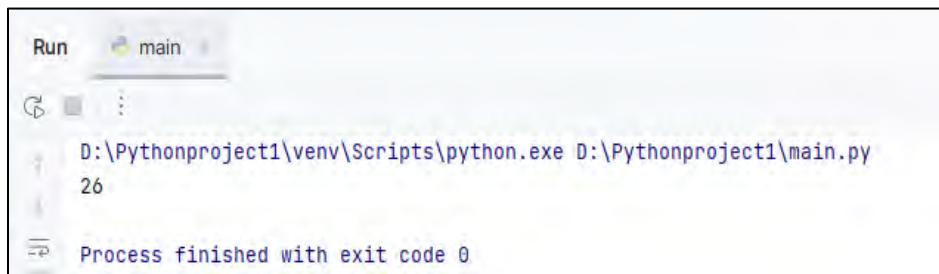


Figure 2.1: Output of Code Snippet 5

2.4 Boolean

The Boolean data type in Python is a fundamental type representing two values: True and False. It is crucial for managing conditional statements and logical operations in Python programming. The program flow is controlled based on the truthiness or falseness of expressions, which are evaluated using Boolean values. It is important to note that the keywords 'True' and 'False' must always be capitalized.

Code Snippet 6 shows representation of using Boolean in Python.

Code Snippet 6:

```
# Assigning Boolean values to variables
is_sunny = True
is_raining = False

# Using Booleans in conditional statements
if is_sunny:
    print("It is a sunny day!")
else:
    print("It is not sunny today.")

# Combining multiple conditions with logical
#operators
temperature = 25
is_summer = True

if temperature > 30 and is_summer:
    print("It is a hot day!")
elif temperature > 20 or not is_summer:
    print("The weather is pleasant.")
else:
    print("It is a bit chilly.")

# Boolean operations
bool_result = True and False
# Outcome of this will be False

bool_result = True or False
# Outcome of this will be True

bool_result = not True
# Outcome of this will be False
```

In Code Snippet 6, the code assigns Boolean values to variables and uses if-else statements for weather-related outputs. It also demonstrates Boolean logic, resulting in specific true or false outcomes through logical operations.

Figure 2.2 depicts the output of this code when executed through PyCharm.

```
C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:\Users\compu\PycharmProjects\pythonProject1\test.py
It is a sunny day!
The weather is pleasant.

Process finished with exit code 0
```

Figure 2.2: Output of Code Snippet 6

In Python, Booleans can also be created through the evaluation of expressions or the result of comparison operations.

Code Snippet 7 shows the evaluation of Boolean expressions.

Code Snippet 7:

```
x = 10
y = 20

is_greater = x > y    # is_greater will be False
is_equal = x == y    # is_equal will be False
is_less_than = x < y # is_less_than will be True
```

In Code Snippet 7, the code compares integers `x` and `y` using relational operators, assigning the results to variables. `is_greater` is `False` because `x` is not greater than `y`, `is_equal` is `False` as `x` does not equal `y`, and `is_less_than` is `True` indicating `x` is less than `y`.

2.5 Type Conversion Functions

Type conversion functions in Python are utilized for converting variables or data from one data type to another. With these functions, the data representation can be altered to enable specific operations or assignments.

The commonly used type conversion functions in Python are listed as follows.

`int()`: Converts a value to an integer

Code Snippet 8 shows conversion of a value to an integer.

Code Snippet 8:

```
num_str = "10"
num_int = int(num_str)
```

```
print(num_int)  
  
# Output: 10
```

In Code Snippet 8, the code converts a string `"10"` to an integer using the `int()` function and prints the integer value `10`.

float(): Converts a value to a floating-point number

Code Snippet 9 shows conversion of a value to a floating-point number.

Code Snippet 9:

```
float_str = "3.14"  
float_num = float(float_str)  
print(float_num)  
  
# Output: 3.14
```

In Code Snippet 9,The code converts the string `"3.14"` to a floating-point number using the `float()` function and prints the number `3.14`.

String(): Converts a value to a string

Code Snippet 10 shows conversion of a value to a string.

Code Snippet 10:

```
number = 42  
string_num = str(number)  
print(string_num)  
  
# Output: "42"
```

In Code Snippet 10,The code converts the integer `42` to a string using the `str()` function and prints the string `"42`.

list(): Converts a value to a list

Code Snippet 11 shows conversion of a value to a list.

Code Snippet 11:

```
num_tuple = (1, 2, 3)
num_list = list(num_tuple)
print(num_list)

# Output: [1, 2, 3]
```

In Code Snippet 11, the code converts a tuple containing the integers `1`, `2`, and `3` into a list with the same elements using the `list()` function and prints the list `[1, 2, 3]`.

`bool()`: Converts a value to a boolean

Code Snippet 12 shows conversion of a value to a boolean.

Code Snippet 12:

```
value = 0
bool_value = bool(value)
print(bool_value)

# Output: False
```

In Code Snippet 12, the code converts the integer `0` to a boolean using the `bool()` function and prints the result. Since `0` is typically interpreted as `False` in boolean context in Python, `bool_value` is set to `False`, which is then printed.

2.6 Special Types

Python has several special types that serve specific purposes and are used in various contexts such as None, strings, lists, tuple, dictionaries, and set. Let us get a basic introduction on each of them.

2.6.1 None Type

In Python, the `None` type is used to represent the absence of a value or a null value. This built-in constant frequently used to indicate that a variable or function does not hold a reference to any valid object. The `None` object is a singleton, meaning there is only one instance of it in memory.

Here are some of the examples on how to use `None` type.

The None type is represented by the keyword `None`. It is not equated with an empty string, zero, or any other false value. `None` is a distinct object representing 'nothing'.

Code Snippet 13 shows how to represent `None` type.

Code Snippet 13:

```
my_variable = None
```

In Code Snippet 13, the code assigns the value `'None'` to the variable `'my_variable'`. `'None'` is a special value in Python used to denote the absence of a value or a null reference.

The `None` type represents the absence of a value or a null value in Python. It is utilized to signify that a variable does not point to any object. The `None` object is a singleton and is often used as a default return value for functions that do not return anything explicitly.

If a function does not explicitly return anything or lacks a return statement, it implicitly returns `None`. This is particularly common for functions with side effects or when there is no meaningful value to return.

Code Snippet 14 shows the return value of functions.

Code Snippet 14:

```
def greet():
    print("Hello, there!")
result = greet()  # Output: "Hello, there!"
print(result)      # Output: None
```

In Code Snippet 14, the code defines a function `'greet()'` that prints `"Hello, there!"` when called. After calling `'greet()'`, the text `"Hello, there!"` is printed. The function does not explicitly return a value, so `'result'` is assigned `'None'`. Printing `'result'` outputs `'None'`.

`None` is often used as a default value for function arguments when you want to allow the caller to omit the argument and use a predefined value.

Code Snippet 15 shows how to represent default value.

Code Snippet 15:

```
def multiply(a, b=None):
    if b is None:
        b = 1
    return a * b
result = multiply(5) # Equivalent to multiply(5, 1)
print(result) # Output: 5
result = multiply(5, 3)
print(result) # Output: 15
```

In Code Snippet 15, the code defines a function named `multiply` that takes two parameters, `a` and an optional `b` with a default value of `None`. If `b` is not provided, it defaults to 1. The function returns the product of `a` and `b`. When `multiply` is called with a single argument, 5, it returns 5, as `b` defaults to 1. Calling `multiply` with 5 and 3 returns 15, demonstrating the function's ability to handle both provided and default arguments.

When comparing to `None`, use `is` instead of `==`, as `is` checks for identity (same object in memory), while `==` checks for equality.

Code Snippet 16 shows how to represent comparison.

Code Snippet 16:

```
a = None
b = None

print(a is None)    # Output: True
print(a == None)    # Output: True
print(a is b)
# Output: True (both are the same None object)
```

In Code Snippet 16, the code checks if variables `'a'` and `'b'`, both initialized with the value `'None'`, are indeed `'None'` and compares them to each other. It prints `'True'` for all checks, confirming that `'a'` is `'None'`, `'a'` equals `'None'`, and `'a'` and `'b'` refer to the same `'None'` object in memory.

Code Snippet 17 shows how `None` type is represented.

Code Snippet 17:

```
Result = None
```

In Code Snippet 17, the variable `Result` is assigned the value `None`, indicating it has no value assigned.

2.6.2 String

Strings, sequences of characters enclosed in single ("") or double ("""") quotes, can contain letters, numbers, symbols, and spaces. Once created, strings cannot be modified directly due to their immutability.

Code Snippet 18 shows how a string is represented.

Code Snippet 18:

```
my_string = "Hello, World!"
```

In Code Snippet 18, the code assigns the string "Hello, World!" to the variable `my_string`.

2.6.3 Lists

Lists are collections of items that are ordered and mutable. Elements of different data types can be contained within a list and it is possible to add, modify, or remove elements from it. Lists are formed by placing elements inside square brackets ([]).

Code Snippet 19 shows how a list is represented.

Code Snippet 19:

```
# Define a list
numbers = [1, 2, 3, 4, 5]

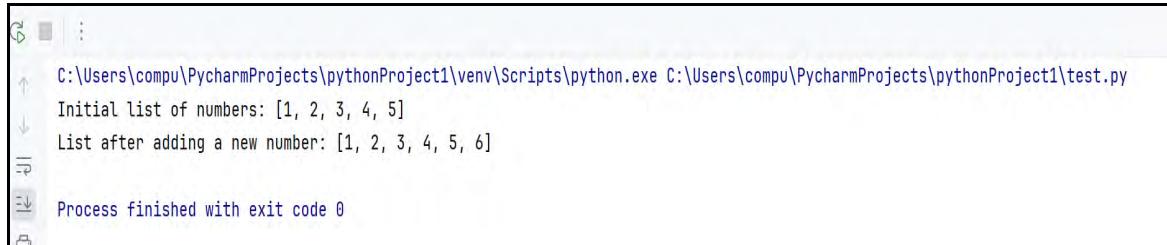
# Print the initial list
print("Initial list of numbers:", numbers)

# Add a new number to the list
new_number = 6
numbers.append(new_number)

# Print the updated list
print("List after adding a new number:", numbers)
```

In Code Snippet 19, the Code Snippet defines a list named `numbers` containing the integers 1 through 5. It then prints this initial list. Afterward, a new number, 6, is added to the list using the `append` method.

Figure 2.3 depicts the output of this code when executed through PyCharm.



The screenshot shows the PyCharm terminal window. The command entered is `C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:\Users\compu\PycharmProjects\pythonProject1\test.py`. The output displayed is:
Initial list of numbers: [1, 2, 3, 4, 5]
List after adding a new number: [1, 2, 3, 4, 5, 6]
Process finished with exit code 0

Figure 2.3: Output of Code Snippet 19

2.6.4 Tuple

Tuples are collections of items that are ordered and immutable, similar to lists, but their elements cannot be changed once created. Tuples are defined using parentheses `()`.

Code Snippet 20 shows how a tuple is represented.

Code Snippet 20:

```
my_tuple = (1, 2, 3, 4, 5)
```

```

# Try to modify the tuple (which should raise an
error)
try:
    my_tuple[0] = 10 # Attempting to modify the first
    element of the tuple
except TypeError as e:
    print("Error occurred:", e)
else:
    print("Tuple modified successfully")

```

In Code Snippet 20, The code attempts to modify the first element of a tuple named `my_tuple`, which contains the integers 1 through 5. Tuples in Python are immutable, meaning their elements cannot be changed once assigned. The attempt to modify the tuple is wrapped in a try-except block to catch the `TypeError` that occurs when trying to modify it. Upon catching the error, the code prints an error message. If no error occurred (which is not possible in this scenario), it would print "Tuple modified successfully."

Figure 2.4 depicts the output of this code when executed through PyCharm.

```

C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:\Users\compu\PycharmProjects\pythonProject1\test.py
Error occurred: 'tuple' object does not support item assignment
Process finished with exit code 0

```

Figure 2.4: Output of Code Snippet 20

2.6.5 Dictionaries

Dictionaries consist of unordered key-value pairs. Each key is unique and used to access its corresponding value. Dictionaries are defined using curly braces ({}) and are useful for fast lookups.

Code Snippet 21 shows how a tuple is represented.

Code Snippet 21:

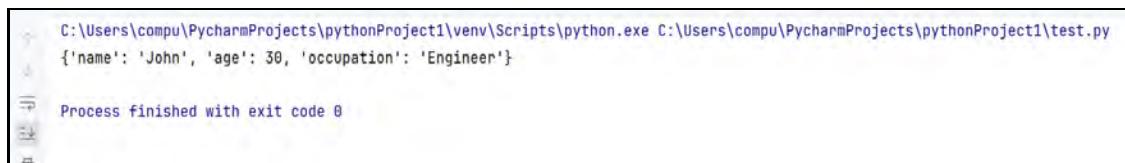
```

person = {"name": "John", "age": 30, "occupation":
"Engineer"}
print(person)

```

In Code Snippet 21 the `person` dictionary contains information about an individual, structured with key-value pairs. Each key represents a specific attribute such as `name`, `age`, and `occupation`, while the corresponding values provide the associated details, such as John for the name, 30 for the age, and "Engineer" for the occupation. This dictionary allows easy access to different aspects of the person's profile, facilitating organized storage and retrieval of information.

Figure 2.5 depicts the output of this code when executed through PyCharm.



```
C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:\Users\compu\PycharmProjects\pythonProject1\test.py
{'name': 'John', 'age': 30, 'occupation': 'Engineer'}
Process finished with exit code 0
```

Figure 2.5: Output of Code Snippet 21

2.6.6 Sets

Sets are collections of unique elements that are unordered. Duplicate values are not allowed in sets. Sets can be defined using curly braces (`{}`) or the `set()` constructor.

Code Snippet 22 shows how a set is represented.

Code Snippet 22:

```
# Define a list with recurring numbers
numbers_list = [1, 2, 3, 4, 5, 5, 6, 6, 7, 8, 8, 9,
9]

# Create a set from the list
unique_numbers_set = set(numbers_list)

# Print the set
print(unique_numbers_set)
```

In Code Snippet 22, the code defines a list named `'numbers_list'` containing integers, some of which are repeated. It then creates a set named `'unique_numbers_set'` from this list, effectively removing any duplicates due to the nature of sets, which only store unique elements. Finally, it prints the set, displaying the unique numbers from the original list.

Figure 2.6 depicts the output of this code when executed through PyCharm.

The screenshot shows a terminal window from PyCharm. The command run was `python test.py`. The output displayed is `{1, 2, 3, 4, 5, 6, 7, 8, 9}`. At the bottom, it says "Process finished with exit code 0".

Figure 2.6: Output of Code Snippet 22

2.7 Escape Chars

Escape characters in Python are special characters that begin with a backslash (\). They serve the purpose of representing characters that might be difficult or even impossible to directly type in a string. With the use of escape characters, it becomes possible to include special characters such as newline (\n), tab (\t), or backslash itself (\) within a string. This allows for greater flexibility and control over the content of strings when working with Python programming language.

- \\n **New Line:** Inserts a new line in the string.
- \\t **Tab:** Inserts a tab space in the string.
- \\ **Backslash:** Inserts a literal backslash in the string.
- ' **Single Quote:** Inserts a single quote within a string that is enclosed in single quotes.
- " **Double Quote:** Inserts a single quote within a string that is enclosed in single quotes.
- \\r **Carriage Return :**The cursor is moved to the beginning of the line without being advanced to a **new line**.

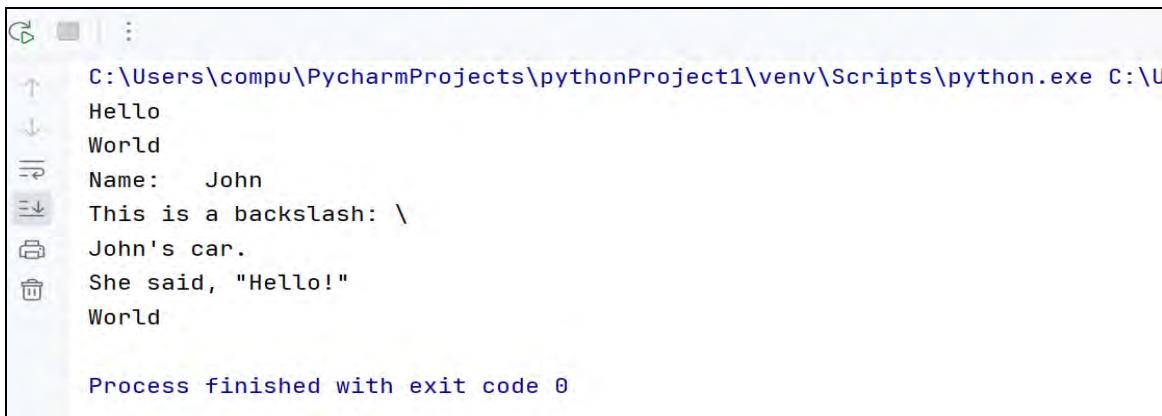
Code Snippet 23 shows how to represent Escape Chars.

Code Snippet 23:

```
# Add a line in the string.  
print("Hello\nWorld")  
  
# Add a tab space in the string  
print("Name:\tJohn")  
  
# Inserts a literal backslash in the string.  
print("This is a backslash: \\")  
  
# Inserts a single quote within a string that is  
# enclosed in single quotes.  
print('John\'s car.')  
  
# Inserts a double quote within a string that is  
# enclosed in double quotes.  
print("She said, \"Hello!\"")  
# Adjust the cursor to the start of the line without  
# jumping to a new line.  
print("Hello\rWorld")
```

In Code Snippet 23, demonstrates the use of escape characters in Python to add new lines, tab spaces, backslashes, single quotes, and double quotes within strings. This technique allows for more complex string formatting.

Figure 2.7 depicts the output of this code when executed through PyCharm.



The screenshot shows the PyCharm terminal window. The command entered is `C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:\U`. The output displayed is:

```
Hello  
World  
Name: John  
This is a backslash: \  
John's car.  
She said, "Hello!"  
World  
  
Process finished with exit code 0
```

Figure 2.7: Output of Code Snippet 23

Escape characters enable creation of strings with special formatting and the inclusion of challenging characters that may be hard to represent directly.

This enhances string flexibility and accommodates the incorporation of special characters that could pose difficulties otherwise.

2.8 Constants

In Python, constants are a special type of variable whose value cannot be changed. Within a module, constants are typically declared and assigned.

The module itself comprises variables, functions, and other code elements in a separate file. This allows the constants to be imported into the main file for use. Although Python lacks built-in support for constants, programmers conventionally employ uppercase variable names to suggest that the variable should be treated as a constant. However, it's important to note that the value of such variables can still technically be changed.

Code Snippets 24a and b show how Constants work in Python.

In the program given in Code Snippet 24a, it will define constants for the freezing and boiling points of water in both Celsius and Fahrenheit scales. Then, it will create a function to convert between the two scales.

Code Snippet 24a:

```
# Constants
FREEZING_C = 0
BOILING_C = 100
FREEZING_F = 32
BOILING_F = 212

# Function to convert from Celsius to Fahrenheit
def celsius_to_fahrenheit(celsius):
    return (celsius * 9/5) + 32

# Function to convert from Fahrenheit to Celsius
def fahrenheit_to_celsius(fahrenheit):
    return (fahrenheit - 32) * 5/9

# Usage
celsius_temp = 25
fahrenheit_temp = celsius_to_fahrenheit(celsius_temp)
```

```

print(f"{celsius_temp} degrees Celsius is
{fahrenheit_temp} degrees Fahrenheit")

fahrenheit_temp = 98.6
celsius_temp = fahrenheit_to_celsius(fahrenheit_temp)
print(f"{fahrenheit_temp} degrees Fahrenheit is
{celsius_temp} degrees Celsius")

```

In Code Snippet 24a, defines constants for the freezing and boiling points of water in Celsius and Fahrenheit and includes functions to convert between these temperature scales, illustrating the use of constants in Python.

Figure 2.8 depicts the output of this code when executed through PyCharm.

```

Run: main
C:\Users\Username\PycharmProjects\pythonProject\venv\Scripts\python.exe
25 degrees Celsius is 77.0 degrees Fahrenheit
98.6 degrees Fahrenheit is 37.0 degrees Celsius

```

Figure 2.8: Output of Code Snippet 24a

In the program given in Code Snippet 24b, it will define a constant named PI based on mathematical constant Pi. It is then used PI to calculate the area and circumference of a circle.

Code Snippet 24b:

```

# Constant.py
PI = 3.14159

# Function to calculate the area of a circle
def calculate_circle_area(radius):
    return PI * radius**2

# Function to calculate the circumference of a
#circle
def calculate_circle_circumference(radius):
    return 2 * PI * radius

# Usage
radius = 5

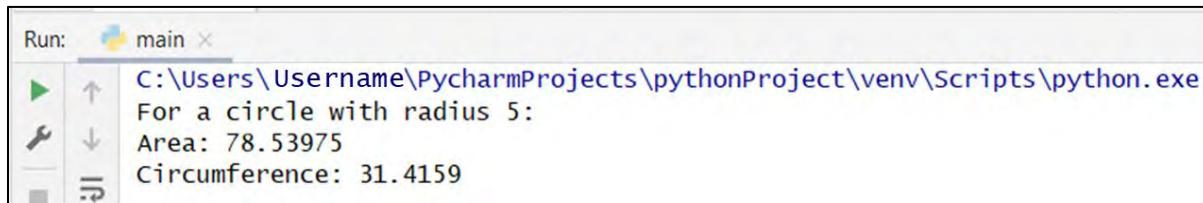
```

```
area = calculate_circle_area(radius)
circumference =
calculate_circle_circumference(radius)

print(f"For a circle with radius {radius}:")
print(f"Area: {area}")
    print(f"Circumference: {circumference}")
```

In Code Snippet 24b, demonstrates defining a constant `PI` and using it to calculate the area and circumference of a circle, showcasing how constants can be used for mathematical calculations.

Figure 2.9 depicts the output of this code when executed through PyCharm.



The screenshot shows the PyCharm interface with a run configuration named 'main'. The terminal window displays the following output:

```
Run: main
C:\Users\Username\PycharmProjects\pythonProject\venv\Scripts\python.exe
For a circle with radius 5:
Area: 78.53975
Circumference: 31.4159
```

Figure 2.9: Output of Code Snippet 24b

In this example, a separate file named 'Constant.py' was created, where `PI` was declared. This constant is used to calculate the area of a circle and circumference respectively of a circle based on a given radius. Since the value of `PI` was not modified throughout the program, it acted as a constant. Constants are useful when we want to use some identifier that does not change its value across the entire program.

2.9 Operators

Operators in Python are symbols or special characters used for performing specific operations on variables or values. A variety of operators are supported by Python, which can be broadly categorized into following types:

Arithmetic Operators, Comparison Operators, Assignment Operators, Logical Operators, Bitwise Operators, Membership Operators, and Identify Operators.

2.9.1 Arithmetic Operators

Basic arithmetic calculations are performed using these operators.

List of arithmetic operators are as follows:

- i. Addition: +
- ii. Subtraction: -

- iii. Multiplication: *
- iv. Division: /
- v. Modulus (remainder): %
- vi. Exponentiation: **
- vii. Floor Division (integer division): //

Consider two values 10 and 3. They are stored in variables a and b.

Code Snippet 25 shows how Arithmetic Operators can be used in Python with these variables.

Code Snippet 25:

```
# Arithmetic Operators
a = 10
b = 3
print(a + b)  # Output: 13
print(a / b)  # Output: 3.333333333333335
print(a % b)  # Output: 1
```

In Code Snippet 25, demonstrates arithmetic operations in Python, such as addition, division, and modulus, showing how to perform basic mathematical calculations with integers.

2.9.2 Comparison Operators

Two values can be compared by these operators and a Boolean result (True or False) is returned.

List of comparison operators are as follows:

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

Code Snippet 26 shows how Comparison Operators can be used in Python.

Code Snippet 26:

```
print(a == b)  # Output: False
print(a > b)  # Output: True
```

```
print(a <= b) # Output: False
```

In Code Snippet 26, comparison operators are used to evaluate the relationship between two variables, illustrating how to compare values for equality, greater than, and less than or equal conditions.

2.9.3 Assignment Operators

Values are assigned to variables using these operators, these are listed as follows:

Assignment: =

Addition assignment: +=

Subtraction assignment: -=

Multiplication assignment: *=

Division assignment: /=

Modulus assignment: %=

Exponentiation assignment: **=

Floor division assignment: // =

Code Snippet 27 shows how Assignment Operators can be used in Python.

Code Snippet 27:

```
c = 5
c += 2           # Equivalent to c = c + 2
print(c)         # Output: 7
```

In Code Snippet 27, this snippet uses an assignment operator to increment a variable's value, highlighting the shorthand method for updating variables in Python.

2.9.4 Logical Operators

Logical operators perform logical operations and return True or False. Some of these are listed as follows:

Logical AND: and

Logical OR: or

Logical NOT: not

Code Snippet 28 shows how Assignment Operators can be used in Python.

Code Snippet 28:

```
x = True
y = False
```

```
print(x and y) # Output: False  
print(x or y) # Output: True  
print(not x) # Output: False
```

In Code Snippet 28, logical operators are demonstrated here, performing logical AND and OR operations on Boolean values, showcasing conditional logic in Python.

2.9.5 Bitwise Operators

Operators that perform bitwise operations on integer values are listed as follows:

- i. Bitwise AND: &
- ii. Bitwise OR: |
- iii. Bitwise XOR: ^
- iv. Bitwise NOT: ~
- v. Left shift: <<
- vi. Right shift: >>

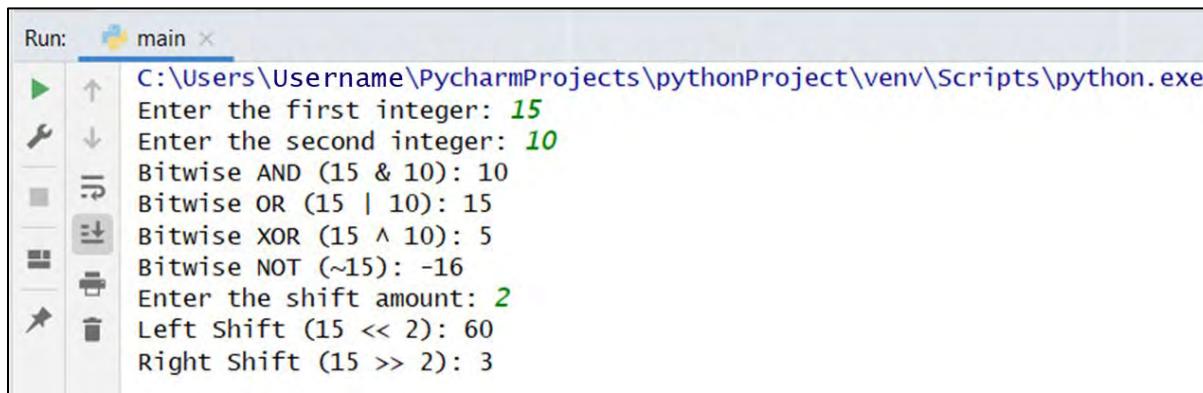
Code Snippet 29 shows how Bitwise Operators can be used in Python.

Code Snippet 29:

```
# Bitwise AND (&)  
result = operand1 & operand2  
  
# Bitwise OR (|)  
result = operand1 | operand2  
  
# Bitwise XOR (^)  
result = operand1 ^ operand2  
  
# Bitwise NOT (~)  
result = ~operand  
  
# Bitwise Left Shift (<<)  
result = operand <<number_of_positions  
  
# Bitwise Right Shift (>>)  
result = operand >>number_of_positions
```

In Code Snippet 28, this snippet exemplifies bitwise operations, including AND, OR, XOR, NOT, left shift, and right shift, demonstrating how to manipulate individual bits within integer values in Python.

Figure 2.10 depicts the output of this code when executed through PyCharm.



The screenshot shows the PyCharm Run window with the title 'Run: main'. The output pane displays the following text:

```
C:\Users\Username\PycharmProjects\pythonProject\venv\Scripts\python.exe
Enter the first integer: 15
Enter the second integer: 10
Bitwise AND (15 & 10): 10
Bitwise OR (15 | 10): 15
Bitwise XOR (15 ^ 10): 5
Bitwise NOT (~15): -16
Enter the shift amount: 2
Left Shift (15 << 2): 60
Right Shift (15 >> 2): 3
```

Figure 2.10: Output of Code Snippet 29

2.9.6 Membership Operators

These operators check for the presence of a value in a sequence (example, string, list, tuple).

Membership (in): in

When using the in operator in Python, if the specified value is present in the sequence (such as a string, list, or tuple), the result will be True; if it is not present, the result will be False.

Not membership (not in): not in

When using the not in operator, if the value is absent in the sequence, the result will be True, and if it is present, the result will be False.

Code Snippet 30 shows how Membership Operators can be used in Python.

Code Snippet 30:

```
fruits = ['apple', 'banana', 'orange']
print('apple' in fruits) # Output: True
print('grape' not in fruits) # Output: True
```

2.9.7 Identity Operators

These operators compare the memory location of two objects.

Identity equality (`is`): is

Identity inequality (`is not`): is not

Code Snippet 31 shows how Identify Operators can be used in Python.

Code Snippet 31:

```
p = [1, 2, 3]
q = [1, 2, 3]
print(p is q)
# Output: False (Different objects in memory)
print(p is not q) # Output: True
```

In Code Snippet 31, the code compares two lists to check if they reference the same object in memory, illustrating the use of identity operators `is` and `is not` for comparison, showing that even identical lists are distinct objects.

2.10 Comments in Python

In Python, lines of text that are intended for explanations, notes, or remarks are treated as comments and are ignored by the Python interpreter during runtime. This allows the codebase to be better understood and maintained by programmers. Python supports two types of comments:

2.10.1 Single-line Comments

Single-line comments in Python are initiated with the hash symbol (#) and continue until the end of the line. Anything appearing after the # symbol on the same line will be treated as a comment and will not be executed by the Python interpreter.

Code 32 shows how Single-line Operators can be used in Python.

Code Snippet 32:

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

In Code Snippet 32, demonstrates how to use single-line comments in Python, allowing for annotations and explanations within the code without affecting the program's execution, showcased by printing a greeting message.

2.10.2 Multi-line comments (Docstrings)

In Python, the usage of triple quotes (''' or ''") is permitted to generate multi-line comments, known as docstrings. Although docstrings are primarily utilized for documenting functions, classes, or modules they can also be employed as multiline comments.

Code Snippet 33 shows how multi-line comments are used in Python.

Code Snippet 33:

```
"""
This is a multi-line comment in Python.
You can use it to add detailed explanations or notes
about your code.

"""

# Your actual code starts here
def my_function():

"""
This is a docstring for the function.
It provides information about what the function does.

"""

print("Hello, world!")
# Calling the function
my_function()

#Output: Hello, world!
```

In Code Snippet 33, introduces the concept of multi-line comments and docstrings in Python, used here to document a function's purpose before printing "Hello, world!", highlighting best practices for code documentation.

2.11 Sequence Type in Python

In Python, sequence types are data types that represent ordered collections of elements, where each element is assigned, a unique index starting from 0. These elements can be of any data type, including integers, strings, tuples, lists, and so on. Common characteristics are shared by sequence types in Python, and they

support specific operations that make them useful in various programming scenarios.

2.11.1 Strings

Strings are sequences of characters and are immutable, meaning once they are created, their contents cannot be changed. You can access individual characters using indexing and slicing.

Code Snippet 34 shows how Strings are used in Python.

Code Snippet 34:

```
my_string = "Hello, World!"  
print(my_string[0]) # Output: 'H'  
print(my_string[7:]) # Output: 'World!'
```

In Code Snippet 34, this snippet creates a range object representing numbers from 0 to 4 and converts it to a list for printing, demonstrating how to generate sequences of numbers in Python.

2.11.2 Range

The range type is used to represent an immutable sequence of numbers within a specified range. It is often used in loops and iterations.

Code Snippet 35 shows how Range is used in Python.

Code Snippet 35:

```
my_range = range(5) # Represents numbers 0 to 4  
print(list(my_range))  
# Output: [0, 1, 2, 3, 4]
```

In Code Snippet 35, the first part of the code demonstrates the creation of a range object representing numbers from 0 to 4 and then converting this range into a list for display purposes.

2.11.3 Bytes and Bytearrays

These types are used to represent sequences of bytes. Bytes are immutable, while bytearrays are mutable.

Code Snippet 36 shows how bytes and bytearray are used in Python.

Code Snippet 36:

```
my_bytes = b"Hello"
print(my_bytes[1]) # Output: 101 (the ASCII code for
'e')

my_bytarray = bytarray(b"Hello")
my_bytarray[0] = 72 # Modify the first byte to
represent 'H'
    print(my_bytarray)

# Output: bytarray(b'Hello')
```

In Code Snippet 36, the code shows accessing an element from a bytes object, `my_bytes`, to print the ASCII code for 'e'. It then modifies a `bytarray` object, `my_bytarray`, to demonstrate its mutability by changing its first element to 'H', showcasing how bytes are immutable but bytarrays can be altered.

Lists, Tuples, and Strings are also categorized as ‘homogeneous’ sequence types since, they can only hold elements of the same type (example, a list of integers, a tuple of strings).

Various common operations are supported by sequence types in Python, such as indexing, slicing, concatenation, repetition, and membership tests. A thorough understanding of these sequence types and their operations can significantly enhance individual Python programming capabilities.

2.12 Summary

- Numeric types, including integers and floating-point numbers are supported in Python.
- Complex numbers, binary, and hexadecimal types can also be used for specific calculations and representations.
- Boolean values, True and False are utilized for logical operations.
- Python provides type conversion functions to convert data between different types when necessary, facilitating seamless operations.
- Special types such as None are used to represent the absence of a value.
- Escape characters allow the inclusion of special characters in strings.
- Constants are fixed values that remain unchanged throughout the program.
- Various operations, such as arithmetic, logical, and comparison are available to manipulate data in Python.
- Comments play a crucial role in code documentation and are not executed during program execution.
- Sequence types such as lists, tuples, and strings are fundamental data structures in Python, used to store collections of elements in an ordered manner.

2.13 Test Your Knowledge

1. Which of the following is a numeric data type in Python?
A) String
B) Boolean
C) Complex
D) List

2. Which function can be used to convert a floating-point number to an integer in Python?
A) `int()`
B) `float()`
C) `str()`
D) `bool()`

3. Logical operators perform _____ operations and return True or False.
A) logical
B) numerical
C) boolean
D) complex

4. What does the None data type represent in Python?
A) An empty string
B) A missing or undefined value
C) A boolean value
D) A numeric value

5. Which of the following is an example of a sequence data type in Python?
A) Set
B) Dictionary
C) Tuple
D) Boolean

6. Which of the following is an escape character in Python?
A) `\n`
B) `\t`
C) `\``
D) All of these

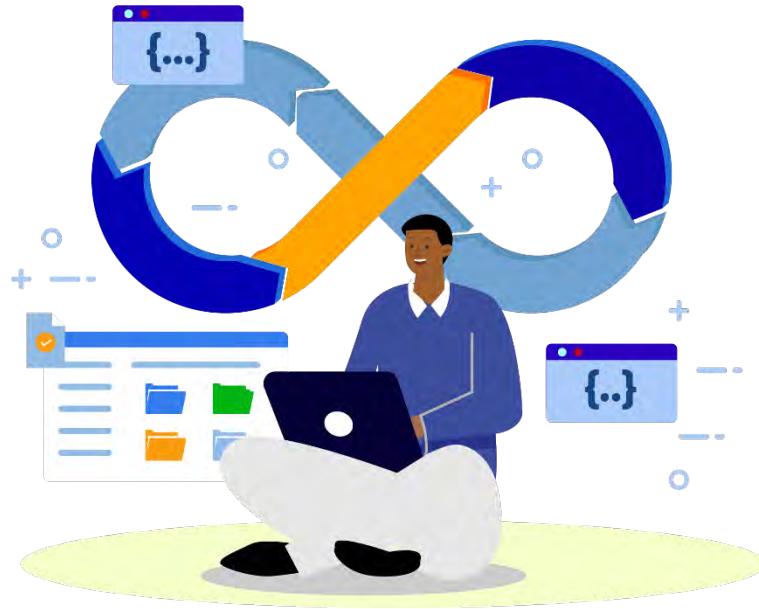
7. Which of the following statements regarding constants in Python is true?
- A) Constants in Python are defined using the 'const' keyword
 - B) Constants in Python can be changed after their initial assignment
 - C) Python does not support constants; all variables can be modified
 - D) Constants in Python are usually written in uppercase letters and cannot be reassigned
8. The '==' operator in Python is used for both assignment and comparison of values.
- A) True
 - B) False
9. Comments in Python are code statements that are executed by the interpreter.
- A) True
 - B) False
10. Which of the following statements about Sequence Types in Python is true?
- A) Sequence types can only contain elements of the same data type
 - B) Lists are immutable sequence types
 - C) Strings are mutable sequence types
 - D) Sequence types preserve the order of elements

2.13.1 Answers to Test Your Knowledge

1. Complex
2. int()
3. logical
4. A missing or undefined value
5. Tuple
6. All of these
7. Constants in Python are usually written in uppercase letters and cannot be reassigned.
8. False
9. False
10. Sequence types preserve the order of elements.

Try It Yourself

1. Write a Python program to calculate the length of a string.
2. Write a Python program to sum all the items in a list.
3. Write a Python program to get the largest number from a list.
4. Create a simple calculator program in Python that allows the user to perform basic arithmetic operations such as addition, subtraction, multiplication, and division.



SESSION 03

FLOW CONTROL STATEMENTS IN PYTHON

Learning Objectives

In this session, students will learn to:

- ◆ Define the use of control flow and its importance
- ◆ Describe the `if`, `elif`, and `else` statements
- ◆ Identify the best practices to maintain code readability when using nested conditionals
- ◆ Explain the `while` and `for` loop and its usage for iterating over sequences
- ◆ Distinguish `break` and `continue` statements and their impact on the loop execution
- ◆ Identify process of exception handling

3.1 Introduction to Flow Control Statement in Python

Flow control statements in Python are employed to manage the sequence in which statements and instructions are executed within a program. It determines the path a program takes through its code based on conditions and decisions made during runtime. Control flow allows you to dictate the sequence in which different parts of your program are executed, enabling you to create dynamic, responsive, and flexible programs.

In Python programming, control flow is essential for various reasons:

Conditional Execution

Control flow statements, such as `if`, `elif`, and `else`, allow you to execute specific blocks of code only if certain conditions are met. This enables your program to make decisions and choose different actions based on different scenarios.

Repetition and Loops

Loops, such as `while` and `for` loops, are crucial for performing repetitive tasks efficiently. They help you avoid writing redundant code by executing a set of instructions multiple times, often with varying inputs or conditions.

Algorithmic Logic

Many algorithms involve steps that require to be executed in a specific order and control flow allows you to implement these algorithms correctly. Sorting, searching, and various mathematical computations often rely on well-structured control flow.

Error Handling

Control flow is used to manage errors and exceptions that may occur during program execution. Using `try` and `except` blocks, you can gracefully handle exceptions, prevent crashes, and provide meaningful error messages to users.

User Interaction

Control flow enables interaction with users by allowing the program to prompt for input, process that input, and provide relevant output based on the user choices or inputs.

Event Handling

Control flow is used to manage errors and exceptions that may occur during program execution. Using `try` and `except` blocks, you can gracefully handle exceptions, prevent crashes, and provide meaningful error messages to users.

Dynamic Behavior

In applications with GUIs or event-driven programming, control flow is essential for responding to user actions or system events in a timely and accurate manner.

Code Organization

Properly structured control flow makes your code more organized, readable, and maintainable. It helps separate different functionalities and keeps related code blocks together.

Efficiency

Control flow statements such as break and continue can improve the efficiency of your code by allowing you to exit loops early or skip certain iterations.

3.1.1 Differences Between Sequential and Conditional Execution

Sequential execution and conditional execution are two fundamental concepts in programming that describe how instructions or statements are executed in a program. Differentiation between these two types of execution are as follows:

Sequential Execution

Sequential execution pertains to the inherent order of statement execution, progressing one after another as they appear in the code.

Each statement is executed in sequence, and the program flows from the top to the bottom of the code without any branching or decision-making. This type of execution is straightforward and is suitable for tasks that involve a linear series of steps.

Code Snippet 1 shows sequential execution in Python. Here all the statements are executed in a straight sequence.

Code Snippet 1:

```
print("Step 1")
print("Step 2")
print("Step 3")
```

Conditional Execution

Conditional execution involves making decisions based on conditions and executing different code blocks depending on whether those conditions are true or false.

Conditional execution uses control flow statements such as `if`, `elif` (`else if`), and `else` to direct the program execution path.

The code inside each conditional block is executed only if the associated condition evaluates to true.

Code Snippet 2 shows conditional execution in Python.

Code Snippet 2:

```
temperature = 25
if temperature > 30:
    print("It is hot!")
elif temperature > 20:
    print("It is warm.")
else:
    print("It is cool.")

#Output: It is warm.
```

3.2 Conditional Statements

Conditional statements in Python are an essential feature that allows you to make decisions in your code based on certain conditions. These statements enable your program to execute different blocks of code depending on whether a given condition is true or false. The primary conditional statements in Python are `if`, `elif` (`else if`), and `else`.

3.2.1 if Statement

The simplest form of a conditional statement is the `if` statement. It allows you to execute a block of code only if a specific condition is true.

Syntax of the `if` statement in Python:

```
if condition:
    # execute the code if the condition is true
```

Code Snippet 3 shows an example of `if` statement in Python.

Code Snippet 3:

```
x = 10
if x > 5:
    print("x is greater than 5")
```

3.2.2 if-else Statement

The if-else statement lets you execute one block of code if a condition is true and another block of code if the condition is false.

Syntax of the if-else statement in Python:

```
if condition:
    # execute the code if the condition is true
else:
    # execute the code if the condition is false
```

Code Snippet 4 shows an example of if-else statement in Python.

Code Snippet 4:

```
# Get user input
age = int(input("Enter your age: "))

# Check eligibility to vote using if-else statement
if age >= 18:
    print("You are eligible to vote.")
else:
    print("You are not eligible to vote yet.")

#Output: Enter your age: 33
You are eligible to vote.
```

if-elif-else Statement

The if-elif-else statement allows you to check multiple conditions in sequence and execute the corresponding block of code for the first true condition encountered. If no conditions are true, the else block is executed (if provided).

Syntax of if-elif-else statement in Python:

```
If condition1:  
    # execute the code if condition1 is true  
elif condition2:  
    # execute the code if condition2 is true  
else:  
    # execute the code if neither condition1 nor #  
# condition2 is true
```

Code Snippet 5 shows an example of if-elif-else statement in Python.

Code Snippet 5:

```
# Get user input  
score = int(input("Enter your exam score: " ))  
  
# Determine the grade using if-elif-else statement  
if score >= 90:  
    grade = "A"  
elif score >= 80:  
    grade = "B"  
elif score >= 70:  
    grade = "C"  
elif score >= 60:  
    grade = "D"  
else:  
    grade = "F"  
  
# Display the grade  
print(f"Your grade is: {grade}")
```

Figure 3.1 depicts the output of this code when executed through PyCharm.

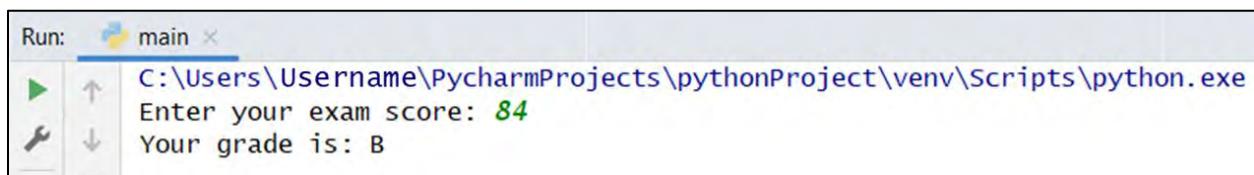


Figure 3.1: Output of Code Snippet 5

Nested if Statements

You can also nest conditional statements within each other to create more complex decision-making structures.

Code Snippet 6 shows how a nested if statement is used.

Code Snippet 6:

```
x = 10
if x > 0:
    if x % 2 == 0:
        print("x is a positive even number")
    else:
        print("x is a positive odd number")
else:
    print("x is not a positive number")

#Output: x is a positive even number
```

3.3Loops

Loops in Python are control structures that allow you to repeatedly execute a block of code. They are essential for performing repetitive tasks, iterating over sequences, and automating actions. Python supports two main types of loops: the `for` loop and the `while` loop.

3.3.1 For Loop

The `for` loop is used to iterate over a sequence (such as a `list`, `tuple`, `string`, and so on.) and execute a block of code for each element in the sequence.

Syntax of the `for` loop in Python:

```
for element in sequence:
    # execute the code for each element
```

Code Snippet 7 shows how the `for` loop is used in Python.

Code Snippet 7:

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

```
for fruit in fruits:  
    print(fruit) # apple, banana, cherry
```

3.3.2 while Loop

The `while` loop repeatedly executes a block of code as long as a certain condition remains true.

Syntax of `while` loop in Python:

```
While condition:  
    # code to execute while condition is true
```

Code Snippet 8 shows how `while` loop is used in Python.

Code Snippet 8:

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

3.4 switch-like Behavior with Dictionaries

To simulate `switch-like` behavior using dictionaries in Python, you can use the dictionary to map keys (cases) to corresponding values (actions or functions). A step-by-step example is explained as follows:

Code Snippet 9 shows how to achieve `switch-like` behavior using dictionaries.

Code Snippet 9:

```
def get_letter_grade(grade):  
    grade_switch = {
```

```

        90: "A",
        80: "B",
        70: "C",
        60: "D"
    }

    for cutoff, letter in grade_switch.items():
        if grade >= cutoff:
            return letter

    return "F"

# Test the function
numerical_grade = int(input("Enter numerical grade:"))
letter_grade = get_letter_grade(numerical_grade)
print("Letter grade:", letter_grade)

#Output: Enter numerical garde: 82
Letter grade: B

```

The `get_letter_grade` function takes a numerical grade as input and defines a `grade_switch` dictionary that maps cutoffs to corresponding letter grades.

To implement a simple calculator that performs basic arithmetic operations, use a series of `if` or `elif` statements. This can be done using a dictionary to map operator symbols to corresponding functions.

Code Snippet 10 shows how to create a calculator using switch-like behavior using dictionaries in Python.

Code Snippet 10:

```

def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):

```

```

        return x * y

def divide(x, y):
    return x / y

operator_functions = {
    '+': add,
    '-': subtract,
    '*': multiply,
    '/': divide
}

operator = input("Enter an operator (+, -, *, /): ")
if operator in operator_functions:
    num1 = float(input("Enter the first number: "))
    num2 = float(input("Enter the second number: "))
    result = operator_functions[operator](num1, num2)
    print("Result:", result)
else:
    print("Invalid operator")

```

Figure 3.2 depicts the output of this code when executed through PyCharm.

```

Run: main ✘
C:\Users\Username\PycharmProjects\pythonProject\venv\Scripts\python.exe
Enter an operator (+, -, *, /): *
Enter the first number: 120
Enter the second number: 40
Result: 4800.0

```

Figure 3.2: Output of Code Snippet 10

3.5 Loop Control Statements

Loop control statements in Python allow you to control the flow of loops by altering their execution based on certain conditions. The main loop control statements are `break`, `continue`, and the optional `else` clause for loops.

break

The `break` statement allows an early exit from a loop, even when the loop condition remains valid. Upon encountering a `break`, the loop instantly concludes, and the program proceeds to the subsequent statement following the loop.

Code Snippet 11 shows an example of the `break` statement.

Code Snippet 11:

```
for i in range(5):
    if i == 3:
        break# Breaks out of loop when i is 3
    print(i)

#Output: 0
1
2
```

Continue

The `continue` statement is employed to bypass the ongoing iteration of a loop and move on to the subsequent iteration. The loop continues to execute, but the statements after the `continue` within the loop block are skipped for that iteration.

Code Snippet 12 shows an example of the `continue` statement.

Code Snippet 12:

```
for i in range(5):
    if i == 2:
        continue # Skips printing when i is 2
    print(i)
# Output: 0
1
2
3
4
```

Loop else Clause

Python allows you to use an optional `else` clause with loops, which is executed only if the loop completes normally (that is, not terminated by a `break` statement). This can be used for a special action at the end of a loop or to check for conditions that were not met during iteration.

Code Snippet 13 shows an example of using `else` statement in a loop.

Code Snippet 13:

```
for i in range(5):
    print(i)
else:
    print("Loop completed without break.")

#Output: 0
1
2
3
4
Loop completed without break
```

Pass

Although not a loop control statement, the `pass` statement is a placeholder that does nothing. It is used when syntactically a statement is required, but the developer does not want any code to be executed. Leaving the code empty would cause an error to be raised because empty code is not allowed in loops, function definitions, class definitions, or in `if` statements. In such a case, `pass` is useful.

Code Snippet 14 shows an example of `pass` statement.

Code Snippet 14:

```
for i in range(3):
    pass # Placeholder for future code
# No output is generated because pass statement does
nothing
```

3.6 Iterating with `enumerate()` and `zip()`

`enumerate()` and `zip()` are two powerful functions in Python that enhance ability to iterate over sequences and collections. They provide convenient ways to work with data and perform various operations during iteration.

3.6.1 `enumerate()` Function

The `enumerate()` function adds a counter to an iterable (such as a `list`, `tuple`, or `string`) and returns an `enumerate` object that produces pairs of index and value during iteration.

Code Snippet 15 shows how `enumerate` function works in Python.

Code Snippet 15

```
fruits = [ "apple", "banana", "cherry" ]
for index, fruit in enumerate(fruits):
    print(f"Index: {index}, Fruit: {fruit}")
```

Code Snippet 16 specifies a starting index for the counter.

Code Snippet 16:

```
For index, fruit in enumerate(fruits, start=1):
print(f"Item {index}: {fruit}")

#Output:
Item 1: apple
Item 2: banana
Item 3: cherry
```

3.6.2 `zip` Function

The `zip()` function combines multiple iterables into an iterator that generates tuples containing elements from each iterable, paired together element-wise.

Code Snippet 17 shows how `zip` function works in Python.

Code Snippet 17:

```
numbers = [1, 2, 3]
letters = ['A', 'B', 'C']
for num, letter in zip(numbers, letters):
    print(num, letter)

#Output:
1 A
2 B
3 C
```

Code Snippet 18 shows if the input iterables are of different lengths, `zip()` stops generating tuples when the shortest iterable is exhausted.

Code Snippet 18:

```
numbers = [1, 2, 3]
letters = ['A', 'B']
for num, letter in zip(numbers, letters):
    print(num, letter)

#Output:
1 A
2 B
```

3.6.3 Using `enumerate()` and `zip()` Together

You can combine `enumerate()` and `zip()` to iterate over two or more sequences while tracking their indices.

Code Snippet 19 shows the combination of `enumerate` and `zip` functions together.

Code Snippet 19:

```
Names = ["Alice", "Bob", "Charlie"]
ages = [25, 30, 22]
```

```
for index, (name, age) in enumerate(zip(names, ages)):
    print(f"Person {index + 1}: {name}, Age: {age}")

#Output:
Person 1: Alice, Age: 25
Person 2: Bob, Age: 30
Person 3: Charlie, Age: 22
```

3.7 List Comprehensions

List comprehensions are a concise and efficient way to create lists in Python. They provide a compact syntax for generating lists by applying an expression to each item in an `iterable`(such as a `range`, a `list`, or a `string`) and collecting the results.

List comprehensions can make code more readable and reduce the necessity for explicit loops.

Code Snippet 20 shows the basic structure of a list comprehension.

Syntax:

```
new_list = [expression for item in iterable]:
```

`expression` is the operation or computation to be applied to each item in the `iterable`.

`item` is the variable representing each element in the `iterable`.

`iterable` is the sequence you are iterating over (such as a `range`, a `list`, or a `string`).

List Comprehensions can include conditions to filter items before they are added to the new list as follows:

```
new_list = [expression for item in iterable if
<condition>]
```

Code Snippets 20 and 21 shows how List Comprehensions work in Python.

Code Snippet 20:

```
Creating a list of squares:  
numbers = [1, 2, 3, 4, 5]  
squares = [x**2 for x in numbers]  
#output: squares = [1, 4, 9, 16, 25]
```

Code Snippet 21:

```
Generating pairs from two lists:  
  
colors = ['red', 'green', 'blue']  
fruits = ['apple', 'grape', 'berry']  
pairs = [(color, fruit) for color in colors for fruit  
in fruits]  
for pair in pairs:  
    print(pair)  
  
#Output:  
('red', 'apple')  
('red', 'grape')  
('red', 'berry')  
('green', 'apple')  
('green', 'grape')  
('green', 'berry')  
('blue', 'apple')  
('blue', 'grape')  
('blue', 'berry')
```

3.8 Error Handling

Error handling using `try` and `except` blocks in Python is a fundamental concept that allows you to handle and manage exceptions (errors) that occurs during the execution of your code. By using these blocks, you can write code that gracefully handles errors, provides useful feedback, and prevents crashes.

Code Snippet 22 shows an example of using `try` and `except` in Python.

Code Snippet 22:

```
Try:  
    # Code that might raise an exception  
    result = 10 / 0 # Division by zero will raise a  
# ZeroDivisionError  
  
    value = int("abc") # Conversion will raise a  
# ValueError  
  
except ZeroDivisionError:  
print("Cannot divide by zero.")  
  
except ValueError as e:  
print(f"ValueError: {e}")  
#output: Cannot divide by zero.
```

In the example provided, it is evident that:

- The `try` block contains the code that you want to monitor for exceptions.
- If an exception occurs, the code execution within the `try` block stops and the corresponding `except` block is executed.

You can have multiple `except` blocks to catch different types of exceptions.

The `except` blocks specify the type of exception to catch (for example, `ZeroDivisionError`, `ValueError`).

You can use the `as` keyword to assign the exception object to a variable for further examination or printing.

3.8.1 Display Custom Error Messages

In the `except` block, print a custom error message that explains the nature of the error and provides context to the user or developer.

Code Snippet 23 shows the error message along with an explanation.

Code Snippet 23:

```
try:  
    value = int("abc")  
except ValueError:  
    print("Error: Invalid value provided. Please enter a  
valid integer.")  
  
# Error: Invalid value provided. Please enter a  
valid integer.
```

3.9 Summary

- Control flow involves sequence of executing instructions within a program.
- Distinct code blocks can be executed based on different conditions through the utilization of `if`, `elif`, and `else` statements.
- Nested conditionals in Python involve placing one set of conditional statements within another.
- `while` loops and `for` loops in Python enable iterative execution of code based on specified conditions and sequences.
- The `break` statement ends the current loop or iteration, while `continue` skips the rest of the current iteration and moves to the next one.
- In Python, specific types of exceptions can be handled using `try` and `except` blocks.

3.10 Test Your Knowledge

1. Control flow is responsible for sequential execution of statements in a program, which in turn affects the program logic and behavior.
 - A) True
 - B) False

2. Which of the following statements is true regarding `if`, `elif`, and `else` statements in Python?
 - A) An `else` statement can be used without an `if` statement
 - B) `elif` stands for `end if`
 - C) An `if` statement can have only one corresponding `elif` statement
 - D) `else` statements are used to handle multiple conditions within the same block

3. In nested conditionals, the inner conditional statements are evaluated and executed before the outer conditional statements.
 - A) True
 - B) False

4. Which statement accurately describes the key difference between a `while` loop and a `for` loop in Python?
- A) A `while` loop is used for iterating over sequences, while a `for` loop is used to execute a block of code repeatedly as long as a condition is met
 - B) A `for` loop is ideal for indefinite loops, while a `while` loop is suitable for iterating over a sequence of elements
 - C) In a `for` loop, the number of iterations is determined by a condition, whereas a `while` loop iterates over a fixed range of values
 - D) A `while` loop can only iterate over lists, while a `for` loop can work with various data structures
5. In a loop, what happens when a `break` statement is encountered?
- A) The loop terminates immediately and control passes to the next iteration
 - B) The loop skips the current iteration and proceeds to the next one
 - C) The loop execution continues normally, but the `break` statement is ignored
 - D) The loop enters an infinite loop and does not terminate
6. What is the purpose of the `try` and `except` blocks in Python?
- A) They are used to define loops within loops, creating nested iterations
 - B) They are used to handle specific types of exceptions and provide alternative code paths when errors occur
 - C) They are used to define global variables that can be accessed from anywhere in the program
 - D) They are used to concatenate strings and manipulate text data

3.10.1 Answers to Test Your Knowledge

1. True
2. An `else` statement can be used without an `if` statement
3. True
4. A `for` loop is ideal for indefinite loops, while a `while` loop is suitable for iterating over a sequence of elements
5. The loop terminates immediately and control passes to the next iteration
6. They are used to handle specific types of exceptions and provide alternative code paths when errors occur

Try It Yourself

1. Write a Python program to calculate the length of a string.
2. Write a Python program to sum all the items in a list.
3. Write a Python program to get the largest number from a list.
4. Create a simple calculator program in Python that allows the user to perform basic arithmetic operations such as addition, subtraction, multiplication, and division.



SESSION 04

FUNCTIONS AND FUNCTION PARAMETERS IN PYTHON

Learning Objectives

In this session, students will learn to:

- ◆ Define the basics of functions in Python
- ◆ Explain the types of arguments and concept of parameters
- ◆ Identify the user-defined functions in Python
- ◆ Explain lambda functions and callables in Python

4.1 Introduction to Functions in Python

In Python, a function refers to a segment of reusable code designed to execute a distinct task. Functions enable the code to be broken down into smaller, manageable pieces, leading to increased organization, readability, and ease of maintenance. Functions promote code reusability and modularity, which are fundamental principles of good programming practices.

Benefits of using functions are as follows:



Readability: Functions contribute to an improved readability of code by allowing segregation of tasks into distinct, named sections. This separation makes it easier for programmers to comprehend the logic and functionality of the codebase.



Reusability: The utilization of functions facilitates enhanced code reusability by encapsulating specific tasks or functionalities. These encapsulated segments can be employed multiple times throughout the codebase, reducing redundancy and promoting efficiency.

4.1.1 Syntax to Declare a Function

In Python, defining a function involves utilizing the `def` keyword, followed by the chosen function name enclosed in parentheses. These parentheses encompass any parameters that the function is designed to take in.

Syntax of a function in Python:

```
def function_name(parameters):
    # Function body
    # Perform tasks here
    return result # Optional
```

4.1.2 Components of a Function

Function Definition: It starts with the `def` keyword, followed by the function name. Function names should be descriptive and follow the lowercase with underscores (`snake_case`) naming convention. Parentheses `()` are used to enclose the parameters that the function accepts.

Parameters: Parameters are placeholders for the values that the function expects to receive when it is called. They provide a way to pass information into the function. Parameters are listed within the parentheses after the function name.

Function Body: The indented block of code inside the function specifies the function's behavior. It contains the logic and operations that are performed when the function is called.

return Statement: The return statement is employed to indicate the value that the function provides upon its execution. Not all functions must return a value; functions can also execute actions without providing any return value.

4.1.3 Types of Functions in Python

In Python, there are several types of functions, each serving a specific purpose and offering different features.

Built-in Functions: These are functions that are pre-defined in Python and are readily available for use without any additional imports.

Examples include `print()`, `len()`, `max()`, `min()`, and `str()`.

User-Defined Functions: These functions are generated by the programmer to perform specific tasks. They provide code modularity and reusability. You define them using the `def` keyword.

4.1.4 Creating a Function

Code Snippet 1 shows an example of a simple Python function.

Code Snippet 1:

```
def greet():
    print("Welcome to the course on Python")
```

In Code Snippet 1, you have a simple function named `greet()` that, when called, prints the message "Welcome to the course on Python".

4.1.5 Calling a Python Function

After a function has been defined in Python, it can be invoked by using the function name followed by parentheses, which may contain the required parameters for that function.

Code Snippet 2 shows an example to call a function.

Code Snippet 2:

```
# A simple Python function
def greet():
    print("Welcome to the course")

# Driver code to call a function
greet()
```

In Code Snippet 2, another instance of a simple function named `greet` is provided, which prints "Welcome to the course" when executed. This snippet further illustrates the use of functions to execute a block of code upon calling, reinforcing the concept of code reuse and modularity.

4.2 Arguments

Arguments in Python are values that are passed to a function when it is called, enabling the function to work with specific data. They provide a way to customize a function behavior and process different inputs such as numeric values, strings, and lists. Functions typically require input data to perform their tasks, and arguments fulfill this requirement.

4.2.1 Types of Function Arguments

Various types of arguments that can be passed during the function call are supported by Python. Python encompasses four types of function arguments.

Default Argument

In Python, default values can be assigned to the parameters of a function. These default values are used if the caller of the function does not provide a value for that parameter. This concept is known as using default arguments.

Syntax of default arguments in Python:

```
def function_name(parameter1=default_value1,  
                  parameter2=default_value2, ...):  
    # Function code goes here
```

Code Snippet 3 shows an example demonstrating the use of default arguments in a function.

Code Snippet 3:

```
def greet(name, greeting="Hello"):  
    print(f"{greeting}, {name}!")  
  
# Calling the function with only the 'name' parameter  
greet("Alice") # Output: Hello, Alice!  
  
# Calling the function with both 'name' and 'greeting'  
# parameters  
greet("Bob", "Hi") # Output: Hi, Bob!
```

In Code Snippet 3, the `greet` function is enhanced with default arguments, enabling personalized greetings. This example demonstrates how functions can be designed to accept parameters with default values, making some arguments optional and showcasing the flexibility of function parameters.

Keyword Arguments (Named Arguments)

Keyword arguments allow you to pass values to a function using the parameter names as keywords, rather than relying on the order of the arguments. This provides more clarity and flexibility when calling functions, especially if the function has multiple parameters.

Syntax of keyword arguments in Python:

```
def example_function(parameter1, parameter2):  
    # Function code here  
  
    # Using keyword arguments to call the function  
example_function(parameter1=value1,  
                  parameter2=value2)
```

Code Snippet 4 shows an example that demonstrates the use of keyword arguments.

Code Snippet 4:

```
def describe_person(name, age, city):
    print(f"{name} is {age} years old and lives in {city}.")

# Calling the function using positional arguments
describe_person("Alice", 30, "New York")

# Calling the function using keyword arguments
describe_person(age=25, city="Los Angeles", name="Bob")

# Output:
# Alice is 30 years old and lives in New York.
# Bob is 25 years old and lives in Los Angeles.
```

In Code Snippet 4, the `describe_person` function is introduced, accepting name, age, and city as parameters. It exemplifies how functions can use both positional and keyword arguments for more readable and flexible code.

Positional Arguments

In Python functions, positional arguments are the simplest form of arguments. They are passed to a function in the order they appear in the function parameter list. The argument's position signifies its correspondence with a specific parameter.

Syntax of positional arguments in Python:

```
def example_function(parameter1, parameter2):
    # Function code here

# Calling the function using positional arguments
example_function(value1, value2)
```

Code Snippet 5 shows an example that demonstrates the use of positional arguments.

Code Snippet 5:

```
def describe_person(name, age, city):
    print(f"{name} is {age} years old and lives in {city}.")

# Calling the function using positional arguments
```

```
describe_person( "Alice" , 30 , "New York" )  
# Output: Alice is 30 years old and lives in New York.
```

In Code Snippet 5, the `describe_person` is utilized to demonstrate the use of positional arguments in detail. This snippet highlights how arguments are matched to parameters based on their position.

Arbitrary Arguments (variable-length arguments *args and **kwargs)

In Python, functions can be defined to accept a variable number of arguments, which may include both positional and keyword arguments. These variable-length argument lists are often referred to as arbitrary arguments. There are two ways to define arbitrary arguments: using `*args` for positional arguments and `**kwargs` for keyword arguments.

Arbitrary Positional Arguments (*args)

The syntax `*args` enables a function in Python to take any number of positional arguments as required. The arguments are collected into a tuple and they can be accessed using index notation.

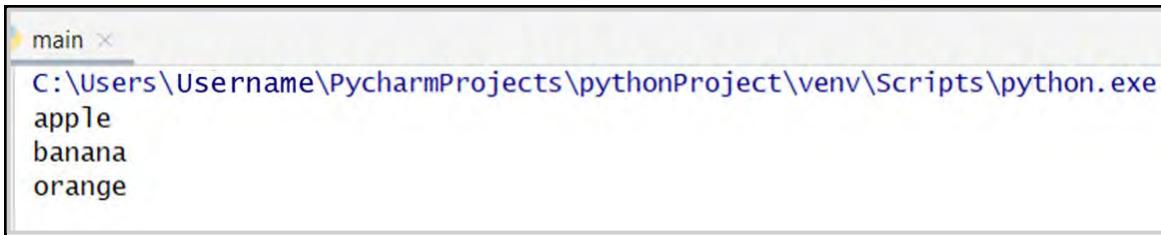
Code Snippet 6 shows an example of `*args`.

Code Snippet 6:

```
def print_arguments(*args):  
    for arg in args:  
        print(arg)  
  
print_arguments( "apple" , "banana" , "orange" )
```

In Code Snippet 6, the `print_arguments` function is showcased, employing `*args` to accept an arbitrary number of positional arguments. This example displays how functions can handle varying numbers of inputs, enhancing their versatility.

Figure 4.1 depicts the output of this code when executed through PyCharm.



```
main
C:\Users\Username\PycharmProjects\pythonProject\venv\Scripts\python.exe
apple
banana
orange
```

Figure 4.1: Output of Code Snippet 6

Arbitrary Keyword Arguments (**kwargs)

The `*kwargs` syntax permits a function to receive a limitless quantity of keyword arguments. The arguments are gathered into a dictionary, where the keys are the argument names and the values are the provided values.

Code Snippet 7 shows an example of `**kwargs`.

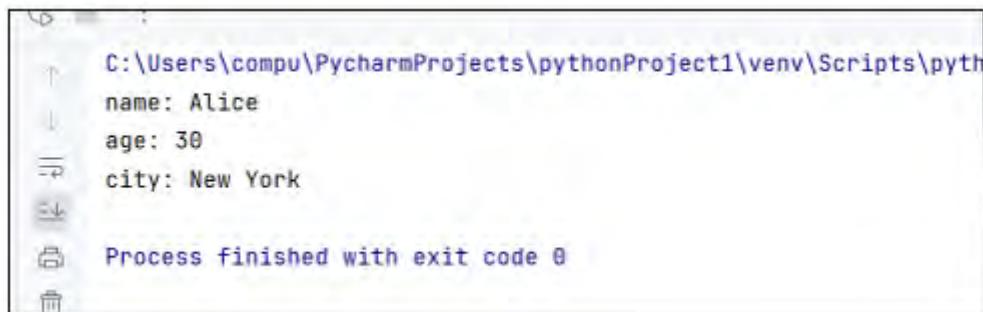
Code Snippet 7:

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

print_keyword_arguments(name="Alice", age=30, city="New
York")
```

In Code Snippet 7, `print_keyword_arguments` is used to illustrate the handling of an arbitrary number of keyword arguments with `**kwargs`. This snippet shows the adaptability of functions to receive and process inputs in a key-value format.

Figure 4.2 depicts the output of this code.



```
C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe
name: Alice
age: 30
city: New York
Process finished with exit code 0
```

Figure 4.2: Output of Code Snippet 7

Using both *args and **kwargs

You can use both *args and **kwargs in the same function definition to accept both positional and keyword arguments.

Code Snippet 8 shows an example of using both *args and **kwargs.

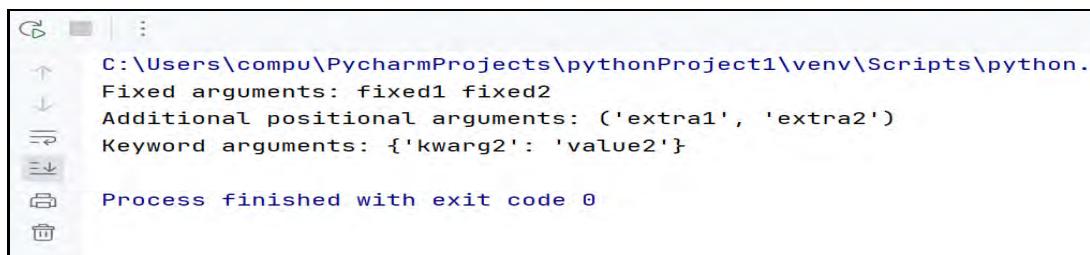
Code Snippet 8:

```
def combined_arguments(arg1, arg2, *args, kwarg1=None,
                      **kwargs):
    print("Fixed arguments:", arg1, arg2)
    print("Additional positional arguments:", args)
    print("Keyword arguments:", kwargs)

combined_arguments("fixed1", "fixed2", "extra1", "extra2",
                    kwarg1="value1", kwarg2="value2")
```

In Code Snippet 8, a function that combines both `*args` and `**kwargs` is presented, demonstrating how functions can be flexible enough to accept both types of arbitrary arguments simultaneously.

Figure 4.3 depicts the output of this code.



The screenshot shows the PyCharm terminal window. The command run was `python combined_arguments.py`. The output is as follows:

```
C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.
Fixed arguments: fixed1 fixed2
Additional positional arguments: ('extra1', 'extra2')
Keyword arguments: {'kwarg2': 'value2'}
Process finished with exit code 0
```

Figure 4.3: Output of Code Snippet 8

4.3 Parameters in Python

In Python, parameters are variables defined in a function declaration to receive and process the values passed to the function when it is called. Parameters enhance the versatility of functions and enable them to handle various inputs.

There are two main types of parameters in Python functions:

Positional Parameters

These are the most common type of parameters. They are defined in the function declaration and their values are assigned based on the order in which the arguments are passed when the function is called.

Syntax of positional parameters in Python:

```
def example_function(parameter1, parameter2):
    # Function code here

# Calling the function with positional arguments
example_function(value1, value2)
```

Keyword Parameters

Keyword parameters (also known as named parameters) are specified using the parameter names as keywords when calling the function. This allows you to pass values to specific parameters out of order.

Syntax of keyword parameters in Python:

```
def example_function(parameter1, parameter2):
    # Function code here

# Calling the function using keyword arguments
example_function(parameter1=value1, parameter2=value2))
```

Functions are flexible and can have any quantity of parameters, including zero. Additionally, default values can be employed for parameters, rendering them as optional when the function is called. If a default value is provided, the parameter becomes a keyword parameter, and it can be omitted when calling the function.

Code Snippet 9 shows the usage of greeting parameters in Python.

Code Snippet 9:

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

# Calling the function with and without the '#greeting'
# parameter
greet("Alice")  # Output: Hello, Alice!
greet("Bob", "Hi")  # Output: Hi, Bob!
```

In this example, the greeting parameter has a default value of Hello, making it optional when calling the function. If no value is provided for greeting, the default value is used.

4.4 User-defined Functions in Python

User-defined functions in Python are blocks of reusable code that you create to perform specific tasks. These functions are defined by the developer to encapsulate a sequence of operations into a single named unit. They enhance code organization, readability, and reusability.

Syntax of defining a user-defined function in Python:

```
def function_name(parameters):
    # Function body
    # Code to perform specific tasks
    return result # Optional return statement
```

The components of a user-defined function are as follows:

01

def:

This keyword is used to declare the start of a function definition.

02

function_name:

Choose a meaningful name for your function. It should follow the same naming conventions as variable names.

03

Parameters:

These are input values that the function accepts. They are optional, but if present, they are enclosed within parentheses. Multiple parameters are separated by commas.

04

Function Body:

This is where you write the code that performs the intended task.

05

return Statement:

It is optional, but if used, it allows the function to return a value as the result of its computation.

Code Snippet 10 shows an example of a simple user-defined function that calculates the square of a number.

Code Snippet 10:

```
def calculate_square(number):
    square = number ** 2
    return square

result = calculate_square(5)
print(result)
# Output: 25
```

In Code Snippet 10, a function named `calculate_square` is defined to compute the square of a number. This snippet emphasizes the basic mathematical operations and return statements in functions, showcasing how functions can process and return results.

In this example, the function `calculate_square` takes a single parameter (`number`) and calculates its square, which is then returned.

Lambda functions, also known as anonymous functions, are a concise way to create small, simple functions in Python. Unlike regular functions defined using the `def` keyword, `lambda` functions are defined using the `lambda` keyword. They are suitable for short tasks where a full function definition might be excessive.

Syntax of a `lambda` function:

```
lambda arguments: expression
```

Code Snippet 11 shows an example of a `lambda` function that calculates the square of a number.

Code Snippet 11:

```
square = lambda x: x ** 2
result = square(5)
print(result)
# Output: 25
```

In Code Snippet 11, a lambda function for squaring a number is shown, highlighting the concise syntax for writing anonymous functions that perform simple operations in a single line.

Lambda functions are commonly used when you require a quick function for a short operation, such as sorting, filtering, or mapping. They are often used as arguments to higher-order functions such as map, filter, and sorted.

For instance, using a lambda function with the sorted function to sort a list of tuples based on the second element.

Code Snippet 12 shows an example of a lambda function to sort a list of tuples based on the second element.

Code Snippet 12:

```
points = [(1, 3), (2, 1), (5, 2)]
sorted_points = sorted(points, key=lambda point: point[1])
print(sorted_points)
# Output: [(2, 1), (5, 2), (1, 3)]
```

In Code Snippet 12, it sorts a list of tuples, `points`, based on the second element of each tuple. The `sorted` function is used with a `lambda` function as the key, which specifies that the sorting should be done according to the second item (`point[1]`) in each tuple. The sorted list is stored in `sorted_points` and then printed, resulting in the list being sorted in ascending order by the second element of the tuples.

Lambda functions are limited in scope and are best suited for simple operations. For more complex functions, it is advisable to use regular named functions defined using the `def` keyword.

4.5 Callable in Python

In Python, the term `callable` refers to objects that can be called as functions. It encompasses various entities, including functions, methods, classes, and objects with a `__call__` method defined. Essentially, any object that can be invoked using parentheses, such as a function call, is considered callable.

Main categories of **callables** in Python are follows:

01

Functions

Regular functions created using the `def` keyword are callable. They can be defined with parameters and a body of code to execute.

02

Methods

Methods are functions associated with objects, typically defined within classes. They are callable when invoked on instances of the class.

03

Lambda Function

Lambda functions, also known as anonymous functions, are short, one-liner functions defined using the `lambda` keyword. They are callable as well.

04

Classes

Classes themselves can be callable if they define a `__call__` method. When an instance of the class is called, the `__call__` method is executed.

05

Functions

Any object that defines a `__call__` method can be invoked as if it were a function. This can be useful for creating custom callable objects.

Function

```
def my_function(x):
    return x + 1
result = my_function(3) # Calling the function
```

Method

```
class MyClass:
    def my_method(self, x):
        return x + 1

obj = MyClass()
result = obj.my_method(3) # Calling the method
```

Lambda Function

```
square = lambda x: x ** 2
result = square(5) # Calling the lambda function
```

Class with `__call__` Method

```
class CallableClass:
    def __call__(self, x):
        return x + 1

obj = CallableClass()
result = obj(3) # Calling the object as if it is a function
```

Callable Object with `__call__` Method

```
class:
    def __call__(self, x):
        return x + 1

callable_obj = CustomCallable()
result = callable_obj(3) # Calling the callable object
```

4.6 Other Functions in Python

In addition to the basic types of functions, other important types are as follows:

Generator Functions

These functions use the `yield` keyword to produce values one at a time, allowing for memory-efficient iteration over large data sets or infinite sequences.

Code Snippet 13 shows how to define a generator function in Python.

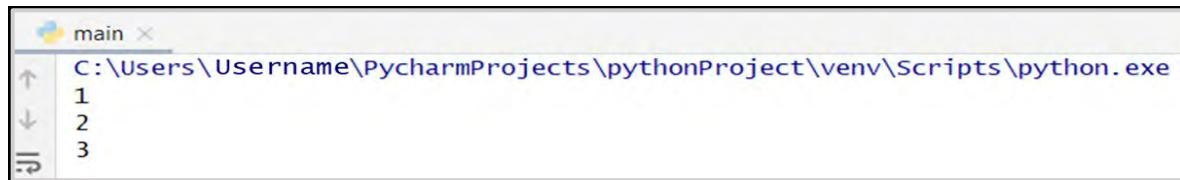
Code Snippet 13:

```
def my_generator():
    yield 1
    yield 2
    yield 3

# Using the generator
gen = my_generator()
for value in gen:
    print(value)
```

In Code Snippet 13, a generator function named `my_generator` is illustrated, using `yield` to produce a sequence of values. This example introduces the concept of generator functions for efficient iteration over sequences.

Figure 4.4 depicts the output of this code when executed through PyCharm.

A screenshot of the PyCharm IDE. The title bar says "main". The code editor shows the path "C:\Users\Username\PycharmProjects\pythonProject\venv\Scripts\python.exe". The code itself is a generator function definition:

```
def my_generator():
    yield 1
    yield 2
    yield 3
```

The output window below the editor shows the numbers 1, 2, and 3, each on a new line, indicating the sequence produced by the generator.

Figure 4.4: Output of Code Snippet 13

Decorator Functions

Decorators modify or enhance the behavior of other functions or methods. They are applied using the @ symbol and are used for tasks such as logging, timing, and access control.

Code Snippet 14 shows how to define a decorator function in Python.

Code Snippet 14:

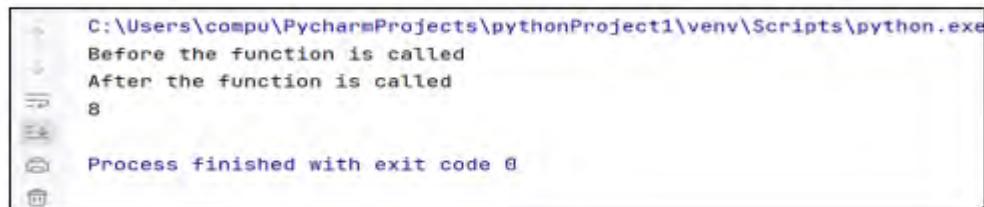
```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Before the function is called")
        result = func(*args, **kwargs)
        print("After the function is called")
        return result
    return wrapper

@my_decorator
def my_function(x, y):
    return x + y

result = my_function(5, 3)
print(result)
```

In Code Snippet 14, a decorator function `my_decorator` is applied to `my_function`, demonstrating how decorators can modify the behavior of functions. This snippet showcases the use of decorators for adding functionality before and after the decorated function's execution.

Figure 4.5 depicts the output of this code.



```
C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe
Before the function is called
After the function is called
8

Process finished with exit code 0
```

Figure 4.5: Output of Code Snippet 14

Special Functions (Magic Methods)

These functions have double underscores on both sides of their names, such as `__init__`, `__str__`, `__len__`, and so on. They allow customization of object behavior during specific operations.

Higher-Order Functions

These are functions that take one or more functions as arguments or return functions as results. They are essential for functional programming techniques.

Recursive Functions

Recursive functions call themselves to solve problems that can be broken down into smaller instances of the same problem. Examples include the calculation of factorial or fibonacci numbers.

Syntax of recursive functions in Python:

```
def recursive_function(parameters):
    # Base case: The condition that halts the recursion
    if base_case_condition:
        return base_case_value

    # Recursive case: call the function with smaller
    # subproblem
    else:
        smaller_problem = modify_parameters(parameters)
        return recursive_function(smaller_problem)
```

Code Snippet 14 shows example of recursive function in Python.

Code Snippet 15:

```
def factorial(n):
    # Base case
    if n == 0:
```

```

        return 1
    # Recursive case
    else:
        return n * factorial(n - 1)

result = factorial(5)
print(result)

# Output: 120

```

In Code Snippet 15, a recursive function for calculating factorial is provided, exemplifying how functions can call themselves to solve problems that can be divided into smaller, similar problems.

Closures

A closure is a nested function that captures and remembers the values in its enclosing function scope even after the outer function has finished executing.

Partial Functions

Partial functions allow the setting of specific arguments for a function, resulting in the creation of a new function with those values already predefined. This is useful for creating specialized functions from more general ones. It is created using `functools.partial`.

Code Snippet 16 shows an example to create partial function in Python.

Code Snippet 16:

```

from functools import partial

# Original function
def power(x, y):
    return x ** y

# Creating a partial function
square = partial(power, y=2)
cube = partial(power, y=3)

print(square(4))  # Output: 16 (4^2)
print(cube(3))   # Output: 27 (3^3)

```

In Code Snippet 16, partial functions are created from a base `power` function using `functools.partial`, illustrating how to pre-specify some arguments of a function to create a new function.

Async Functions

Used for asynchronous programming, these functions allow tasks to run concurrently without waiting for each to complete before moving on to the next task.

Code Snippet 17 shows an example to define and use `async` function in Python.

Code Snippet 17:

```
import asyncio

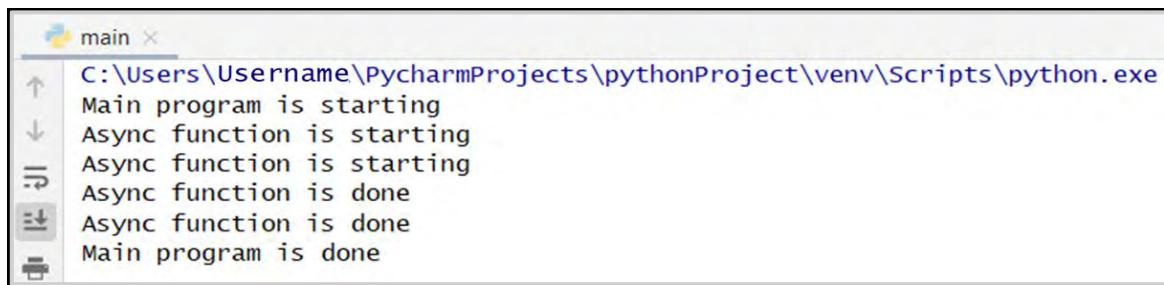
async def async_function():
    print("Async function is starting")
    await asyncio.sleep(2)
    print("Async function is done")

async def main():
    print("Main program is starting")
    await asyncio.gather(async_function(), async_function())
    print("Main program is done")

# Run the async program
asyncio.run(main())
```

In Code Snippet 17, an asynchronous function example is given, showing how to define and use `async` functions for concurrent execution without blocking.

Figure 4.6 depicts the output of this code when executed through PyCharm.



The screenshot shows the PyCharm terminal window titled "main". The terminal displays the following output:

```
C:\Users\Username\PycharmProjects\pythonProject\venv\Scripts\python.exe
Main program is starting
Async function is starting
Async function is starting
Async function is done
Async function is done
Main program is done
```

Figure 4.6: Output of Code Snippet 17

Namespace Functions

Functions associated with namespaces, such as `globals()` and `locals()`, provide access to the dictionaries of the global or local namespace.

Error Handling Functions

Functions such as `try`, `except`, and `finally` are used for handling exceptions and ensuring proper cleanup in case of errors.

Code Snippet 18 shows an example of error handling function in Python.

Code Snippet 18:

```
def divide(x, y):
    try:
        result = x / y

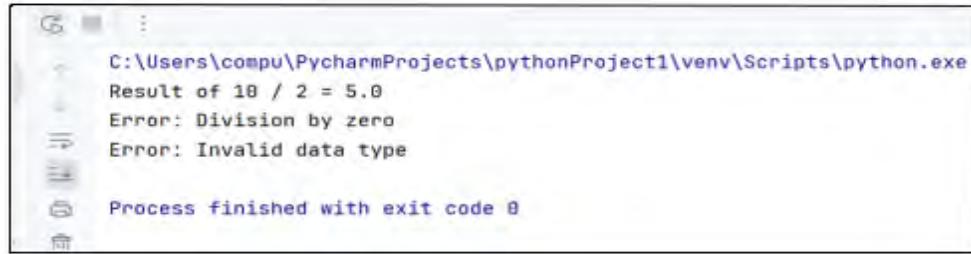
    except ZeroDivisionError:
        print("Error: Division by zero")
        return None
    except TypeError:
        print("Error: Invalid data type")
        return None
    except Exception as e:
        print("An unexpected error occurred:", e)
        return None
    else:
        return result

numerator = 10

# Test cases
denominators = [2, 0, "abc"]
for denominator in denominators:
    result = divide(numerator, denominator)
    if result is not None:
        print(f"Result of {numerator} / {denominator} = {result}")
```

In Code Snippet 18, an error-handling function named `divide` is displayed, using try-except blocks to gracefully handle different types of errors that may occur during division.

Figure 4.7 depicts the output of this code.



```
C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe
Result of 10 / 2 = 5.0
Error: Division by zero
Error: Invalid data type

Process finished with exit code 0
```

Figure 4.7: Output of Code Snippet 18

In this example, the divide function attempts to perform division and handles various exceptions using the `try`, `except`, and `else` blocks:

- i. If a `ZeroDivisionError` occurs, it prints an error message.
- ii. If a `TypeError` occurs (due to non-numeric input), it prints an error message.
- iii. If any other exception occurs, it prints a generic error message.
- iv. If no exception occurs, the division result is returned.

Code Snippet 19 shows an example for calculating the area of a rectangle using length and width.

Code Snippet 19:

```
# Define a function to calculate the area of a
#rectangle
def calculate_rectangle_area(length, width):
    area = length * width
    return area

# Input from the user
length = float(input("Enter the length of the
rectangle: "))
width = float(input("Enter the width of the
rectangle: "))

# Call the function to calculate the area
area = calculate_rectangle_area(length, width)

# Display the result
print(f"The area of the rectangle with length
{length} and width {width} is {area}")
```

In Code Snippet 19, a function to calculate the area of a rectangle is shown, demonstrating basic arithmetic operations and interaction with user input.

Figure 4.8 depicts the output of this code when executed through PyCharm.

```
main
C:\Users\Username\PycharmProjects\pythonProject\venv\Scripts\python.exe
Enter the length of the rectangle: 15
Enter the width of the rectangle: 8
The area of the rectangle with length 15.0 and width 8.0 is 120.0
Process finished with exit code 0
```

Figure 4.8: Output of Code Snippet 19

Code Snippet 20 shows an example for calculating the volume of a rectangular prism based on its length, width, and height.

Code Snippet 20:

```
# Define a function to calculate the volume of a
#rectangular prism
def calculate_volume(length, width, height=1):
    volume = length * width * height
    return volume

# Input from the user
length = float(input("Enter the length of the
rectangular prism: "))
width = float(input("Enter the width of the
rectangular prism: "))
height = float(input("Enter the height of the
rectangular prism (optional, press Enter to use
default value of 1): ") or 1)

# Call the function with positional parameters
volume_pos = calculate_volume(length, width, height)

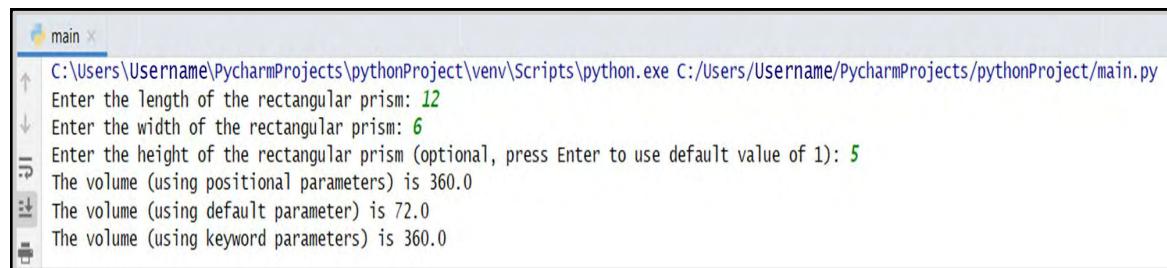
# Call the function with default parameter for
height
volume_default = calculate_volume(length, width)
```

```
# Call the function with keyword parameters
volume_keyword = calculate_volume(length=length,
width=width, height=height)

# Display the results
print(f"The volume (using positional parameters) is
{volume_pos}")
print(f"The volume (using default parameter) is
{volume_default}")
print(f"The volume (using keyword parameters) is
{volume_keyword}")
```

In Code Snippet 20, a function for calculating the volume of a rectangular prism is illustrated, showcasing default parameter values and the use of both positional and keyword arguments for flexible function calls.

Figure 4.9 depicts the output of this code when executed through PyCharm.



```
C:\Users\Username\PycharmProjects\pythonProject\venv\Scripts\python.exe C:/Users/Username/PycharmProjects/pythonProject/main.py
Enter the length of the rectangular prism: 12
Enter the width of the rectangular prism: 6
Enter the height of the rectangular prism (optional, press Enter to use default value of 1): 5
The volume (using positional parameters) is 360.0
The volume (using default parameter) is 72.0
The volume (using keyword parameters) is 360.0
```

Figure 4.9: Output of Code Snippet 20

4.7 Summary

- Functions in Python are reusable blocks of code defined using the `def` keyword.
- In Python, arguments are values passed to functions when they are called.
- In Python, parameters are placeholders defined in a function declaration.
- User-defined functions in Python are custom-created blocks of code that perform specific tasks, promoting modularity and code reusability.
- Lambda functions in Python are concise, anonymous functions created using the `lambda` keyword.
- Callables in Python refer to entities that can be invoked or called such as functions, including functions, methods, and objects with `__call__` methods.
- Other functions in Python encompass various specialized types, such as generator functions, decorator functions, and magic methods.

4.8 Test Your Knowledge

1. What is a function in Python?
 - A) A reserved keyword
 - B) A block of reusable code that performs a specific task
 - C) A built-in module
 - D) An operator for mathematical operations

2. Which keyword is used to define a function in Python?
 - A) `function`
 - B) `def`
 - C) `Define`
 - D) `func`

3. What is the purpose of parameters in a function?
 - A) They provide the function name
 - B) They store the function result
 - C) They hold the data used by the function
 - D) They determine the function return type

4. What is the role of the `return` statement in a function?
 - A) It defines the function name
 - B) It specifies the parameters of the function
 - C) It indicates the value that the function will output
 - D) It handles exceptions in the function

5. Which of the following is true about default parameter values?

- A) They are not allowed in Python functions
- B) They must be assigned to a value outside the function
- C) They provide values to parameters if no argument is provided
- D) They are used only for complex data types

6. What does the term 'higher-order function' refer to in Python?

- A) A function that returns multiple values
- B) A function that works with higher numbers only
- C) A function that takes other functions as arguments or returns them
- D) A function that is defined within another function

4.8.1 Answers to Test Your Knowledge

1. A block of reusable code that performs a specific task
2. `def`
3. They hold the data used by the function
4. It indicates the value that the function will output
5. They provide values to parameters if no argument is provided
6. A function that takes other functions as arguments or returns them

Try It Yourself

1. Write a Python program to calculate the factorial of a given number using a user-defined function.
2. Write a Python program on a function that takes a list of numbers as input and returns the sum of all the numbers.
3. Write a Python program to calculate the factorial of a given number using recursion.
4. Write a Python program on a function that takes a list of strings and returns a new list with all strings capitalized.



SESSION 05

LIST AND TUPLES IN PYTHON

Learning Objectives

In this session, students will learn to:

- ◆ Outline the concept of List in Python
- ◆ Explain the creation and working of Lists in Python
- ◆ Describe the process of receiving input and adding elements
- ◆ Define the process of concatenating and slicing Tuples in Python

5.1 Introduction to List in Python

In Python, a list is a commonly used data structure that allows you to store a collection of items. These items can be of any data type, including integers, strings, floating-point numbers, other lists, and even more complex objects. Lists are mutable, which means you can modify their contents by adding, removing, or updating elements after they are created.

5.1.1 Creating a List in Python

Lists in Python are created by enclosing items within square brackets [], with commas used to separate individual elements inside the list.

Code Snippet 1 shows an example of creating a list.

Code Snippet 1:

```
# Creating a list of integers
my_integer_list = [1, 2, 3, 4, 5]
print("My Integer List:", my_integer_list)

# Creating a list of strings
my_string_list = ["apple", "banana", "cherry"]
print("My String List:", my_string_list)

# Forming a list that contains a combination of different data
# types.
mixed_list = [10, "hello", 3.14, True]
print("Mixed List:", mixed_list)

# Creating an empty list
empty_list = []
print("Empty List:", empty_list)
```

In Code Snippet 1, various types of lists in Python are created, demonstrating the creation of lists with integers, strings, mixed data types, and an empty list. This snippet showcases the versatility of Python lists in storing different types of data.

Figure 5.1 depicts the output of this code when executed through PyCharm.

```
C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:\Us
My Integer List: [1, 2, 3, 4, 5]
My String List: ['apple', 'banana', 'cherry']
Mixed List: [10, 'hello', 3.14, True]
Empty List: []

Process finished with exit code 0
```

Figure 5.1: Output of Code Snippet 1

Code Snippet 2 shows a Python program to create an empty list.

Code Snippet 2:

```
def main():
    # Using square brackets
    list1 = [1, 2, 3, 4, 5]

    # Using the list() constructor
    list2 = list([6, 7, 8, 9, 10])
    # List comprehension
    list3 = [x * 2 for x in range(5)]

    # Creating an empty list
    empty_list = []

    # Using the append() method
    append_list = []
    append_list.append(1)
    append_list.append(2)
    append_list.append(3)

    # Using the extend() method
    extend_list = [1, 2, 3]
    extend_list.extend([4, 5, 6])

    # Using repetition operator
    repeated_list = [0] * 5
```

```
# Using list slicing
original_list = [1, 2, 3, 4, 5]
sliced_list = original_list[1:4]

# Using the list() function with other iterables
string = "hello"
char_list = list(string)

# Print the created lists
print("list1:", list1)
print("list2:", list2)
print("list3:", list3)
print("empty_list:", empty_list)
print("append_list:", append_list)
print("extend_list:", extend_list)
print("repeated_list:", repeated_list)
print("sliced_list:", sliced_list)
print("char_list:", char_list)

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

In Code Snippet 2, different methods to create and manipulate lists are explored, including using the list constructor, list comprehension, the append and extend methods, repetition operator, slicing, and converting a string to a list. This snippet provides a comprehensive overview of list operations in Python.

Figure 5.2 depicts the output of this code when executed through PyCharm.

```
C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:/Users/compu/PycharmProjects/pythonProject1/list_exercise.py
list1: [1, 2, 3, 4, 5]
list2: [6, 7, 8, 9, 10]
list3: [0, 2, 4, 6, 8]
empty_list: []
append_list: [1, 2, 3]
extend_list: [1, 2, 3, 4, 5, 6]
repeated_list: [0, 0, 0, 0, 0]
sliced_list: [2, 3, 4]
char_list: ['h', 'e', 'l', 'l', 'o']

Process finished with exit code 0
```

Figure 5.2: Output of Code Snippet 2

5.1.2 Accessing Elements from a List

Elements in a Python list can be accessed using indexing. Indexing begins at 0 for the initial element and both positive and negative indices can be employed to retrieve elements from the list's start and end, respectively.

Code Snippet 3 shows an example of accessing elements from a list.

Code Snippet 3:

```
my_list = [10, 20, 30, 40, 50]

# Accessing elements using positive indices
first_element = my_list[0]    # Accesses the first #element
(10)
second_element = my_list[1]   # Accesses the second #element
(20)

# Accessing elements using negative indices
last_element = my_list[-1]    # Accesses the last #element
(50)
second_to_last = my_list[-2]  # Accesses the second-to-last
element (40)

# Utilizing slicing to retrieve a specified range of #
elements.
```

```
sliced_elements = my_list[1:4] # Accesses elements # from  
index 1 to 3 ([20, 30, 40])  
  
print(first_element)  
print(second_element)  
print(last_element)  
print(second_to_last)  
print(sliced_elements)
```

In Code Snippet 3, accessing elements from a list using positive and negative indices, along with slicing to retrieve a range of elements, is demonstrated. This illustrates how to access and manipulate list items.

Figure 5.3 depicts the output of this code when executed through PyCharm.

```
C:\Users\Username\PycharmProjects\pythonProject\venv\Scripts\python.exe  
10  
20  
50  
40  
[20, 30, 40]
```

Figure 5.3: Output of Code Snippet 3

5.1.3 Negative Indexing

Negative indexing in Python permits access to elements from the end of a list or other sequences. The index -1 corresponds to the last element, -2 to the second-to-last element, and so on. Negative indices can be very useful for quickly accessing elements from the end of a sequence without requiring to know its length.

Code Snippet 4 shows an example of negative indexing.

Code Snippet 4:

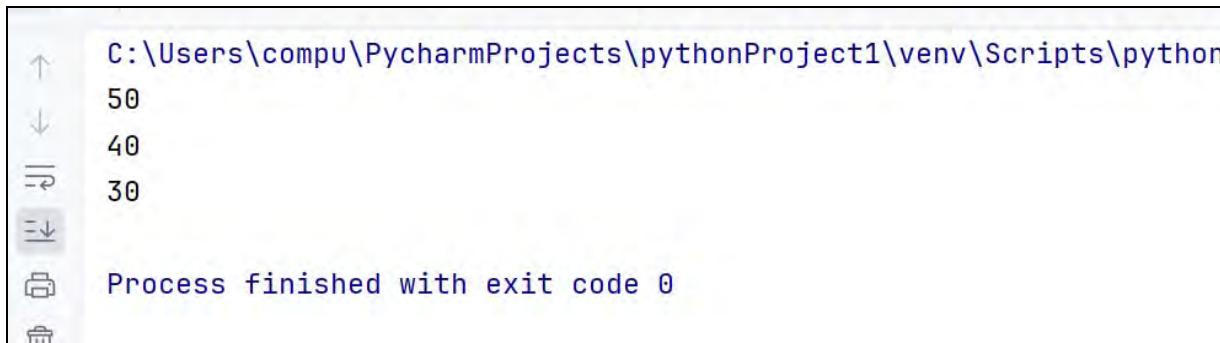
```
my_list = [10, 20, 30, 40, 50]
```

```
last_element = my_list[-1]
# Accesses the last element (50)
second_to_last = my_list[-2]
# Accesses the second-to-last element (40)
third_from_end = my_list[-3]
# Accesses the third-to-last element (30)

print(last_element)
print(second_to_last)
print(third_from_end)
```

In Code Snippet 4, negative indexing to access elements from the end of a list is shown. This method allows easy retrieval of elements without needing the list's length.

Figure 5.4 depicts the output of this code when executed through PyCharm.



The screenshot shows the PyCharm terminal window. It displays the following text:
C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python
50
40
30
Process finished with exit code 0

Figure 5.4: Output of Code Snippet 4

5.1.4 Getting the Size of Python List

The size or the number of elements in a Python list, can be obtained using the built-in `len()` function. The `len()` function returns the length of the list, which is the count of elements it contains.

Code Snippet 5 shows an example of the `len()` function in Python.

Code Snippet 5:

```
my_list = [10, 20, 30, 40, 50]

list_size = len(my_list)
```

```
print("Size of the list:", list_size)
# Output: Size of the list: 5
```

In Code Snippet 5, the `len()` function is used to get the size of a list, highlighting how to determine the number of elements in a list.

5.1.5 Taking Input of a Python List

To input a Python list, use the `input()` function to receive a comma-separated string of values. Then, convert this string into a list using the `split()` method.

Code Snippet 6 shows an example to take input from the programmer.

Code Snippet 6:

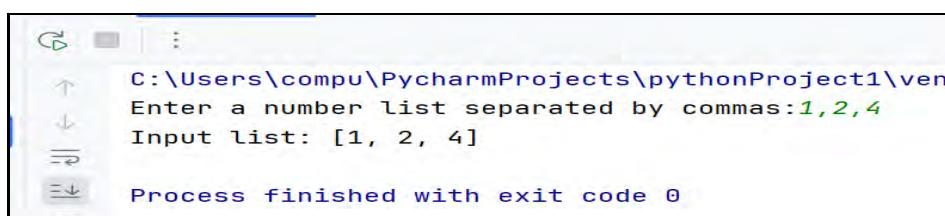
```
# Taking input from the user as a comma-separated #string
input_string = input("Enter a number list separated by
commas:")

# Splitting the input string and converting it into #a list
# of integers
input_list = [int(x) for x in input_string.split(',')]

print("Input list:", input_list)
```

In Code Snippet 6, input from a user is taken and converted into a list of integers, demonstrating dynamic list creation based on user input.

Figure 5.5 depicts the output of this code when executed through PyCharm.



```
C:\Users\compu\PycharmProjects\pythonProject1\venv
Enter a number list separated by commas:1,2,4
Input list: [1, 2, 4]
Process finished with exit code 0
```

Figure 5.5: Output of Code Snippet 6

5.1.6 Adding Elements to Python List

Elements can be added to a Python list using several methods.

Three most commonly used methods are as follows:

1. Using `append()` Method

In Python, the `append()` method is a built-in function for adding elements to a list. This method enables the addition of one element to the end of the list at a time.

The `append()` method is particularly useful for adding elements sequentially, and it is very efficient for this purpose.

```
my_list = [1, 2, 3]
my_list.append(4)
# Adds 4 to the end of the list
print(my_list)
```

2. Using `insert()` Method

This method permits the insertion of an element at a specific index within the list.

```
my_list = [1, 2, 3]
my_list.insert(1, 5)
# Inserts 5 at index 1
print(my_list)
# Output: [1, 5, 2, 3]
```

3. Using `concatenation (+)` Method

Two lists can be concatenated to combine elements from one list with another.

```
my_list = [1, 2, 3]
new_elements = [4, 5]
my_list += new_elements
# Concatenates the two lists
print(my_list)
# Output: [1, 2, 3, 4, 5]
```

Code Snippet 7 shows the program to add elements to Python list.

Code Snippet 7:

```
def main():
    # Initialize an empty list
    my_list = [ ]

    # Take input from the user to create the initial list
    while True:
        element = input("Provide a new element for inclusion
in the list(or 'done' to finish): ")

        if element.lower() == 'done':
            break

        my_list.append(element)

    # Display the initial list
    print("Initial list:", my_list)

    # Allow the user to add more elements to the list
    while True:
        new_element = input("Enter an element to add to the
list (or 'done' to finish): ")

        if new_element.lower() == 'done':
            break

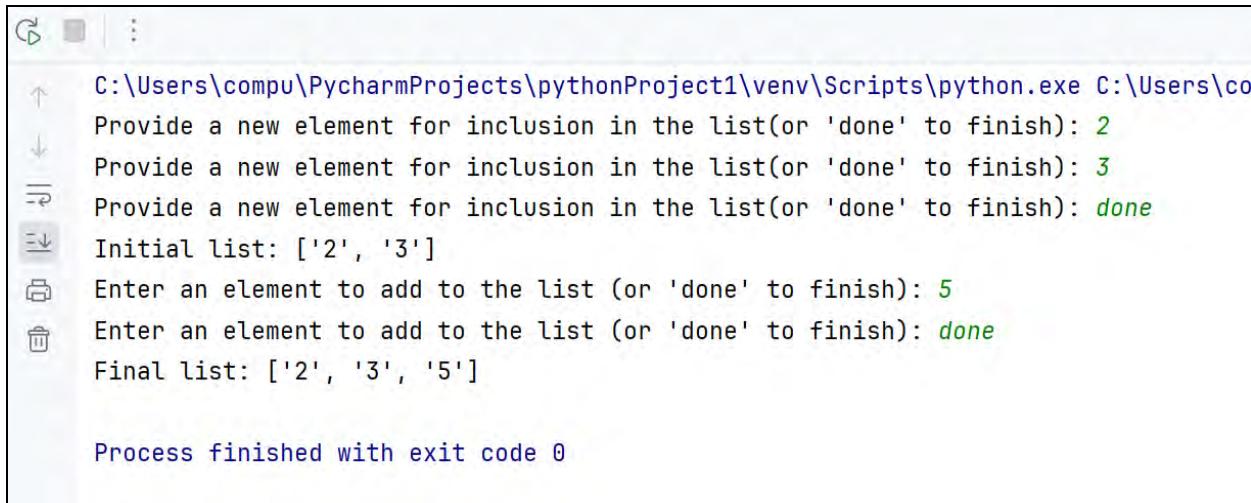
        my_list.append(new_element)

    # Display the final list
    print("Final list:", my_list)

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

In Code Snippet 7, a program for adding elements to a list through user input is shown, detailing the process of dynamically expanding a list based on user-provided values.

Figure 5.6 depicts the output of this code when executed through PyCharm.



```
C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:/Users/compu/PycharmProjects/pythonProject1/main.py
Provide a new element for inclusion in the list(or 'done' to finish): 2
Provide a new element for inclusion in the list(or 'done' to finish): 3
Provide a new element for inclusion in the list(or 'done' to finish): done
Initial list: ['2', '3']
Enter an element to add to the list (or 'done' to finish): 5
Enter an element to add to the list (or 'done' to finish): done
Final list: ['2', '3', '5']

Process finished with exit code 0
```

Figure 5.6: Output of Code Snippet 7

5.1.7 Reversing a List

In Python, list reversal can be achieved through either the `reverse()` method or by utilizing slicing.

These methods are explained as follows:

1. Using `reverse()` Method

The built-in `reverse()` method is designed to invert the order of list elements directly within the original list, resulting in an in-place modification.

```
my_list = [1, 2, 3, 4, 5]
my_list.reverse()
# Reverses the list in
#place
print(my_list)
# Output: [5, 4, 3, 2, 1]
```

2. Using slicing Method

Reversing a list is also possible using slicing. This approach generates a new list that represents the reversed version of the original list.

```
my_list = [1, 2, 3, 4, 5]
reversed_list =
my_list[::-1]
# Slicing with a step of
#-1
print(reversed_list)
# Output: [5, 4, 3, 2, 1]
```

Both methods achieve same result, but one must note that using `reverse()` method modifies the original list, whereas slicing creates a new reversed list without changing original list.

5.1.8 Removing Elements from the List in Python

1. Using `remove()` Method

The `remove()` method is used to remove the first occurrence of a specific value from the list.

```
my_list = [10, 20, 30, 40, 50]
my_list.remove(30)
# Removes the value 30 from the list
print(my_list)
```

2. Using `pop` Method

Reversing a list can be accomplished using slicing. This method creates a new list that is the reversed version of the original list.

```
my_list = [10, 20, 30, 40, 50]
removed_value = my_list.pop(2)
# Removes the element at index 2 (30)
print(removed_value)

# Output: 30
print(my_list)
```

3. Using Del Statement

The del statement can be used to remove elements by specifying the index or a slice.

```
my_list = [10, 20, 30, 40, 50]
del my_list[1]
# Deletes the element at index 1 (20)
print(my_list)
# Output: [10, 30, 40, 50]

del my_list[1:3]
# Removes elements from index 1 to 2 ([30, 40])
print(my_list)
# Output: [10, 50]
```

Lists can be indexed and sliced. Indexing starts from 0, indicating that the first element in the list has an index of 0, with subsequent elements assigned indexes incrementing by 1. Negative indices count from the end of the list, where -1 corresponds to the last element, -2 to the second-to-last, and so forth.

Code Snippet 8 shows the examples of list indexing and slicing.

Code Snippet 8:

```
my_list = [10, 20, 30, 40, 50]

print(my_list[0])
print(my_list[-1])
print(my_list[1:4])
```

In Code Snippet 8, indexing and slicing to access and display specific elements and ranges within a list are illustrated, showcasing basic list manipulation techniques.

Figure 5.7 depicts the output of this code when executed through PyCharm.

```
C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python
10
50
[20, 30, 40]

Process finished with exit code 0
```

Figure 5.7: Output of Code Snippet 8

5.1.9 Slicing of a List

Slicing in Python refers to the technique of extracting a portion of a list by specifying a start index, an end index, and a step size.

Syntax for slicing a list:

```
new_list = original_list[start:end:step]
```

START

start: Specify the index of the initial element you wish to incorporate within the slice.



end: The index just after the last element you want to include in the slice. The slice goes up to, but does not include, this index. In case of exclusion, it automatically assumes the default value of the list length.



step: The interval between elements in the slice. If omitted, it defaults to 1 (sequential elements).

Code Snippet 9:

```
original_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Extract elements from index 2 to index 5 (not including 5)
slice1 = original_list[2:5]
print("Slice 1:", slice1)

# Extract elements from index 3 to the end
slice2 = original_list[3:]
print("Slice 2:", slice2)

# Extract elements from the beginning to index 4 (not including 4)
slice3 = original_list[:4]
```

```

print("Slice 3:", slice3)

# Extract every alternate element, beginning from index 1.
slice4 = original_list[1::2]
print("Slice 4:", slice4)

# Reverse the list using a step of -1
reverse_list = original_list[::-1]
print("Reversed List:", reverse_list)

```

Code Snippet 9 utilizes slicing to extract specific portions from the original list. Slice 1 gathers elements from index 2 to 4, excluding 5. Slice 2 captures elements from index 3 to the end. Slice 3 acquires elements from the start to index 3. Slice 4 retrieves every alternate element from index 1. Slice 5 reverses the entire list.

Figure 5.8 depicts the output of this code when executed through PyCharm.

```

C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe
Slice 1: [2, 3, 4]
Slice 2: [3, 4, 5, 6, 7, 8, 9]
Slice 3: [0, 1, 2, 3]
Slice 4: [1, 3, 5, 7, 9]
Reversed List: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

Process finished with exit code 0

```

Figure 5.8: Output of Code Snippet 9

5.1.10 List Comprehension

Python list comprehension serves the purpose of generating fresh lists from different iterables such as tuples, strings, arrays, and existing lists. Within the confines of a list comprehension, one finds brackets encompassing an expression that gets executed for each element. This occurs in conjunction with a `for` loop, which facilitates iteration over each individual element.

Syntax of list comprehension:

```

newList = [ expression(element) for element in oldList if
condition ]

```

5.1.11 List Methods

Python lists are versatile data structures equipped with a range of built-in methods that facilitate the execution of common list operations.

Following are several frequently utilized list methods:

append(item): Adds the specified item to the end of the list.

Example:

```
my_list = [1, 2, 3]
my_list.append(4)
# my_list is now [1, 2, 3, 4]
```

extend(iterable): Appends the elements from an iterable (such as a list, tuple, and so on.) to the conclusion of the list.

Example:

```
my_list = [1, 2, 3]
my_list.extend([4, 5])
# my_list is now [1, 2, 3, 4, 5]
```

insert(index, item): Inserts the specified item at the given index.

Example:

```
my_list = [1, 2, 3]
my_list.insert(1, 4)
# my_list is now [1, 4, 2, 3]
```

remove(item): Removes the first occurrence of the specified item from the list.

Example:

```
my_list = [1, 2, 3, 2]
my_list.remove(2)
# my_list is now [1, 3, 2]
```

pop(index): Removes and retrieves the element at the designated index. If no index is provided, it removes and returns the last element.

Example:

```
my_list = [1, 2, 3]
removed_item = my_list.pop(1)
# my_list is now [1, 3], and removed_item is 2
```

index(item, start, end): Returns the index of the first occurrence of the specified item within the specified range.

Example:

```
my_list = [1, 2, 3, 2]
index = my_list.index(2)
# index is 1
```

count(item): Returns the number of occurrences of the specified item in the list.

Example:

```
my_list = [1, 2, 3, 2]
count = my_list.count(2)
# count is 2
```

sort(key=None, reverse=False): Arranges the list elements in ascending order. The key and reverse parameters can be used to customize the sorting behavior.

Example:

```
my_list = [3, 1, 4, 1, 5, 9, 2, 6]
my_list.sort()
# my_list is now [1, 1, 2, 3, 4, 5, 6, 9]
```

reverse(): Reverses the arrangement of elements within the list.

Example:

```
my_list = [1, 2, 3, 4]
my_list.reverse()
# my_list is now [4, 3, 2, 1]
```

clear(): Removes all elements from the list, making it empty.

Example:

```
my_list = [1, 2, 3]
```

```
my_list.clear()
# my_list is now [ ]
```

5.1.12 List Functions

In addition to the list methods, Python provides a set of built-in functions that can be used with lists to perform various operations.

Table 5.1 describes some commonly used built-in functions for working with lists.

Function	Description	Example
<code>len(list)</code>	Returns the count of elements present in the list.	<code>my_list = [1, 2, 3, 4, 5] length = len(my_list) # length is 5</code>
<code>min(list)</code>	Returns the smallest value among the list elements.	<code>my_list = [5, 2, 9, 1, 7] minimum = min(my_list) # minimum is 1</code>
<code>max(list)</code>	Returns the largest value among the list elements.	<code>my_list = [5, 2, 9, 1, 7] maximum = max(my_list) # maximum is 9</code>
<code>sum(list)</code>	Returns the sum of all elements in the list (works for numeric lists).	<code>my_list = [1, 2, 3, 4, 5] total_sum = sum(my_list) # total_sum is 15</code>
<code>sorted(iterable, key=None, reverse=False)</code>	Returns a new sorted list from the elements of the provided iterable.	<code>my_list = [3, 1, 4, 1, 5, 9, 2, 6] sorted_list = sorted(my_list) # sorted_list is [1, 1, 2, 3, 4, 5, 6, 9]</code>
<code>any(iterable)</code>	Returns True if there is at least one True element within the iterable. For lists, this checks if any element evaluates to True.	<code>my_list = [False, True, False] result = any(my_list) # result is True</code>
<code>all(iterable)</code>	Returns True if all elements in the	<code>my_list = [True, True, False]</code>

Function	Description	Example
	iterable are True. For lists, this checks if all elements evaluate to True.	<pre>result = all(my_list) # result is False</pre>
<code>enumerate(iterable, start=0)</code>	Returns an iterator that yields pairs of index and value for each element in the iterable.	<pre>my_list = ['a', 'b', 'c'] for index, value in enumerate(my_list): print(f"Index {index}: {value}") # Output: # Index 0: a # Index 1: b # Index 2: c</pre>
<code>zip(iterable1, iterable2, ...)</code>	Returns an iterator that aggregates elements from multiple iterables into tuples.	<pre>names = ['Alice', 'Bob', 'Charlie'] scores = [85, 92, 78] for name, score in zip(names, scores): print(f"{name}: {score}") # Output: # Alice: 85 # Bob: 92 # Charlie: 78</pre>
<code>reversed(seq)</code>	Returns a reversed iterator over the elements of the sequence.	<pre>my_list = [1, 2, 3, 4] reversed_list = list(reversed(my_list)) # reversed_list is [4, 3, 2, 1]</pre>

Table 5.1: Functions Used with Lists

Code Snippet 10 shows the Python program for maintaining movies data using list.

Code Snippet 10:

```
def main():
    movies = [] # Initialize an empty list to store #movie
    names
```

```
while True:
    print("\nMovie List Management")
    print("1. Add Movie")
    print("2. View Movies")
    print("3. Quit")
    choice = input("Enter your choice: ")
    if choice == "1":
        movie_name = input("Enter the movie name: ")
        movies.append(movie_name)
        print(f"Movie '{movie_name}' added
              successfully.")
    elif choice == "2":
        print("\nList of Movies:")
        for index, movie in enumerate(movies,
                                       start=1):
            print(f"{index}. {movie}")
    elif choice == "3":
        print("Exiting the program.")
        break
    else:
        print("Invalid choice. Please select a valid
option.")
if __name__ == "__main__":
    main()
```

In Code Snippet 10, a program for managing a movie list demonstrates adding and viewing movies through a simple menu-driven interface, illustrating list usage in practical applications.

Figure 5.9 depicts the output of this code when executed through PyCharm.

```
C:\Users\Username\PycharmProjects\pythonProject\venv

Movie List Management
1. Add Movie
2. View Movies
3. Quit
Enter your choice: 1
Enter the movie name: Mission Impossible
Movie 'Mission Impossible' added successfully.

Movie List Management
1. Add Movie
2. View Movies
3. Quit
Enter your choice: 1
Enter the movie name: Interstellar
Movie 'Interstellar' added successfully.

Movie List Management
1. Add Movie
2. View Movies
3. Quit
Enter your choice: 2

List of Movies:
1. Mission Impossible
2. Interstellar
```

Figure 5.9: Output of Code Snippet 10

5.2 Tuple

A tuple in Python is a collection of ordered, immutable elements. It shares resemblances with a list; nonetheless, there are notable differences:

Immutable: Tuples are immutable, meaning their elements cannot be changed after creation. Once a tuple is created, you cannot modify, add, or remove elements from it.

Ordered: Lists, tuples are ordered, which means that the elements have a specific order and can be accessed by their index.

Syntax: Tuples are defined using parentheses () and comma, to separate elements. For example: (1, 2, 3).

5.2.1 Introduction to Tuples

Tuples are frequently employed to group correlated data and owing to their immutability, can serve as keys in dictionaries (since keys must be hashable) and in scenarios where data preservation is essential.

Code Snippet 11 shows an example of creating and using tuples.

Code Snippet 11:

```
# Creating a tuple
my_tuple = (1, 2, 3, 4, 5)

# Accessing elements by index
print(my_tuple[0]) # Output: 1
print(my_tuple[2]) # Output: 3

# Slicing a tuple
slice_tuple = my_tuple[1:4] # Output: (2, 3, 4)

# Length of a tuple
length = len(my_tuple) # Output: 5

# Iterating through a tuple
for item in my_tuple:
    print(item)

# Tuple unpacking
a, b, c, d, e = my_tuple
print(a, b, c, d, e) # Output: 1 2 3 4 5

# Nested tuples
nested_tuple = ((1, 2), (3, 4))

# Tuple as dictionary keys
my_dict = {(1, 2): 'value'}

# Concatenating tuples
concatenated_tuple = my_tuple + (6, 7)

# Repeating a tuple
repeated_tuple = my_tuple * 3
```

In Code Snippet 11, creating, accessing, slicing, and manipulating tuples are shown, along with using tuples as dictionary keys and in operations such as concatenation and repetition, emphasizing the immutable nature of tuples.

Figure 5.10 depicts the output of this code when executed through PyCharm.

Figure 5.10: Output of Code Snippet 11

5.2.2 Creating a Tuple

In Python, a tuple can be formed by placing a sequence of elements within parentheses and separating them with commas.

Using Parentheses:

```
my_tuple = (1, 2, 3, "apple", "banana")
```

Using the tuple() Constructor:

```
my_tuple = tuple([1, 2, 3, "apple", "banana"] )
```

Tuple Packing:

```
item1 = "apple"
item2 = "banana"
my_tuple = item1, item2
```

Using a Generator Expression:

```
my_tuple = tuple(x for x in range(5))
```

Creating an Empty Tuple:

```
empty_tuple = ()
```

Single-Element Tuple (Note the Comma):

```
single_element_tuple = (42,)
```

5.2.3 Accessing Tuple

Accessing elements in a tuple in Python is similar to accessing elements in a list. Indexing can be employed to obtain individual elements, or slicing can be used to extract a segment of the tuple.

Code Snippet 12 shows an example of accessing a Tuple.

Code Snippet 12:

```
my_tuple = (1, 2, 3, 4, 5)

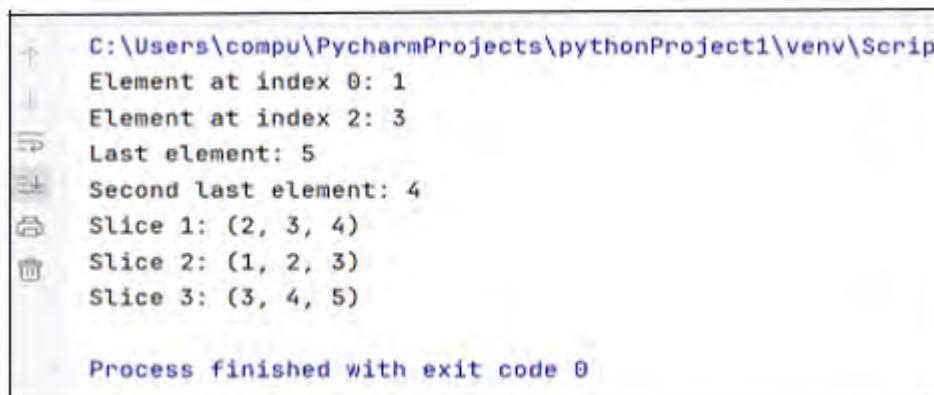
# Accessing individual elements by index
element_at_index_0 = my_tuple[0]
print("Element at index 0:", element_at_index_0) # Output: 1
element_at_index_2 = my_tuple[2]
print("Element at index 2:", element_at_index_2) # Output: 3

# Negative indexing (counting from the end)
last_element = my_tuple[-1]
print("Last element:", last_element) # Output: 5
second_last_element = my_tuple[-2]
print("Second last element:", second_last_element) # Output: 4

# Slicing to retrieve a segment of the tuple.
slice1 = my_tuple[1:4]
print("Slice 1:", slice1) # Output: (2, 3, 4)
slice2 = my_tuple[:3]
print("Slice 2:", slice2) # Output: (1, 2, 3)
slice3 = my_tuple[2:]
print("Slice 3:", slice3) # Output: (3, 4, 5)
```

In Code Snippet 12, accessing elements and slicing in tuples are demonstrated, showing how to work with tuple elements and sub-tuples.

Figure 5.11 depicts the output of this code when executed through PyCharm.



The screenshot shows the PyCharm terminal window with the following output:

```
C:\Users\compu\PycharmProjects\pythonProject1\venv\Script
Element at index 0: 1
Element at index 2: 3
Last element: 5
Second last element: 4
Slice 1: (2, 3, 4)
Slice 2: (1, 2, 3)
Slice 3: (3, 4, 5)

Process finished with exit code 0
```

Figure 5.11: Output of Code Snippet 12

5.2.4 Concatenation of Tuple

Concatenation of tuples in Python involves combining two or more tuples to create a new tuple. Tuple concatenation can be accomplished using the + operator.

Code Snippet 13 shows an example of Concatenation.

Code Snippet 13:

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)

concatenated_tuple = tuple1 + tuple2
print(concatenated_tuple)
# Output: (1, 2, 3, 4, 5, 6)
```

In Code Snippet 13, the concatenation of tuples is explored, with two tuples and, showcasing how to combine multiple tuples into one.

Code Snippet 14 shows how to concatenate more than two tuples together.

Code Snippet 14:

```
tuple3 = (7, 8, 9)

concatenated_tuple = tuple1 + tuple2 + tuple3
print(concatenated_tuple)
# Output: (1, 2, 3, 4, 5, 6, 7, 8, 9)
```

In Code Snippet 14, the concatenation of tuples is explored, with three tuples and, showcasing how to combine multiple tuples into one.

5.2.5 Slicing of Tuple

Slicing a tuple in Python operates in a manner analogous to slicing a list. Indexing and slicing notation can be applied to retrieve specific elements or sub-tuples from a tuple.

Code Snippet 15 shows an example of Slicing a Tuple.

Code Snippet 15:

```
my_tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9)

# Accessing individual elements by index
element_at_index_0 = my_tuple[0]
print("Element at index 0:", element_at_index_0) # Output: 1
element_at_index_2 = my_tuple[2]
print("Element at index 2:", element_at_index_2) # Output: 3

# Negative indexing (counting from the end)
last_element = my_tuple[-1]
print("Last element:", last_element) # Output: 9
second_last_element = my_tuple[-2]
print("Second last element:", second_last_element) # Output: 8

# Slicing to extract a portion of the tuple
slice1 = my_tuple[1:4]
print("Slice 1:", slice1) # Output: (2, 3, 4)
slice2 = my_tuple[:3]
print("Slice 2:", slice2) # Output: (1, 2, 3)
slice3 = my_tuple[3:]
print("Slice 3:", slice3) # Output: (4, 5, 6, 7, 8, 9)

# Using negative indexing in slicing
slice4 = my_tuple[-5:-2]
print("Slice 4:", slice4) # Output: (5, 6, 7)

# Slicing with a step
slice5 = my_tuple[::-2]
print("Slice 5:", slice5) # Output: (1, 3, 5, 7, 9)
slice6 = my_tuple[1::2]
print("Slice 6:", slice6) # Output: (2, 4, 6, 8)

# Reversing a tuple using slicing
reversed_tuple = my_tuple[::-1]
print("Reversed Tuple:", reversed_tuple) # Output: (9, 8, 7, 6, 5, 4, 3, 2, 1)
```

In Code Snippet 15, advanced slicing techniques in tuples, including negative indexing and slicing with steps, are discussed, showing the flexibility of tuple slicing. Figure 5.12 depicts the output of this code when executed through PyCharm.

```
C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe
Element at index 0: 1
Element at index 2: 3
Last element: 9
Second last element: 8
Slice 1: (2, 3, 4)
Slice 2: (1, 2, 3)
Slice 3: (4, 5, 6, 7, 8, 9)
Slice 4: (5, 6, 7)
Slice 5: (1, 3, 5, 7, 9)
Slice 6: (2, 4, 6, 8)
Reversed Tuple: (9, 8, 7, 6, 5, 4, 3, 2, 1)

Process finished with exit code 0
```

Figure 5.12: Output of Code Snippet 15

5.2.6 Deleting a Tuple

For deleting a tuple in Python, the `del` statement can be used.

Code Snippet 16 shows an example of to delete a Tuple.

Code Snippet 16:

```
my_tuple = (1, 2, 3, 4, 5)

# Deleting the entire tuple
del my_tuple

# Attempting to access the tuple after deletion will #result in
# an error
print(my_tuple) # Uncommenting this line will raise # a NameError
```

In Code Snippet 16, the deletion of a tuple using the `del` statement is shown, highlighting the removal of tuples from memory.

Figure 5.13 depicts the output of this code when executed through PyCharm.

```
C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:\Users\compu\Py
Traceback (most recent call last):
  File "C:\Users\compu\PycharmProjects\pythonProject1\test.py", line 7, in <module>
    print(my_tuple) # Uncommenting this line will raise # a NameError
               ^
NameError: name 'my_tuple' is not defined

Process finished with exit code 1
```

Figure 5.13: Output of Code Snippet 16

Code Snippet 17 shows the Python program on Tuple for following tasks:
Creating a tuple with information about a student.

Accessing tuple elements using indexing.

Tuple unpacking to assign tuple elements to separate variables.

Attempting to modify a tuple (which will result in a `TypeError`).

Code Snippet 17:

```
def main():
    # Creating a tuple
    student = ("John", 20, "Computer Science")

    # Accessing tuple elements
    print("Student Name:", student[0])
    print("Age:", student[1])
    print("Major:", student[2])

    # Tuple unpacking
    name, age, major = student
    print("\nTuple Unpacking:")
    print("Name:", name)
    print("Age:", age)
    print("Major:", major)

    # Attempting to modify a tuple (will result in an #error)
    try:
        student[1] = 21 # This line will raise an error
    except TypeError as e:
        print("\nError:", e)
        print("Tuples are immutable, cannot modify elements.")

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

In Code Snippet 17, tuple creation, element access, unpacking, and the immutability of tuples are illustrated through a practical example involving student information, demonstrating tuple's utility in storing and accessing related data without alteration.

Figure 5.14 depicts the output of this code when executed through PyCharm.

```
C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:\Users\compu\Py
Student Name: John
Age: 20
Major: Computer Science
 Tuple Unpacking:
Name: John
Age: 20
Major: Computer Science

Error: 'tuple' object does not support item assignment
Tuples are immutable, cannot modify elements.

Process finished with exit code 0
```

Figure 5.14: Output of Code Snippet 17

5.3 Summary

- A list in Python is an ordered, mutable collection of elements.
- Creating a list in Python involves enclosing elements within square brackets and separating them with commas.
- Input for a Python list can be obtained by using the `input()` function to collect values.
- A tuple in Python is an immutable ordered collection of elements.
- Creating a tuple in Python involves enclosing elements within parentheses and separating them with commas.
- Concatenating tuples involves using the `+` operator.

5.4 Test Your Knowledge

1. Which of the following is the correct way to create an empty list in Python?

- A) my_list = {}
- B) my_list = []
- C) my_list = ()
- D) my_list = None

2. What will be the output of the following code?

```
my_list = [1, 2, 3, 4, 5]
new_list = my_list[1:4]
print(new_list)
```

- A) [1, 2, 3]
- B) [2, 3, 4]
- C) [1, 2, 3, 4]
- D) [2, 3, 4, 5]

3. Which of the following methods is used to add an element to the end of a list in Python?

- A) insert()
- B) add()
- C) append()
- D) extend()

4. Which of the following is a key characteristic of tuples in Python?

- A) Tuples are mutable
- B) Tuples can store elements of different data types
- C) Tuples are unordered collections
- D) Tuples are immutable

5. What will be the output of the following code?

```
my_tuple = (1, 2, 3, 4, 5)
new_tuple = my_tuple[1:4]
print(new_tuple)
```

- A) (1, 2, 3)
- B) (2, 3, 4)
- C) (1, 2, 3, 4)
- D) (2, 3, 4, 5)

6. Which of the following statements about tuples is correct?

- A) Tuples support item assignment
- B) Tuples can be modified after creation
- C) Tuples allow duplicate elements
- D) Tuples use curly braces {} for creation

5.4.1 Answers to Test Your Knowledge:

1. my_list = []
2. [2,3,4]
3. append()
4. Tuples are immutable
5. (2,3,4)
6. Tuples allow duplicate elements

Try It Yourself

1. Write a Python program that does the following:
 - i. Creates an empty list called numbers.
 - ii. Takes input from the user for the number of elements they want to add to the list.
 - iii. Using a loop, prompt the user to input each number and adds it to the numbers list.
 - iv. Calculates and prints the sum of all the numbers in the list.
 - v. Prints the largest and smallest numbers in the list.
2. Write a Python program that uses list comprehension to create a new list called squares containing the squares of all even numbers from 1 to 10 (inclusive). After creating the squares list, print its contents.
3. Write a Python program that does the following:
 - i. Initializes a tuple named `student_info` with the following elements: student name, age, grade, and favorite subject.
 - ii. Prints the entire tuple.
 - iii. Using tuple unpacking, assigns the elements of the tuple to separate variables: name, age, grade, and subject.
 - iv. Prints each variable's value individually.

4. Write a Python program that uses tuples to convert temperatures between Celsius and Fahrenheit. Your program should do the following:
 - i. Creates a list of tuples, where each tuple contains a temperature in Celsius followed by its equivalent temperature in Fahrenheit.
 - ii. Asks the user to input a temperature in Celsius.
 - iii. Using a loop, searches the list of tuples to find the corresponding Fahrenheit temperature.
 - iv. Prints the input temperature in Celsius and its equivalent temperature in Fahrenheit.
 - v. The formula for converting Celsius to Fahrenheit is: $F = (C * 9/5) + 32$.



SESSION 06

DICTIONARIES AND SETS IN PYTHON

Learning Objectives

In this session, students will learn to:

- ◆ Describe dictionaries and key-value pairs
- ◆ Explain working with dictionaries operations
- ◆ Distinguish between shallow and deep copies of dictionaries
- ◆ Define sets using the `set()` constructor or curly braces `{}`
- ◆ Explain about set operations and set methods

6.1 Introduction to Dictionaries in Python

In Python, a built-in data type known as a dictionary enables data to be stored and organized in key-value pairs. In other programming languages, dictionaries are also recognized as associative arrays or hash maps. Each key within a dictionary corresponds to a specific value, with the requirement that the keys are unique. Dictionaries are characterized as unordered collections, resulting in the absence of guaranteed key-value pair order.

The key-value pairs in a dictionary are enclosed in curly braces {} and are separated by colons.

Syntax of dictionaries:

```
my_dict = {  
    "key1": "value1",  
    "key2": "value2",  
    "key3": "value3",  
    # ...  
}
```

Important components of dictionary are as follows:

Keys: Keys are the labels that are used to identify and access the corresponding values in the dictionary. Keys must be immutable, which means they cannot be changed after they are created. Common examples of keys include strings, numbers, and tuples.

Values: Values are the data elements associated with each key. Values can be of any data type, including strings, numbers, lists, other dictionaries, and more. Following Snippet shows a basic example of Dictionaries.

Example:

```
Dict = {1: 'Python', 2: 'Is', 3: 'Amazing'}  
print(Dict)  
#Output:  
{1: 'Python', 2: 'Is', 3: 'Amazing'}
```

Dictionaries are widely used in Python programming due to their efficiency in providing fast and constant-time lookups for values based on their keys.

They prove especially valuable for structured data storage and retrieval when the key-value relationship is critical.

Some common use cases for dictionaries in Python include:

01
Storing Configuration Settings: Dictionaries can be used to store configuration settings for applications, where each key represents a setting and the corresponding value stores its value.

02
Counting and Frequency Analysis: Dictionaries are often used to count the occurrences of items in a list or to perform frequency analysis on a dataset.

03
Mapping Relationships: Dictionaries are ideal for creating mappings between entities. For example, a dictionary could be employed to associate employee IDs with their respective names.

04
Caching and Memorization: Dictionaries can be used as a cache to store results of expensive computations, which helps in optimizing performance by avoiding redundant calculations.

05
Data Transformation: Dictionaries are used to transform data from one format to another. For instance, when working with JSON data, dictionaries are used to represent the hierarchical structure.

Following Snippet shows an example of a simple dictionary in Python.

Example:

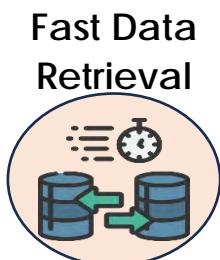
```
student_info = {  
    "name": "Alice",  
    "age": 20,  
    "major": "Computer Science"  
}
```

In above example, `name`, `age` and `major` are keys, and Alice, 20, and Computer Science are their corresponding values, respectively.

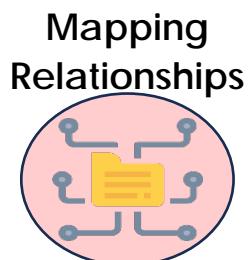
6.2 Dictionaries Using {} Key-value Pairs

Dictionaries provide a highly efficient and flexible way to manage and organize data.

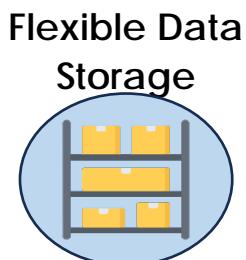
Some of the key reasons to use {} syntax to create them are as follows:



Fast Data Retrieval



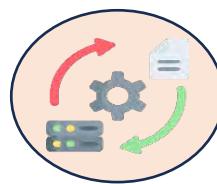
Mapping Relationships



Flexible Data Storage



Dynamic Data Management



Data Transformation



Efficient Key Lookup

An example of mapping countries to their capitals is shown here.

Example:

```
# A dictionary mapping country to their capital cities
capitals = {
    "USA": "Washington, D.C.",
    "France": "Paris",
    "Japan": "Tokyo",
    "Spain": "Madrid"
}
```

Example shows information about books using Nested Dictionaries.

Example:

```
# A dictionary represents information about books
books = {
    "book1": {
        "title": "The Great Gatsby",
        "author": "F. Scott Fitzgerald",
        "year": 1925
    }
}
```

```
    } ,
    "book2": {
        "title": "To Kill a Mockingbird",
        "author": "Harper Lee",
        "year": 1960
    }
}
```

6.3 Dictionary Methods

In Python, dictionary methods are essential for effectively managing dictionaries, facilitating easy access, modification, and interaction with the key-value data structure. Some of the commonly used Dictionary Methods are `get()`, `keys()`, `values()`, `items()`, and `pop()`.

6.3.1 `get(key, default=None)`

The `get()` method retrieves the value associated with a given key in the dictionary. When the key is present, the method gives back the associated value. If the key is not found, it returns the default value (if provided), otherwise, it returns `None`. This method is useful to avoid raising a `KeyError` when accessing a key that might not exist.

Code Snippet 1 shows an example of `get(key, default=None)`.

Code Snippet 1:

```
# Generating a dictionary by adding certain key-value #pairs.
student_scores = {
    "Alice": 95,
    "Bob": 88,
    "Carol": 92
}

# Using the get() method to retrieve scores
alice_score = student_scores.get("Alice")
# Returns 95 (key exists)
dave_score = student_scores.get("Dave")
# Returns None (key doesn't exist)

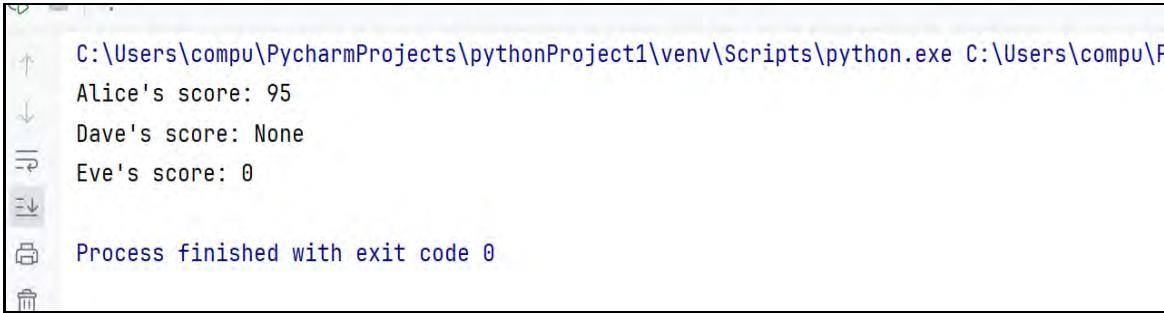
# Using the get() method with a default value
eve_score = student_scores.get("Eve", 0)
```

```
# Returns 0 (key doesn't exist, default provided)

# Printing the results
print("Alice's score:", alice_score)
print("Dave's score:", dave_score)
print("Eve's score:", eve_score)
```

In Code Snippet 1, there is a dictionary called `student_scores` that stores students' scores. The `get()` method is utilized to retrieve scores for different students.

Figure 6.1 depicts the output of this code when executed through PyCharm.



```
C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:\Users\compu\Py
Alice's score: 95
Dave's score: None
Eve's score: 0
Process finished with exit code 0
```

Figure 6.1: Output of Code Snippet 1

6.3.2 `keys()`

The `keys()` method provides a view encompassing all the keys within the dictionary. This view is iterable, enabling iteration through the keys, or it can be transformed into a list when necessary. This is valuable for scenarios where there is a requirement to iterate through or inspect the keys independently of their associated values.

Code Snippet 2 shows an example of `keys()`.

Code Snippet 2:

```
# Generating a dictionary by some key-value pairs
student_scores = {
    "Emma": 95,
    "Liam": 88,
    "Olivia": 92
}
```

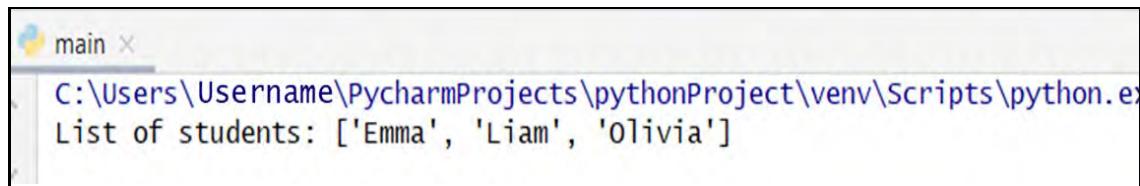
```
# Using the keys() method to retrieve and print the #keys
all_students = student_scores.keys()

# Converting the keys view into a list for #demonstration
purposes
student_list = list(all_students)

# Printing the list of keys
print("List of students:", student_list)
```

In Code Snippet 2, the `keys()` method is used to list all keys in a dictionary.

Figure 6.2 depicts the output of this code when executed through PyCharm.



```
C:\Users\Username\PycharmProjects\pythonProject\venv\Scripts\python.exe
List of students: ['Emma', 'Liam', 'Olivia']
```

Figure 6.2: Output of Code Snippet 2

The view is then, converted to a list for demonstration purposes. Finally, the list of keys is printed, providing the names of all the students in the dictionary.

6.3.3 `values()`

The `values()` method provides a display of all the values found in the dictionary. Similar to `keys()`, this view is iterable and can be converted into a list. This is useful when it is required to access the values without being concerned about the associated keys.

Code Snippet 3 shows an example of `values()`.

Code Snippet 3:

```
# Generating a dictionary by some key-value pairs
student_scores = {
    "Elena": 95,
    "Javier": 88,
    "Luisa": 92
}
```

```
# Using the values() method to retrieve and print the #values
all_scores = student_scores.values()

# Converting the values view into a list for #demonstration
purposes
score_list = list(all_scores)

# Printing the list of values
print("List of scores:", score_list)
```

In Code Snippet 3, the `values()` method is employed to list all values from a dictionary.

Figure 6.3 depicts the output of this code when executed through PyCharm.

A screenshot of a PyCharm terminal window titled "main". The command line shows the path "C:\Users\Username\PycharmProjects\pythonProject\venv\Scripts\python.exe". The output of the code execution is displayed as "List of scores: [95, 88, 92]".

Figure 6.3: Output of Code Snippet 3

In Code Snippet 3, the `values()` method is used to retrieve a view of all the values present in the `student_scores` dictionary. The view is then, converted to a list for demonstration purposes. Finally, the list of values is printed, which gives the scores of all the students in the dictionary.

6.3.4 `items()`

The `items()` method provides a perspective of the dictionary's key-value pairs as tuples, where each tuple includes a key paired with its associated value. This view is useful when you must iterate through both keys and values simultaneously, or when you want to transform the dictionary's data structure.

Code Snippet 4 shows an example of `items()`.

Code Snippet 4:

```
# Generating a dictionary by some key-value pairs
student_scores = {
    "John": 95,
    "James": 88,
    "Robert": 92
```

```

}

# Using the items() method to retrieve and print the key-value pairs
all_student_info = student_scores.items()

# Converting the items view into a list of tuples for demonstration purposes
student_info_list = list(all_student_info)

# Printing the list of key-value pairs
print("List of student information:", student_info_list)

```

In Code Snippet 4, key-value pairs are fetched as tuples from a dictionary using the `items()` method.

Figure 6.4 depicts the output of this code when executed through PyCharm.

```

C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:\Users\compu\PycharmProjects\py
List of student information: [('John', 95), ('James', 88), ('Robert', 92)]
Process finished with exit code 0

```

Figure 6.4: Output of Code Snippet 4

In Code Snippet 4, the `items()` method is used to retrieve a view of all the key-value pairs present in the `student_scores` dictionary. The view is then converted to a list of tuples for demonstration purposes. Finally, the list of key-value pairs is printed, showing both the student names and their corresponding scores.

6.3.5 `pop(key, default=None)`

The `pop()` method is used to remove and return the value associated with a given key from the dictionary. If the key is present, its value is returned and the key-value pair is removed. If the key is not found, the method returns the default value (if provided) or raises a `KeyError`. It is often used to retrieve a value and remove it from the dictionary in one step.

Code Snippet 5 shows an example of `pop(key, default=None)`.

Code Snippet 5:

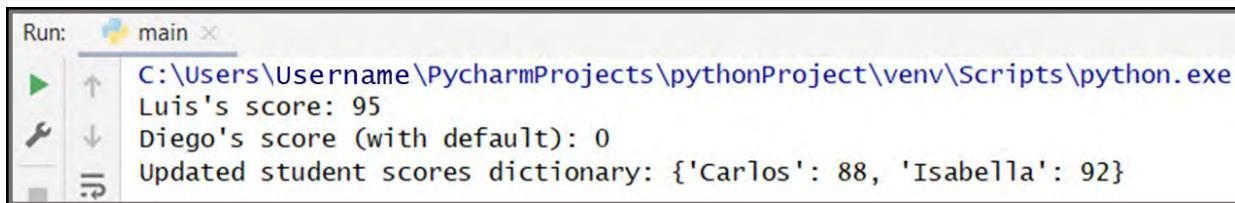
```
# Generating a dictionary by some key-value pairs
student_scores = {
    "Luis": 95,
    "Carlos": 88,
    "Isabella": 92
}

# Using the pop() method to retrieve and remove values #based
on keys
luis_score = student_scores.pop("Luis") # Removing #and
getting the value for "Luis"
diego_score = student_scores.pop("Diego", 0) # #Removing and getting the value for "Diego" (with #default)

# Printing the removed values and the updated #dictionary
print("Luis's score:", luis_score)
print("Diego's score (with default):", diego_score)
print("Updated student scores dictionary:", student_scores)
```

In Code Snippet 5, item removal using the `pop()` method is demonstrated, along with handling of non-existent keys through default values.

Figure 6.5 depicts the output of this code when executed through PyCharm.



The screenshot shows the PyCharm Run window with the following output:

```
Run: main ×
C:\Users\Username\PycharmProjects\pythonProject\venv\Scripts\python.exe
Luis's score: 95
Diego's score (with default): 0
Updated student scores dictionary: {'Carlos': 88, 'Isabella': 92}
```

Figure 6.5: Output of Code Snippet 5

In Code Snippet 5, the `pop()` method is used to retrieve and remove values based on specific keys from the `student_scores` dictionary. It demonstrates both scenarios:

- For Luis, the key exists, so `student_scores.pop(Luis)` removes the key-value pair and returns the value (score) associated with Luis.
- For Diego the key does not exist, so `student_scores.pop(Diego, 0)` removes nothing and returns the default value (0) that was provided.

6.3.6 Iterate Through Keys, Values, and Key-value Pairs Using Loops and Dictionary Methods

Iterating through keys, values, and key-value pairs using loops and dictionary methods is a common operation in Python.

Code Snippet 6 presents an example that demonstrates achieving this.

Code Snippet 6:

```
# Generating a dictionary by some key-value pairs
student_scores = {
    "Emily": 95,
    "James": 88,
    "Sophia": 92
}

# Iterating through keys using a loop
print("Iterating through keys:")
for student_name in student_scores:
    print(student_name)

# Iterating through values using a loop
print("\nIterating through values:")
for score in student_scores.values():
    print(score)

# Iterating through key-value pairs using a loop
print("\nIterating through key-value pairs:")
for student_name, score in student_scores.items():
    print(student_name, ":", score)
```

In Code Snippet 6, iteration over keys, values, and key-value pairs in a dictionary using loops is shown.

Figure 6.6 depicts the output of this code when executed through PyCharm.

```
C:\Users\Username\PycharmProjects\pythonProject\venv\Scripts\python.exe
Iterating through keys:
Emily
James
Sophia

Iterating through values:
95
88
92

Iterating through key-value pairs:
Emily : 95
James : 88
Sophia : 92
```

Figure 6.6: Output of Code Snippet 6

In the example, the first loop iterates through the keys using the dictionary directly. Each iteration provides the student's name. The second loop iterates through the values using the `values()` method. Each iteration provides the student's score. The third loop iterates through the key-value pairs using the `items()` method. Each iteration provides both the student's name and score as a tuple.

6.3.7 Manipulation of Dictionaries that Contain Other Dictionaries as Values

Code Snippet 7 shows manipulation of dictionaries that contain other dictionaries as values.

Code Snippet 7:

```
# Creating a dictionary of students with their information
# as nested dictionaries
students = {
    "Emma": {
        "age": 20,
        "major": "Engineering",
        "grades": [85, 90, 78]
    },
    "Liam": {
        "age": 22,
        "major": "Business",
```

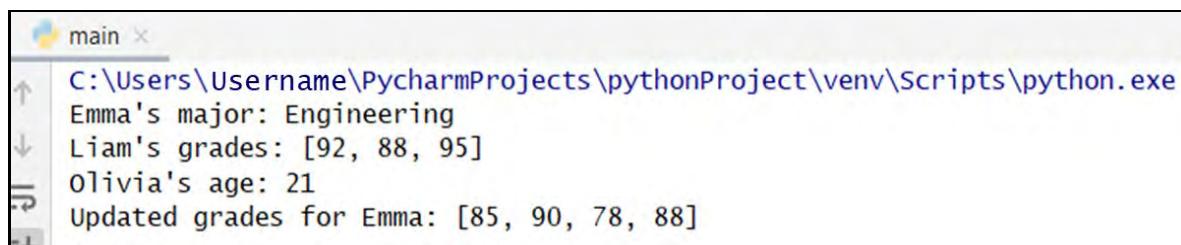
```

        "grades": [92, 88, 95]
    },
    "Olivia": {
        "age": 21,
        "major": "Computer Science",
        "grades": [75, 82, 79]
    }
}
# Accessing and manipulating nested dictionary values
emma_major = students["Emma"]["major"] # Accessing Emma's major
liam_grades = students["Liam"]["grades"] #Accessing Liam's grade
olivia_age = students["Olivia"]["age"] #Accessing Olivia's age
# Modifying nested dictionary values
students["Emma"]["grades"].append(88) # Adding a new grade for Emma
# Printing the updated information
print("Emma's major:", emma_major)
print("Liam's grades:", liam_grades)
print("Olivia's age:", olivia_age)
print("Updated grades for Emma:", students["Emma"]["grades"])

```

In Code Snippet 7, nested dictionaries are manipulated, showcasing how to access and modify nested data structures.

Figure 6.7 depicts the output of this code when executed through PyCharm.



```

main ×
C:\Users\Username\PycharmProjects\pythonProject\venv\Scripts\python.exe
Emma's major: Engineering
Liam's grades: [92, 88, 95]
Olivia's age: 21
Updated grades for Emma: [85, 90, 78, 88]

```

Figure 6.7: Output of Code Snippet 7

6.3.8 Compare Shallow and Deep Copies of Dictionaries

Shallow Copy

A shallow copy of a dictionary creates a new dictionary object, but it does not create new copies of the nested objects (such as dictionaries or lists) contained within the original dictionary. Instead, the shallow copy maintains references to the same nested objects as the original dictionary. Any modifications made to the nested objects will be reflected in both the original and the copied dictionary.

Deep Copy

A deep copy of a dictionary creates a completely independent copy of both the outer dictionary and all the nested objects it contains. This means that changes made to nested objects within the deep copied dictionary will not affect the original dictionary or its nested objects.

Syntax for shallow copy:

```
original_dict = {
    "nested_dict": { "key": "value" },
    "list": [1, 2, 3]
}
shallow_copied_dict = original_dict.copy()

# Shallow copy using copy() method
# Or
shallow_copied_dict = dict(original_dict)

# Shallow copy using dict() constructor
```

Syntax for deep copy:

```
import copy
original_dict = {
    "nested_dict": { "key": "value" },
    "list": [1, 2, 3]
}
deep_copied_dict = copy.deepcopy(original_dict)

# Deep copy using deepcopy() function
```

6.4 Introduction to a Set

In programming, a set refers to an arrangement of distinct elements without any specific order. Unlike lists or arrays, sets do not store elements in any specific order and each element within a set must be unique. Sets are a fundamental data structure used for tasks that involve storing and managing distinct values.

6.4.1 Key Characteristics of a Set



A set ensures that each element it contains is unique. It automatically removes duplicate values, maintaining only distinct elements. This property is particularly useful for tasks where duplicate data is not required or must be avoided.



The elements in a set do not have a fixed order. This means that you cannot access elements in a set using an index, unlike lists or arrays. The absence of a specific order is useful when the sequence of elements is not relevant.



Sets are mutable, allowing for the addition and removal of elements after their creation. This allows you to modify the contents of a set dynamically. However, the elements themselves must be immutable (example, numbers, strings) to ensure consistency within the set.



Sets support various operations for set theory and set manipulation, such as union, intersection, and difference. These operations provide convenient ways to work with multiple sets and perform comparisons between them.



You can iterate through the elements of a set using loops. Although the order of iteration is not guaranteed to be in the order of insertion, it covers all the elements within the set.



Use Case

Sets are used for tasks where uniqueness matters, such as removing duplicates from a collection of data, checking membership of an element, and performing mathematical operations involving sets.

Code Snippet 8 shows example in Python to illustrate the characteristics of sets.

Code Snippet 8:

```
# Creating a set
fruits = {"apple", "banana", "orange"}

# Adding elements to the set
fruits.add("grape")

# Removing an element from the set
fruits.remove("banana")

# Checking membership
print("Is 'apple' in the set?", "apple" in fruits)

# Iterating through the set
print("Fruits in the set:")
for fruit in fruits:
    print(fruit)
```

In Code Snippet 8, basic set operations including element addition and removal, membership checks, and iteration through a set are presented.

Figure 6.8 depicts the output of this code when executed through PyCharm.

The screenshot shows the PyCharm interface with a single file named 'main.py'. The code has been run, and the output is displayed in the terminal window. The output shows the results of the set operations: it first checks if 'apple' is in the set (returning True), then prints the set itself, and finally iterates through the set to print each fruit name (orange, grape, apple) on a new line.

```
C:\Users\Username\PycharmProjects\pythonProject\venv\Scripts\python.exe
Is 'apple' in the set? True
Fruits in the set:
orange
grape
apple
```

Figure 6.8: Output of Code Snippet 8

In Code Snippet, the set 'fruits' contain unique elements and duplicate elements are automatically removed. The unordered nature of the set is evident when iterating through the elements.

6.4.2 Create Sets Using the `set()` Constructor or Curly Braces {}

In Python, a set is a collection of unique elements and sets can be created using either the `set()` constructor or curly braces {}. Sets become especially handy when dealing with exclusive values, and they enable operations such as union, intersection, and difference to be carried out.

Using the `set()` Constructor

Set can be using the `set()` constructor by passing an iterable (such as a list, tuple, or string) as an argument. The constructor will automatically remove duplicate values, leaving you with a set of unique elements.

Code Snippet 9 shows an example to create a `set()` constructor.

Code Snippet 9:

```
# Creating a set using the set() constructor
fruits_set = set(["apple", "banana", "orange", "apple"])
print(fruits_set)
# Output: {'apple', 'banana', 'orange'}
```

In Code Snippet 9, the `set()` constructor is used for creating a set in Python, showcasing the initialization of sets from other data structures.

Using Curly Braces {}

A set can be created using curly braces {}. Similar to the `set()` constructor, duplicate values are automatically eliminated.

Code Snippet 10 shows an example to create a set using curly braces.

Code Snippet 10:

```
# Creating a set using curly braces {}
colors_set = {"red", "blue", "green", "red"}
print(colors_set)
# Output: {'red', 'blue', 'green'}
```

In Code Snippet 10, initialization of sets using curly braces {} is demonstrated, providing an alternative method for set creation that is concise and straightforward.

Please note: When using curly braces to create a set, be careful not to confuse it with creating an empty dictionary, which also uses curly braces, but with key-value pairs.

6.4.3 Set Operations

Set operations in Python refer to various operations that can be performed on sets to manipulate their elements and relationships.

Some of the common set operations in Python along with the respective examples are as follows:

i. Union ('|')

The union of two sets returns a new set containing all unique elements from both sets. The union operator | or the `union()` method can be used.

Following Snippet shows an example of a `union` method.

Example:

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_result = set1 | set2 # or set1.union(set2)
```

In above example, the union operation on sets is illustrated, showing how to combine elements from two sets into a single set without duplicates.

ii. Intersection ('&')

The intersection of two sets returns a new set containing elements that are common to both sets. The intersection operator & or the `intersection()` method can be used.

Following Snippet shows an example of an intersection method.

Example:

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
intersection_result = set1 & set2
# or set1.intersection(set2)
```

In above example, the intersection of sets is discussed, highlighting how to find common elements between two sets.

iii. Difference ('-')

The difference of two sets returns a new set containing elements that are in the first set, but not in the second set. The difference operator - or the `difference()` method can be used.

Following Snippet shows an example of a difference method.

Example:

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
difference_result = set1 - set2
# or set1.difference(set2)
```

In above example, the difference operation is used to find elements present in one set but not in another, showcasing set subtraction.

iv. Symmetric Difference ('^')

The symmetric difference of two sets returns a new set containing elements that are unique to either of the sets, but not in both. The symmetric difference operator ^ or the `symmetric_difference()` method can be used.

Following Snippet shows an example of the symmetric difference method.

Example:

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
symmetric_difference_result = set1 ^ set2
# or set1.symmetric_difference(set2)
```

In above example, symmetric difference is covered, demonstrating how to obtain elements in either of two sets but not in their intersection.

v. Subset ('<=' or `issubset()`)

A set 'A' is a subset of set 'B' if all elements of 'A' are also elements of 'B'. This can be checked using the `subset` operator <= or the `issubset()` method.

Following Snippet shows an example of the subset method.

Example:

```
set1 = {1, 2}
set2 = {1, 2, 3, 4}
is_subset = set1 <= set2 # or set1.issubset(set2)
```

In above example, the concept of a subset is explored, showing how to check if all elements of one set are contained within another set.

vi. Superset ('>=' or `issuperset()`)

A set A is a superset of set B if all elements of B are also elements of A. This can be checked using the `superset` operator `>=` or the `issuperset()` method.

Following Snippet shows an example of the superset method.

Example:

```
set1 = {1, 2, 3, 4}
set2 = {1, 2}
is_superset = set1 >= set2 # or set1.issuperset(set2)
```

In above example, superset checks are performed, illustrating the method to determine if one set contains all elements of another set.

6.4.4 Common Set Methods

Some of the common set methods include `add()`, `remove()`, `discard()`, `pop()`, and `clear()`. The details of each methods along with their explanations are as follows:

i. `add()`

The `add()` method adds a single element to the set. If the element is already present, the set remains unchanged.

Following Snippet shows an example of the add method.

Example:

```
my_set = {1, 2, 3}
my_set.add(4) # Adds 4 to the set
```

In above example, the `add()` method is used to add an element to a set, emphasizing the dynamic nature of sets.

ii. `remove()`

The `remove()` method removes the specified element from the set. If the element is not found, a `KeyError` is raised.

Following Snippet shows an example of the remove method.

Example:

```
my_set = {1, 2, 3}
```

```
my_set.remove(2) # Removes 2 from the set
```

In above example, the `remove()` method is discussed, detailing the removal of a specified element from a set and handling cases where the element does not exist.

iii. `discard()`

The `discard()` method removes the specified element from the set if it exists. If the element is not found, no error is raised.

Following Snippet shows an example of the `discard` method.

Example:

```
my_set = {1, 2, 3}
my_set.discard(2) # Removes 2 if present
```

In above example, the `discard()` method is introduced, offering a way to remove an element from a set without raising an error if the element is not present.

iv. `pop()`

The `pop()` method removes and returns an arbitrary element from the set. As sets are unordered, the specific element removed is not predictable.

Following Snippet shows an example of the `pop` method.

Example:

```
my_set = {1, 2, 3}
popped_element = my_set.pop() # Removes and returns #an
element
```

In above example, the `pop()` method is shown, which removes and returns an arbitrary element from the set, useful in certain algorithms.

v. `clear()`

The `clear()` method removes all elements from the set, making it empty.

Following Snippet shows an example of the `clear` method.

Example:

```
my_set = {1, 2, 3}
my_set.clear() # Clears all elements, resulting in an #empty se
```

In above example, the `clear()` method is used to remove all elements from a set, effectively resetting it to an empty state.

6.5 Summary

- Dictionaries in Python are mutable data structures that store key-value pairs.
- Dictionary keys must be unique and immutable, while values can be of any data type.
- Dictionary values can be accessed, modified, or added using their associated keys.
- Python dictionaries provide methods for fetching keys, values, items, and executing dictionary-specific operations.
- Dictionaries are used to represent structured data and manage configurations.
- Set operations in Python include combining elements, finding common elements, and identifying unique or differing elements.
- Set methods in Python provide functionalities such as adding, removing, and checking elements in sets.

6.6 Test Your Knowledge

1. What is a dictionary in Python?
 - A) An ordered collection of unique elements
 - B) A mutable sequence of values
 - C) An unordered collection of key-value pairs
 - D) A fixed-size container for storing homogeneous data

2. Which of the following is a valid way to create an empty dictionary in Python?
 - A) `empty_dict = {}`
 - B) `empty_dict = dict()`
 - C) `empty_dict = set()`
 - D) `empty_dict = []`

3. In a dictionary, what is the purpose of a key?
 - A) To store values in a sequential order
 - B) To provide an index for accessing elements
 - C) To identify and access a value in the dictionary
 - D) To store multiple values associated with a single element

4. Which dictionary method is used to retrieve the list of keys present in the dictionary?
 - A) `keys()`
 - B) `values()`
 - C) `items()`
 - D) `get()`

5. What is a unique characteristic of sets in Python?
 - A) They allow duplicate elements
 - B) They are ordered collections
 - C) They can only store integers
 - D) They store only keys, not values

6. Which set operation returns a new set containing elements that are common to both sets?
 - A) Union
 - B) Difference
 - C) Intersection
 - D) Symmetric Difference

6.6.1 Answers to Test Your Knowledge

1. An unordered collection of key-value pairs
2. `empty_dict = {}`
3. To identify and access a value in the dictionary
4. `keys()`
5. They allow duplicate elements
6. Union

Try It Yourself

1. Write a Python program that performs the following operations on a dictionary of student grades:
 - a) Create an empty dictionary called `student_grades`.
 - b) Add the following students and their corresponding grades to the dictionary:
 - "Alice" - 92
 - "Bob" - 85
 - "Charlie" - 78
 - "David" - 95
 - "Eve" - 88
 - c) Calculate and print the average grade of all students.
 - d) Find and print the student with the highest grade.
 - e) Remove the student "Charlie" from the dictionary.
 - f) Add a new student, "Frank," with a grade of 89.
 - g) Print the updated dictionary of student grades.



SESSION 07

ITERATORS, GENERATORS AND DECORATORS

Learning Objectives

In this session, students will learn to:

- ◆ Define an iterator and its purpose in Python programming
- ◆ Explain the benefits of iterators
- ◆ Explain the iterator protocol
- ◆ Describe the iterability of common Python data structures
- ◆ Explain the use of for loop to iterate over various types of iterables
- ◆ Distinguish between regular functions and generator functions
- ◆ Describe the memory-efficient iteration
- ◆ Define a decorator and its roles
- ◆ Explain the applications of decorators
- ◆ Describe the creation of decorators using classes

7.1 Iterators

An iterator in Python is an object that enables the sequential retrieval of elements from a collection, permitting the iteration and handling of each item individually. It follows a specific protocol, implemented through the `__iter__()` and `__next__()` methods. The `__iter__()` method initializes the iterator and returns itself, while the `__next__()` method retrieves the next value in the sequence and advances the iterator.

The primary purpose of iterators in Python programming is to provide a standardized and efficient way to traverse through various types of data structures, such as lists, tuples, dictionaries, and more without requiring direct access to the underlying data storage.

Advantages of iterators are as follows:



Efficient Memory Usage: By fetching elements one at a time, iterators enable the processing of large datasets without loading everything into memory simultaneously.



Lazy Evaluation: Iterators support lazy evaluation, which means that elements are computed or fetched only when requested. This enhances performance by avoiding unnecessary computations.



Consistent Interface: Regardless of the underlying data structure, iterators provide a uniform interface for accessing elements. This abstraction simplifies programming and promotes code reusability.



Sequential Processing: Iterators enforce a step-by-step processing of elements, simplifying tasks that involve performing actions on each item sequentially.



Infinite Sequences: Iterators can generate infinite sequences of values, beneficial for scenarios such as simulations or continuous data streams.

Code Snippet 1 shows a basic example of an Iterator.

Code Snippet 1:

```
class CountdownIterator:  
    def __init__(self, start):  
        self.start = start  
    def __iter__(self):  
        return self  
    def __next__(self):  
        if self.start < 0:  
            raise StopIteration  
        else:  
            self.start -= 1  
            return self.start + 1  
  
# Using the iterator  
countdown = CountdownIterator(5)  
for number in countdown:  
    print(number)
```

#Output: 5 4 3 2 1 0

7.1.1 Benefits of Using Iterators Including Memory Efficiency and Lazy Evaluation

Memory Efficiency

When dealing with large datasets, memory efficiency becomes crucial. Iterators excel in this area as they enable the handling of data one element at a time, eliminating the requirement to load the complete dataset into memory simultaneously. Traditional methods such as creating lists with all elements would consume a significant amount of memory, which could lead to performance issues or even crashes when dealing with very large datasets. Iterators, on the other hand, retrieve and process data as required, leading to a much smaller memory footprint. This makes iterators particularly useful when working with data that does not fit comfortably in memory.

Lazy Evaluation

Lazy evaluation is a powerful concept that iterators facilitate. It means that elements are computed or fetched only when they are actually required, rather than being computed all at once beforehand. Major advantages of Lazy Evaluation are as follows:

Improved Performance: In case all the elements are not required, lazy evaluation can significantly speed up code execution.

Reduced Resource Consumption: Lazy evaluation optimizes resource usage by allocating them only when required, leading to efficient utilization of CPU time and memory. This is particularly advantageous when processing large datasets, as only one element is processed at a time, minimizing memory and CPU utilization.

7.1.2 Iterable and Iterator Protocols

Iterable Protocol and the role of the `__iter__()` Method

The iterable protocol in Python is a fundamental concept that specifies the way objects can be iterated over using iterations.

An object adhering to the iterable protocol is capable of producing an iterator, which is an object used to traverse through its elements one by one.

The `__iter__()` method plays a crucial role in this protocol. When an object is designed to be iterable, it should implement the `__iter__()` method. This method is responsible for returning an iterator object that will be used to iterate through the elements of the iterable.

Code Snippet 2 shows an example of Python class that implements the `__iter__()` method.

Code Snippet 2:

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

    def __iter__(self):
        return self

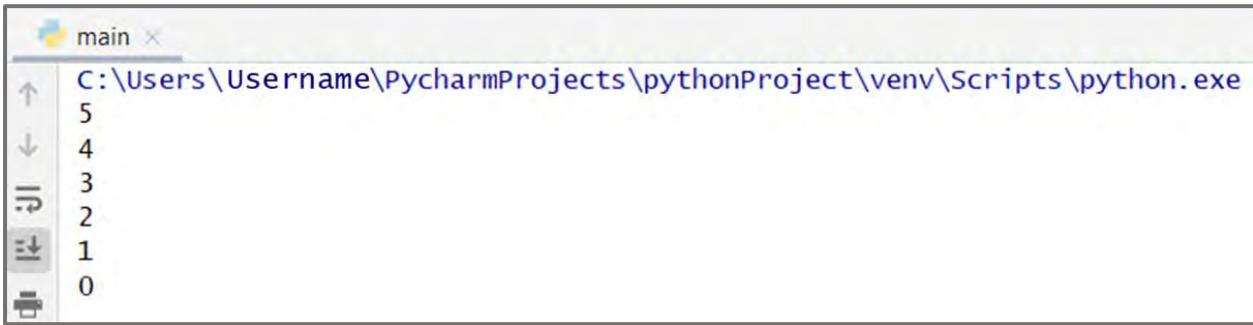
    def __next__(self):
        if self.start < 0:
            raise StopIteration
        else:
            self.start -= 1
            return self.start + 1

# Using the Countdown iterator
countdown = Countdown(5)
for number in countdown:
    print(number)
```

In this example, the `Countdown` class defines an iterator for a countdown sequence starting from the given number and stopping at 0. The `__iter__()`

method is implemented to return the instance itself and the `__next__()` method decrements the counter and returns the next value in the sequence until it reaches zero. The `StopIteration` exception is raised when the sequence is exhausted.

Figure 7.1 depicts the output of this code when executed through PyCharm.



```
C:\Users\Username\PycharmProjects\pythonProject\venv\Scripts\python.exe
5
4
3
2
1
0
```

Figure 7.1: Output of Code Snippet 2

Protocol of Iterator and the Role of the `__next__()` Method

The iterator protocol is a fundamental concept in Python that defines the process of sequentially traversing objects using iterations. Objects adhering to the iterator protocol can produce successive elements in a sequence. The primary component of the iterator protocol is the `__next__()` method.

The `__next__()` method plays a central role in the iterator protocol. When an object is designed to be an iterator, it must implement the `__next__()` method. This method is responsible for providing the next element in the sequence during each iteration.

The `__next__()` method performs two main tasks:

Returning the Next Element: It returns the next element in the sequence being traversed. If there are no more elements, the method raises the `StopIteration` exception to signal the end of the iteration.

Updating State: The method updates the internal state of the iterator to keep track of the current position in the sequence, allowing it to retrieve the correct element during the next iteration.

Code Snippet 3 shows a basic example of using `__next__()` method.

Code Snippet 3:

```
class SimpleIterator:
    def __init__(self, max_value):
        self.max_value = max_value
        self.current = 0

    def __next__(self):
        if self.current < self.max_value:
            self.current += 1
            return self.current - 1
        else:
            raise StopIteration

# Using the SimpleIterator
iterator = SimpleIterator(5)
try:
    while True:
        value = next(iterator)
        print(value)
except StopIteration:
    pass

#Output:
0
1
2
3
4
```

In this example, the `SimpleIterator` class defines an iterator that generates numbers from 0 to `max_value` - 1. The `__next__()` method is implemented to return the next value in the sequence until the maximum value is reached, at which point it raises the `StopIteration` exception.

7.1.3 `StopIteration` Exception

The `StopIteration` exception is employed to indicate the conclusion of an iteration process. This exception is raised when an iterator has no more elements

to provide, serving as a signal to terminate the iteration loop. By catching this exception, the program can effectively recognize that all elements have been iterated through, facilitating a controlled exit from the loop.

Code Snippet 4 shows an example of StopIteration Exception.

Code Snippet 4:

```
class NumberGenerator:
    def __init__(self, max_value):
        self.max_value = max_value
        self.current = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.current < self.max_value:
            self.current += 1
            return self.current - 1
        else:
            raise StopIteration

# Using the NumberGenerator iterator
generator = NumberGenerator(5)
for number in generator:
    print(number)

#Output:
0
1
2
3
4
```

In this program, the `NumberGenerator` class implements an iterator that generates numbers from 0 to `max_value - 1`. The `__next__()` method is responsible for returning the next value in the sequence until the maximum value is reached, at which point it raises the `StopIteration` exception to signal the end of iteration.

7.1.4 Common Data Structures in Python Enabling Iteration

Common Python data structures such as lists, dictionaries, sets, and strings are iterable, which means you can traverse through their elements using iterations. This is made possible by their adherence to the iterable protocol, which involves implementing the `__iter__()` method that returns an iterator object. Here is how each of these data structures is iterable:

i. List

Lists in Python are ordered collections of elements that are iterable. The `for` loop or other iteration constructs can be used to loop through each element in the list. Lists maintain the order of elements, so iterating through a list ensures that elements are processed in the same order they were added.

ii. Dictionaries

Dictionaries are unordered collections of key-value pairs. When iterating through a dictionary, the process involves iterating through its keys. Dictionaries are inherently iterable, allowing looping through keys using `for key in dictionary`.

iii. Sets

In Python, sets represent unsorted groupings of distinct elements such as dictionaries. When they are iterated through a set, you are actually iterating through its elements. Sets maintain uniqueness, so iterating through a set ensures that each unique element is processed only once.

iv. String

Strings are sequences of characters and each character can be considered an element. Strings are iterable character by character, meaning you can use iterations to traverse through each character in the string.

Code Snippet 5 shows iteration for each of the data structure.

Code Snippet 5:

```
# List iteration
my_list = [1, 2, 3, 4, 5]
for item in my_list:
    print(item)

# Output:
1
2
3
4
```

```
5
```

```
# Dictionary iteration
my_dict = {'a': 3, 'b': 4, 'c': 5}
for key in my_dict:
    print(key, my_dict[key])
# Output:
a 3
b 4
c 5

# Set iteration
my_set = {5, 6, 7, 8, 9}
for element in my_set:
    print(element)
# Output:
5
6
7
8
9

# String iteration
my_string = "Hello"
for char in my_string:
    print(char)
# Output:
H
e
l
l
o
```

7.1.5 Using `for` Loop to Iterate Over Various Types of Iterables

The `for` loop iterates over diverse iterables, including built-in types like `lists`, `dictionaries`, `sets`, and `strings`, as well as user-defined iterables.

Following examples demonstrate the use of `for` loop to iterate over different types of iterables:

i. Lists

Lists are one of the most common iterables. The `for` loop goes through each item within the list.

Code Snippet 6:

```
# Define a list
my_list = [1, 2, 3, 4, 5]

# Iterate over the list and print each item
for item in my_list:
    print(item) # Output: 1, 2, 3, 4, 5
```

ii. Dictionaries

When iterating through dictionaries, the loop iterates through the keys by default. You can access the corresponding values using the keys.

Code Snippet 7:

```
# Define a dictionary
my_dict = {'a': 3, 'b': 4, 'c': 5}
# Iterate over the dictionary and print each key-value pair
for key in my_dict:
    print(key, my_dict[key])

# Output:
a 3
b 4
c 5
```

iii. Sets

Sets are iterated similarly to lists, but remember that sets are unordered, so the order of iteration is not guaranteed.

Code Snippet 8:

```
# Define a set
my_set = {5, 6, 7, 8, 9}

# Iterate over the set and print each element
for element in my_set:
    print(element)
# Output:
5
6
7
8
9
```

iv. Strings

Strings are sequences of characters and the `for` loop iterates through each character in the string.

Code Snippet 9:

```
# Define a string
my_string = "Hello"

# Iterate over the string and print each character
for char in my_string:
    print(char)
# Output:
H
e
l
l
o
```

v. User-Defined Iterables

Custom iterables can be created by defining classes that implement the iterable protocol. This involves having an `__iter__()` method that returns an iterator object and the iterator object must have a `__next__()` method.

Code Snippet 10:

```
class MyIterable:
    def __init__(self, data):
        self.data = data
        self.index = 0

    def __iter__(self):
        return self
    def __next__(self):
        if self.index < len(self.data):
            value = self.data[self.index]
            self.index += 1
            return value
        else:
            raise StopIteration

my_iterable = MyIterable([10, 20, 30])
for value in my_iterable:
    print(value)

#Output: 10 20 30
```

7.2 Generators in Python

Generators in Python are a special type of iterable that allows you to create iterators in a more concise and memory-efficient manner. They are functions that use the `yield` keyword to produce a series of values one at a time during iteration.

Unlike regular functions that return a value and then terminate, generators can be paused and resumed, maintaining their internal state between iterations.

This unique behavior makes them particularly useful for dealing with large datasets, infinite sequences, or situations where memory efficiency is crucial.

7.2.1 Key Characteristics of Generators

Lazy Evaluation: Generators use lazy evaluation, which means that they generate and yield values only when requested during iteration. This contrasts with creating a complete list of values upfront, which can consume a significant amount of memory.

Memory Efficiency: Since generators do not store the entire sequence of values in memory, they are memory-efficient. This makes them suitable for processing large datasets or infinite sequences.

State Retention: Generators retain their internal state between iterations. This allows you to resume iteration from where it left off, making them useful for scenarios where maintaining context is important.

Simple Syntax: Generators are defined using a function with the `yield` keyword instead of the `return` keyword. When the generator function is called, it returns a generator object that can be iterated over.

Here is a simple example of a generator that generates fibonacci numbers:

Code Snippet 11 Produces Fibonacci numbers infinitely and showcases its usage.

Code Snippet 11:

```
def fibonacci_generator():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

# Using the generator
fib_gen = fibonacci_generator()
for i in range(10):
    print(next(fib_gen))
```

#Output:

0
1

```
1  
2  
3  
5  
8  
13  
21  
34
```

In this example, the `fibonacci_generator()` function is a generator that yields fibonacci numbers indefinitely. The `for` loop uses the `next()` function to retrieve and print the next value from the generator.

7.2.2 Differentiate Between Regular Functions and Generator Functions in Python

Regular functions and generator functions in Python serve different purposes and have distinct characteristics.

Table 7.1 describes the differences between the Regular and Generator functions.

Regular Functions	Generator Functions
Return: Regular functions use the <code>return</code> statement to send a value back to the caller and terminate the function's execution.	Yield: Generator functions use the <code>yield</code> keyword to produce a value one at a time during each iteration. When the <code>yield</code> statement is encountered, the function's state is saved, and the yielded value is returned. The function can continue its execution from the point it stopped.
Execution: Regular functions execute completely from start to finish when called. The entire function body is executed in one go.	Execution: Generator functions execute partially, pausing and resuming based on the <code>yield</code> statements. The function's state is maintained between iterations.
Memory: Regular functions will create and manipulate data structures in memory. If the function returns a collection (example, a list), the entire collection is generated and stored in memory.	Memory: Generator functions are memory-efficient. They do not store the entire sequence of values in memory, which is particularly useful for large datasets or infinite sequences.
Iteration: To iterate over the function's results, the entire collection of results (for example, a list of values) must be created beforehand before iterating over them.	Iteration: Generator functions generate values lazily, producing them one at a time when requested during iteration. They are well-suited for scenarios where you want to process data on-the-fly

Regular Functions	Generator Functions
	without creating a large collection upfront.
State: Regular functions do not inherently maintain state between calls. Each time a regular function is called, it starts executing from the beginning.	State: Generator functions retain their internal state between iterations. This allows you to resume the function's execution from where it left off.

Table 7.1: Difference Between Regular and Generator Functions

Example for Regular Functions:

Code Snippet 12 Produces squares of numbers up to n-1 using a generator function and demonstrates its usage

Code Snippet 12:

```
def get_squares(n):
    result = []
    for i in range(n):
        result.append(i ** 2)
    return result

squares = get_squares(5)
for square in squares:
    print(square)
```

#Output: 0 1 4 9 16

The code defines a function `get_squares(n)` that generates a list of squares up to n-1, which is then called with n=5, and the resulting list is iterated over to print the squares

Example for Generator Functions:

Code Snippet 13 Iterating over the generator object `squares_gen` yields the squares of numbers from 0 to n-1, printing each square

Code Snippet 13:

```
def generate_squares(n):
```

```
for i in range(n):
    yield i ** 2

squares_gen = generate_squares(5)
for square in squares_gen:
print(square)
```

```
#Output: 0 1 4 9 16
```

The code defines a generator function `generate_squares(n)` that yields squares of numbers from 0 to n-1. It creates a generator object `squares_gen` by calling `generate_squares(5)` and iterates over it using a for loop, printing each square.

In summary, regular functions return a value and terminate, while generator functions yield values one at a time and can be paused and resumed. Generators are memory-efficient and well-suited for scenarios to process the data lazily or efficiently generate sequences.

7.2.3 Generator Functions Using the `yield` Keyword

In this example, the `fibonacci_generator()` function generates fibonacci numbers using the `yield` keyword. The function's state is saved between iterations, allowing it to continue generating numbers efficiently.

Code Snippet 14 is used for iterating over the Fibonacci generator `fib_gen` and printing the next 10 Fibonacci numbers.

Code Snippet 14:

```
def fibonacci_generator():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

# Using the generator
fib_gen = fibonacci_generator()
for i in range(10):
    print(next(fib_gen))

#Output
0
1
```

```
1
2
3
5
8
13
21
34
```

In this example, the `generate_squares(n)` generator function yields squares of numbers from 0 to `n-1`. The function lazily generates and yields squares one at a time during iteration.

Code Snippet 15 is Iterating over the generator object `squares_gen`, yielding squares of numbers from 0 to `n-1`, and printing each square.

Code Snippet 15:

```
def generate_squares(n):
    for i in range(n):
        yield i ** 2
# Using the generator
squares_gen = generate_squares(5)
for square in squares_gen:
    print(square)

#Output: 0 1 4 9 16
```

7.2.4 Generator Functions Execution Using `yield`

Generator functions in Python pause and resume execution using the `yield` keyword. This behavior allows them to maintain their internal state between iterations.

The execution process of Generator Functions is as follows:

Pausing Execution with `yield`

When a generator function encounters the `yield` keyword, it temporarily suspends its execution and yields the value specified after the `yield` keyword. At this point, the function's state is saved, including the current position in the code and the values of local variables.

Returning Value and Pausing

The yielded value is returned to the caller (such as a loop using the generator). The function's execution is paused and the generator remains in a suspended state until the next iteration request.

Resuming Execution

When a generator function encounters the `yield` keyword, it temporarily suspends its execution and yields the value specified after the `yield` keyword. At this point, the function's state is saved, including the current position in the code and the values of local variables.

State Retention

Due to this pausing and resuming behavior, generator functions retain their internal state between iterations. This allows them to maintain context and generate values efficiently without recalculating or recomputing everything from scratch.

Code Snippet 16 is an example for Generator Functions Execution.

Code Snippet 16:

```
def countdown_generator(n):
    while n > 0:
        yield n
        n -= 1

    # Using the generator
countdown_gen = countdown_generator(5)
print(next(countdown_gen))      # Pauses at yield,
print(5)
print(next(countdown_gen))    # Resumes from where it
pauses, prints 4
print(next(countdown_gen))    # Resumes again, prints 3
```

In this example, the generator is created with `countdown_gen = countdown_generator(5)`.

The first call to `next(countdown_gen)` starts the generator and it executes until the `yield n` line. The value 5 is yielded and returned.

The next call to `next(countdown_gen)` resumes execution immediately after the `yield` statement. The value of `n` has been decremented, so 4 is yielded and returned.

Subsequent calls to `next(countdown_gen)` follow the same pattern, with the generator resuming execution where it left off and yielding the decremented value of `n` until the loop ends.

7.2.5 Lazy Evaluation in the Context of Generators

Lazy evaluation is a programming concept that refers to the practice of delaying the computation of a value until the moment when that value is actually required. In Python, lazy evaluation is a fundamental principle guiding the behavior of generators. It offers various advantages, especially when handling potentially extensive or infinite data sequences.

Generators are a type of iterable in Python that allows you to create iterators using a special kind of function called a generator function. The defining characteristic of generators is that they produce values one at a time, on-the-fly, as you iterate over them.

This is in contrast to constructing and storing all the values upfront, which can lead to unnecessary memory consumption and processing time, especially when dealing with large datasets.

Lazy Evaluation works with Generators in following pattern:

7.2.6 Generating Values On-the-Fly with Generator Expressions

Delayed Computation: Generators do not immediately compute values upon creation; instead, the generator function defines the process to generate values when requested.

Value Generation on Demand: While iterating over a generator, the generator function is invoked step by step, producing the next value in the sequence with each iteration.

Efficient Memory Usage: Generators are memory-efficient as they produce values one at a time and avoid storing the entire sequence, which is crucial when handling large datasets exceeding memory capacity.

Infinite Sequences: Generators adeptly manage infinite sequences by generating values on-the-fly. This enables creating generators for theoretically boundless sequences (example, prime numbers) without memory or computation problems.

Processing Streams: Generators excel at handling data streams such as reading lines from files or network data. They process data incrementally, avoiding the requirement to load the entire stream into memory.

7.2.7 Generating Values On-the-Fly with Generator Expressions

A generator expression is a concise way to create a generator in Python. It allows you to define a generator using a single line of code, similar to a list comprehension, but with parentheses instead of square brackets. Generator expressions are particularly useful when you must generate values on-the-fly without storing them in memory, which is ideal for memory-efficient processing of large datasets or infinite sequences.

Following example demonstrates a generator expression and its ability to generate values dynamically as required.

Syntax of Generator Expression:

```
(generator_expression)
```

Example for Generating Square numbers:

```
squares_generator = (x ** 2 for x in range(10))
```

In the previous example, `(x ** 2 for x in range(10))` is a generator expression that defines the generation of squares of numbers from 0 to 9 (inclusive).

As you iterate over the generator, the expression `(x ** 2 for x in range(10))` is evaluated for each value of x in the range. The result of `x ** 2` is the square of the current x value.

Generator expressions are a compact and efficient way to create generators that generate values on-the-fly. They offer memory-efficient processing of data, are well-suited for large datasets or infinite sequences, and follow the lazy evaluation principle, making them an essential tool for effective Python programming.

7.3 Decorators

A decorator is a programming concept used to enhance or modify the behavior of functions or methods in a flexible and reusable manner. It is a higher-order function, which means it takes a function as an argument and returns a new function that often extends or alters the original function's functionality.

Decorators play a crucial role in software development by enabling the separation of concerns, modularity, and code reusability. They are commonly used to add functionality to functions or methods without modifying their core implementation. This makes code maintenance and development more manageable, as changes can be made to the decorator itself rather than to each function that uses it.

The syntax to use decorators typically involves placing the decorator's name, prefixed by the @ symbol, discussed in the function definition. When the function is called, it passes through the decorator, which can modify its behavior, add functionality, or process inputs and outputs before or after execution.

7.3.1 Creating a Decorator to Modify the Functions Behavior

In this example, the `measure_time` decorator calculates and prints the execution time of a function. In the second example, the `uppercase_args` decorator converts all string-type input arguments to uppercase before passing them to the decorated function. Both of these decorators modify the behavior of the functions they are applied to without altering the original code of those functions.

Code Snippet 17 shows the Decorator function to calculate execution time of a function.

Code Snippet 17:

```
#Decorator to Calculate execution time of a function

import time
def measure_time(func):
    def wrapper(*args, **kwargs):
```

```

        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"{func.__name__} took {end_time - start_time:.4f} seconds to execute")
    return result
return wrapper

@measure_time
def slow_function():
    time.sleep(2)
    print("Function execution complete")

slow_function()

# Decorator to convert input arguments to uppercase
def uppercase_args(func):
    def wrapper(*args, **kwargs):
        modified_args = [arg.upper() if isinstance(arg, str) else arg for arg in args]
        result = func(*modified_args, **kwargs)
        return result
    return wrapper

@uppercase_args
def greet(name):
    return f"Hello, {name}!"

print(greet("Alice"))
print(greet("Bob"))

```

Figure 7.1 depicts the output of this code when executed through PyCharm.

```

Function execution complete
slow_function took 2.0143 seconds to execute
Hello, ALICE!
Hello, BOB!

```

Figure 7.1: Output of Code Snippet 17

Output Explanation

The line function execution complete is the output of the `slow_function()` call inside the `measure_time` decorator. It indicates that the function execution is complete.

The line `slow_function` took 2.0006 seconds to execute is the result of the `measure_time` decorator. It shows the name of the function and the time it took to execute in seconds.

The lines Hello, ALICE! and Hello, BOB! are the results of the `greet` function calls after the `uppercase_args` decorator has modified the input arguments. The decorator has converted the names to uppercase before greeting them.

7.3.2 Application of Decorators

Decorators are incredibly useful for adding specific functionalities to functions or methods without modifying their original code.

Decorators can be applied to achieve common purposes such as logging, authentication, and memorization.

i. Logging

Logging is essential for debugging and monitoring the behavior of functions. A decorator can be used to automatically log information about function calls, arguments, and results without cluttering the actual function code.

Code Snippet 18 shows the application of log function.

Code Snippet 18:

```
def log_function_call(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with args: {args}, kwargs: {kwargs}")
        result = func(*args, **kwargs)
        print(f"{func.__name__} returned: {result}") # Fixed the print statement
        return result
    return wrapper

@log_function_call
def add(a, b):
```

```
    return a + b

result = add(3, 5)

#Output:
Calling add with args: (3, 5), kwargs: {}
add returned: 8
```

When you call the add function, the decorator `log_function_call` logs information about the function call.

ii. Authentication

Decorators can be used to implement authentication checks for protecting certain functions or methods. This ensures that only authorized users can access sensitive parts of the code.

Code Snippet 19 shows the application of authenticate function.

Code Snippet 19:

```
def authenticate(func):
    def wrapper(*args, **kwargs):
        if is_authenticated():
            return func(*args, **kwargs)
        else:
            return "Access denied"
    return wrapper

@authenticate
def view_sensitive_data():
    return "Sensitive data"

def is_authenticated():
    return True      # Replace with actual
#authentication logic

result = view_sensitive_data()
print(result)
```

```
#Output: Sensitive data
```

The authenticate decorator checks whether the user is authenticated before allowing them to access the `view_sensitive_data` function.

iii. Memoization

Memoization is a technique to optimize functions by caching their results for certain input arguments. Decorators can be used to automatically cache function results and return cached results if the same arguments are encountered again.

Code Snippet 20 shows the application of Memoization.

Code Snippet 20:

```
def memoize(func):
    cache = {}

    def wrapper(*args):
        if args in cache:
            return cache[args]
        result = func(*args)
        cache[args] = result
        return result
    return wrapper

@memoize
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(10))
# Calculates and caches Fibonacci(0) to Fibonacci(10)
```

The `memoize` decorator helps to store previously computed fibonacci values, making subsequent calls faster.

7.3.3 Decorators Using Classes Instead of Functions

Decorators can also be implemented using classes instead of functions. This involves creating a class that acts as a decorator by implementing the `__call__` method.

Code Snippet 21 shows how to create decorators using classes for logging.

Code Snippet 21:

```
class LogFunctionCall:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print(f"Calling {self.func.__name__} with
args: {args}, kwargs: {kwargs}")
        result = self.func(*args, **kwargs)
        print(f"{self.func.__name__} returned:
{result}")
        return result

@LogFunctionCall
def add(a, b):
    return a + b
result = add(3, 5)
```

Code Snippet 22 shows how to create decorators using classes for authentication.

Code Snippet 22:

```
class Authenticate:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        if self.is_authenticated():
            return self.func(*args, **kwargs)
```

```

        else:
            return "Access denied"
    def is_authenticated(self):
        return True      # Replace with the actual
#authentication logic
@Authenticate
def view_sensitive_data():
    return "Sensitive data"
result = view_sensitive_data()
print(result)

```

Code Snippet 23 shows how to create decorators using classes for memorization.

Code Snippet 23:

```

class Memoize:
    def __init__(self, func):
        self.func = func
        self.cache = {}

    def __call__(self, *args):
        if args in self.cache:
            return self.cache[args]
        result = self.func(*args)
        self.cache[args] = result
        return result

@Memoize
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(10))

```

In these examples, each class serves as a decorator by implementing the `call__` method. This method is executed when the decorated function is called, allowing you to modify the function's behavior or add extra functionality. Class-based decorators offer more flexibility, as you can store additional state or methods within the decorator class itself.

7.4 Summary

- A Python iterator enables sequential access through a collection without storing the entire set-in memory.
- Iterators in Python offer memory-efficient traversal and lazy evaluation, enhancing resource utilization and performance.
- The iterator protocol involves the `__iter__()` method for returning the iterator object and the `__next__()` method for retrieving successive elements, enabling sequential iteration through a collection.
- Python data structures such as lists, dictionaries, sets, and strings are inherently iterable, allowing for easy traversal through their elements.
- The `for` loop in Python provides a universal mechanism to iterate over diverse iterables, including user-defined ones.
- Generators enable memory-efficient iteration over extensive datasets by producing values immediately rather than preloading them into memory.
- A decorator is a Python construct that enhances or modifies functions/methods by wrapping them with additional behavior without changing their core implementation.
- Decorators are applied to functions in Python for purposes such as logging, authentication, memorization, and so on, seamlessly enhancing their functionality.
- Decorators can be implemented using classes by defining a class with a `__call__()` method to encapsulate the additional behavior.

7.5 Test Your Knowledge

1.What is an iterator in Python?

- A) An iterator is a built-in data type in Python used for storing key-value pairs
- B) An iterator is a function that performs mathematical operations on iterable objects
- C) An iterator is an object used to iterate over elements in a sequence or collection
- D) An iterator is a graphical user interface (GUI) library in Python

2.What are the benefits of using iterators in Python?

- A) Iterators help in creating complex data structures such as graphs and trees
- B) Iterators ensure eager evaluation of all elements in a sequence
- C) Iterators automatically manage memory allocation and deallocation
- D) Iterators provide memory efficiency by fetching and processing elements one at a time

3.What are the purposes of the `__iter__()` and `__next__()` methods in Python iterators?

- A) `__iter__()` is used to define the iterator object, while `__next__()` is used to fetch the next element in the iteration
- B) `__iter__()` is used to fetch the next element in the iteration, while `__next__()` is used to define the iterator object
- C) Both `__iter__()` and `__next__()` are used to define the iterator object
- D) Both `__iter__()` and `__next__()` are used to fetch the next element in the iteration

4.Python data structures such as lists, dictionaries, sets, and strings are iterable.

- A) True
- B) False

5.What are the main differences between regular functions and generator functions in Python?

- A) Regular functions can return multiple values, while generator functions can only return a single value.
- B) Regular functions can be paused and resumed during execution, while generator functions cannot be paused.
- C) Generator functions use the yield keyword to produce a sequence of values lazily, while regular functions use the return keyword to provide a single result.
- D) Regular functions are more memory-efficient compared to generator functions.

6. What is a decorator in Python?

- A) A decorator is a Python module used for creating Graphical User Interfaces (GUI)
- B) A decorator is a built-in function used for arithmetic operations on data structures
- C) A decorator is a design pattern used to enhance or modify the behavior of functions or methods without changing their source code
- D) A decorator is a keyword used to define new classes in Python

7.Decorators in Python are commonly used for purposes such as logging, authentication, memorization, and more.

- A) True
- B) False

8.How can you create decorators using classes instead of functions in Python?

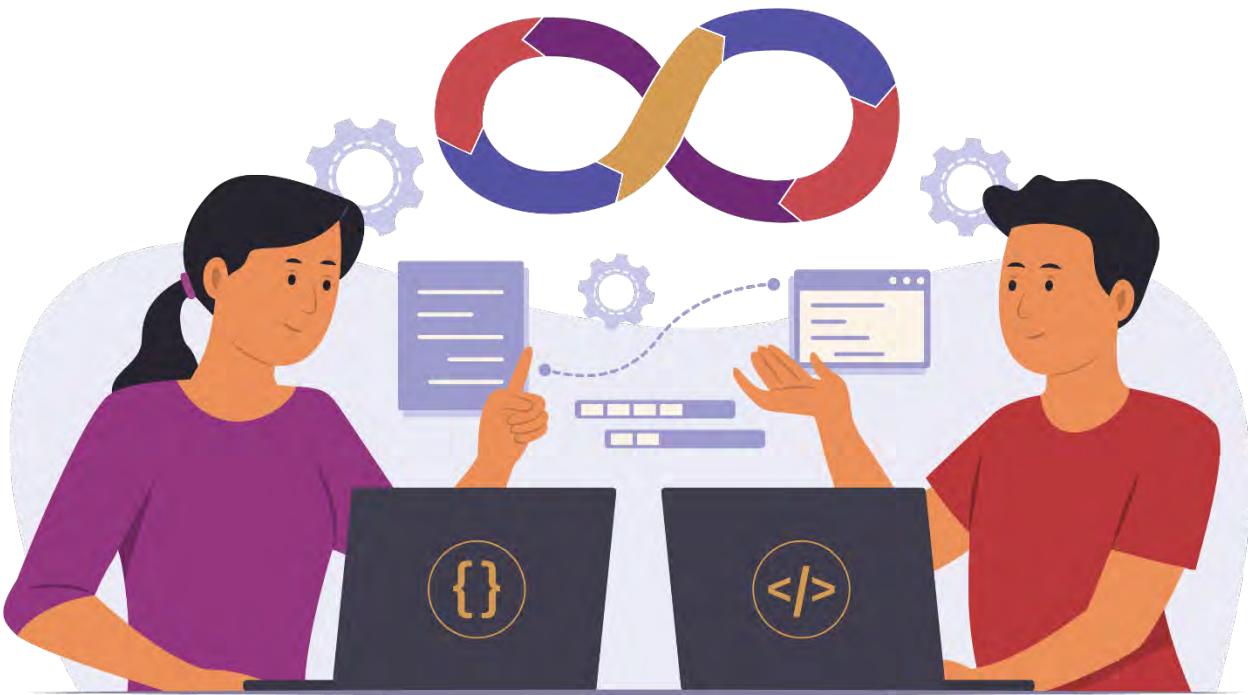
- A) It is not possible to create decorators using classes; decorators can only be implemented as functions
- B) By defining a class with a `__init__` method to initialize and a `__call__` method to implement the decorator behavior
- C) By creating a class with a `decorate` method and using it as a wrapper for the functions you want to enhance
- D) By inheriting the`@decorator` class and providing the desired modification logic in the child class

7.5.1 Answers to Test Your Knowledge

1. An iterator is an object used to iterate over elements in a sequence or collection.
2. Python has a strong emphasis on readability and uses indentation for code blocks instead of braces or keywords.
3. Iterators automatically manage memory allocation and deallocation.
4. True
5. Generator functions use the yield keyword to produce a sequence of values lazily, while regular functions use the return keyword to provide a single result.
6. A decorator is a design pattern used to enhance or modify the behavior of functions or methods without changing their source code.
7. True
8. By defining a class with a `__init__` method to initialize and a `__call__` method to implement the decorator behavior.

Try It Yourself

1. Create a Python program to generate a Fibonacci sequence using a custom iterator class.
2. Create a Python program that demonstrates creating a class-based decorator to measure the execution time of a function.



SESSION 08

OBJECT ORIENTED PROGRAMMING (OOPS)

Learning Objectives

In this session, students will learn to:

- ◆ Describe a class in Python
- ◆ Explain the use of objects and Constructor
- ◆ List the Methods, Getter, and Setter
- ◆ Explain the use of Static Field
- ◆ Define and describe Garbage Collection

8.1 Introduction to OOPs

It is a programming paradigm that focuses on designing and structuring code around the concept of objects, which are instances of classes. OOP aims to organize code in a more modular and logical manner by encapsulating data and behavior into objects. Python employs classes and objects to implement the principles of OOP.

8.2 Class in Python

A class is a blueprint or template for creating objects. It establishes the characteristics (data members) and behaviors (functions) that will be possessed by the class's objects. Classes serve as a way to define the structure and behavior of objects.

Code Snippet 1 shows basic example to define and use a class in Python.

Code Snippet 1:

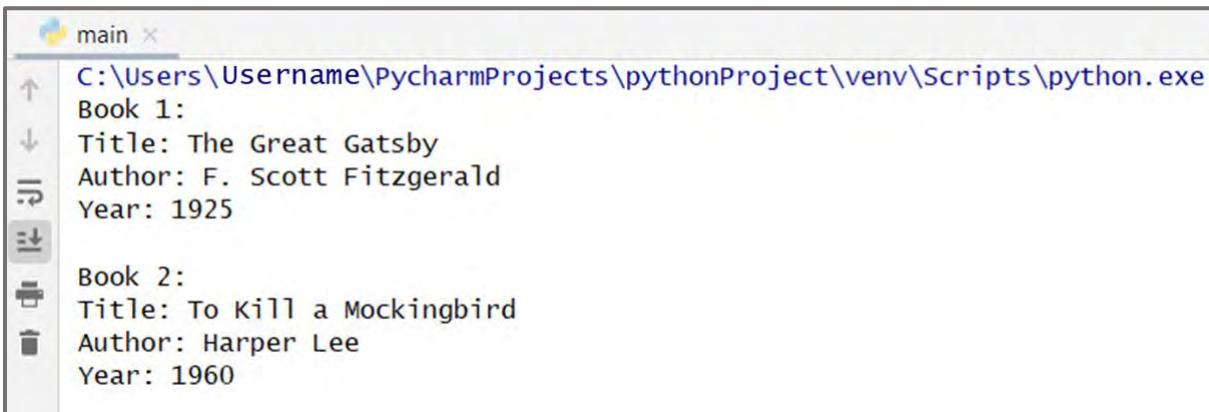
```
class Book:
    def __init__(self, title, author, year):
        self.title = title
        self.author = author
        self.year = year

    def display_info(self):
        print(f"Title: {self.title}")
        print(f"Author: {self.author}")
        print(f"Year: {self.year}")

# Create instances of the Book class
book1 = Book("The Great Gatsby", "F. Scott Fitzgerald", 1925)
book2 = Book("To Kill a Mockingbird", "Harper Lee", 1960)
# Display information about the books
print("Book 1:")
book1.display_info()
print("\nBook 2:")
book2.display_info()
```

Code Snippet 1 defines a `Book` class with attributes for `title`, `author`, and `year`. Instances of the class are created for two books, and their information is displayed using the `display_info` method, which prints the `title`, `author`, and `year`.

Figure 8.1 depicts the output of this code when executed through PyCharm.



```
C:\Users\Username\PycharmProjects\pythonProject\venv\Scripts\python.exe
Book 1:
Title: The Great Gatsby
Author: F. Scott Fitzgerald
Year: 1925

Book 2:
Title: To Kill a Mockingbird
Author: Harper Lee
Year: 1960
```

Figure 8.1: Output of Code Snippet 1

A Book class is defined, with an `__init__` constructor responsible for initializing the `title`, `author` and `year` attributes for each instance of a book.

Two instances of the Book class are created `book1` and `book2`, providing values for the constructor parameters.

The `display_info` method is called on each instance to display information about the books.

8.3 Objects in Python

In Python, an object is a fundamental concept that represents a real-world entity or data structure that can have attributes (characteristics) and methods (functions that operate on the object's data). Everything in Python is an object, whether it is a simple data type such as numbers and strings, or more complex structures such as lists, dictionaries, classes, and instances.

Following are some of the important factors about Objects in Python.

i. Attributes

An object's attributes are its properties. For example, a string object might have attributes such as its length or its content. You can access an object's attributes using dot notation, such as `object_name.attribute_name`.

Code Snippet 2 shows an example to use attributes in Python.

Code Snippet 2:

```
class Car:  
    # Class attribute  
    wheels = 4  
  
    def __init__(self, make, model, year):  
        # Instance attributes  
        self.make = make  
        self.model = model  
        self.year = year  
        self.speed = 0 # Initial speed  
  
    def accelerate(self, increment):  
        self.speed += increment  
  
    def brake(self, decrement):  
        self.speed -= decrement  
  
    def display_info(self):  
        print(f"Make: {self.make}")  
        print(f"Model: {self.model}")  
        print(f"Year: {self.year}")  
        print(f"Number of Wheels: {Car.wheels}")  
        print(f"Current Speed: {self.speed} km/h")  
  
# Create instances of the Car class  
car1 = Car("Toyota", "Corolla", 2022)  
car2 = Car("Ford", "Mustang", 2023)  
  
# Accessing and modifying instance attributes  
car1.accelerate(30)  
car2.accelerate(50)  
car1.display_info()  
car2.display_info()  
  
# Accessing class attribute  
print(f"Number of wheels for car1: {car1.wheels}")  
print(f"Number of wheels for car2: {car2.wheels}")  
  
# Modifying class attribute  
Car.wheels = 6  
print("After modifying class attribute:")  
car1.display_info()  
car2.display_info()
```

Code Snippet 2 defines a `Car` class with attributes for `make`, `model`, `year`, and `speed`. Methods are provided to accelerate and brake the car's speed, as well as display its information. Instances of the class are created for two cars, their speeds are adjusted, and their information is displayed. The class attribute `wheels` is also accessed and modified for both cars.

Figure 8.2 depicts the output of this code when executed through PyCharm.

```
main
C:\Users\Username\PycharmProjects\pythonProject\venv\Scripts\python.exe
Make: Toyota
Model: Corolla
Year: 2022
Number of Wheels: 4
Current Speed: 30 km/h
Make: Ford
Model: Mustang
Year: 2023
Number of Wheels: 4
Current Speed: 50 km/h
Number of wheels for car1: 4
Number of wheels for car2: 4
After modifying class attribute:
Make: Toyota
Model: Corolla
Year: 2022
Number of Wheels: 6
Current Speed: 30 km/h
Make: Ford
Model: Mustang
Year: 2023
Number of Wheels: 6
Current Speed: 50 km/h
```

Figure 8.2: Output of Code Snippet 2

ii. Methods

Methods are functions that are associated with an object and can operate on its data. They are essentially functions that are bound to objects. An object's methods can be called using a dot notation as well, such as `object_name.method_name()`.

Code Snippet 3 shows an example to use Methods in Python.

Code Snippet 3:

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

    def calculate_area(self):
        area = 3.14159 * self.radius * self.radius
```

```

        return area

    def calculate_circumference(self):
        circumference = 2 * 3.14159 * self.radius
        return circumference

    def display_info(self):
        print(f"Circle with radius {self.radius}")
        print(f"Area: {self.calculate_area()}")
        print(f"Circumference: {self.calculate_circumference()}")
    }

# Create instances of the Circle class
circle1 = Circle(5)
circle2 = Circle(8)

# Calling methods on instances
circle1.display_info()
circle2.display_info()

```

Code Snippet 3 defines a `Circle` class with methods to calculate its `area` and `circumference` based on the given `radius`. Instances of the class are created for two circles, and their information is displayed, including the `radius`, `area`, and `circumference`.

Figure 8.3 depicts the output of this code when executed through PyCharm.

```

main ×
C:\Users\Username\PycharmProjects\pythonProject\venv\Scripts\python.exe
Circle with radius 5
Area: 78.53975
Circumference: 31.4159
Circle with radius 8
Area: 201.06176
Circumference: 50.26544

```

Figure 8.3: Output of Code Snippet 3

iii. Classes and Instances

In Python, custom objects can be created by defining classes. A class is a blueprint for creating objects of a specific type. When an object is created from a class, it is referred to as an instance. The attributes and methods defined in the class are shared by all instances of that class.

Code Snippet 4 shows an example of using a class in Python.

Code Snippet 4:

```
class Person:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
    def set_name(self, new_name):  
        self.name = new_name  
  
    def set_age(self, new_age):  
        self.age = new_age  
  
    def display_info(self):  
        print(f"Name: {self.name}")  
        print(f"Age: {self.age}")  
  
# Create instances of the Person class  
person1 = Person("Alice", 25)  
person2 = Person("Bob", 30)  
  
# Display initial information  
print("Initial information:")  
person1.display_info()  
person2.display_info()  
  
# Update information using methods  
person1.set_name("Alicia")  
person2.set_age(32)  
  
# Display updated information  
print("\nUpdated information:")  
person1.display_info()  
person2.display_info()
```

Code Snippet 4 defines a `Person` class with attributes for `name` and `age`, along with methods to set new values for these attributes and display the information. Instances of the class are created for two persons, their initial information is displayed, then their information is updated using the provided methods, and finally, the updated information is displayed.

Figure 8.4 depicts the output of this code when executed through PyCharm.

```
C:\Users\Username\PycharmProjects\pythonProject\venv\Scripts\python.exe
Initial information:
Name: Alice
Age: 25
Name: Bob
Age: 30

Updated information:
Name: Alicia
Age: 25
Name: Bob
Age: 32
```

Figure 8.4: Output of Code Snippet 4

iv. Instantiation

Instantiation is the term used to denote the procedure of generating an instance from a class. This involves creating a new object based on the class's blueprint.

Code Snippet 5 shows an example of using instantiation.

Code Snippet 5:

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

    def display_info(self):
        print(f"Name: {self.name}")
        print(f"Age: {self.age}")
        print(f"Grade: {self.grade}")

# Creating instances using the constructor
student1 = Student("Alice", 15, 9)
student2 = Student("Bob", 16, 10)

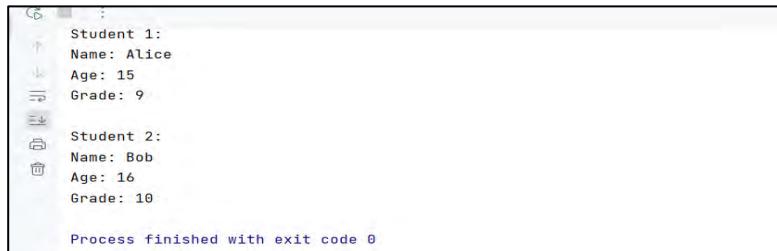
# Displaying information using the display_info method
print("Student 1:")
student1.display_info()

print("\nStudent 2:")
student2.display_info()
```

Code Snippet 5 defines a `Student` class with attributes for `name`, `age`, and `grade`, along with a method to display this information. Instances of the class are created

for two students using the constructor, and their information is displayed using the `display_info()` method.

Figure 8.5 depicts the output of this code when executed through PyCharm.



```
Student 1:  
Name: Alice  
Age: 15  
Grade: 9  
  
Student 2:  
Name: Bob  
Age: 16  
Grade: 10  
  
Process finished with exit code 0
```

Figure 8.5: Output of Code Snippet 5

8.4 Constructors

A Constructor is a special method that gets automatically called when an object of a class is created. It is used to initialize the attributes and perform any necessary setup for the object. The constructor method is named `__init__` and it is a fundamental part of object-oriented programming in Python.

Following Syntax shows a basic structure of a constructor in Python.

Syntax of a constructor:

```
class ClassName:  
    def __init__(self, parameter1, parameter2, ...):  
        # Initialize attributes here  
        self.attribute1 = parameter1  
        self.attribute2 = parameter2  
    # ...
```

Code Snippet 6 shows an example of a constructor.

Code Snippet 6:

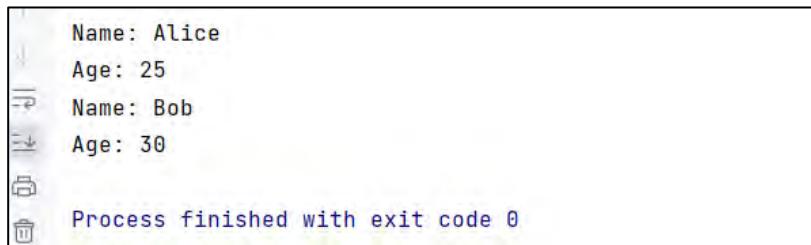
```
class Person:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
    def display_info(self):  
        print(f"Name: {self.name}")  
        print(f"Age: {self.age}")
```

```
# Creating instances using the constructor
person1 = Person("Alice", 25)
person2 = Person("Bob", 30)

# Displaying information using the display_info method
person1.display_info()
person2.display_info()
```

Code Snippet 6 defines a `Person` class with attributes for `name` and `age`, along with a method to display this information. Instances of the class are created for two persons using the constructor, and their information is displayed using the `display_info()` method.

Figure 8.6 depicts the output of this code when executed through PyCharm.



The screenshot shows the PyCharm terminal window. It displays the output of the Python code execution. The output consists of two sets of information, each representing a person's details. The first set is for 'Alice' (Name: Alice, Age: 25) and the second is for 'Bob' (Name: Bob, Age: 30). Both entries show a small tree icon to the left of the name, indicating they are objects. At the bottom of the terminal, there is a message 'Process finished with exit code 0'.

Figure 8.6: Output of Code Snippet 6

8.5 Methods

Methods are functions that are defined within a class and are associated with instances of that class. Methods allow you to define actions that can be performed on the objects created from the class. They operate on the data (attributes) of the class instances and can modify their state or provide useful functionality.

Methods are an essential part of object-oriented programming, allowing you to encapsulate behavior within your class definitions. They can access and manipulate the instance attributes and perform various operations related to the class.

Instance Method

These methods are associated with instances of a class and have access to instance attributes and other instance methods. The first parameter of an instance method is always `self`, which refers to the instance itself.

Class Method

These methods are defined using the `@classmethod` decorator. They take the class itself as their first parameter (`cls`) and can be used to operate on class-level attributes or perform actions related to the entire class.

Static Method

These methods are defined using the `@staticmethod` decorator. They do not have access to the instance or class attributes and behave as regular functions that are part of the class namespace. They are usually used for utility functions that are related to the class but do not require access to its state.

Code Snippet 7 shows an example demonstrating different types of methods.

Code Snippet 7:

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

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

    @classmethod
    def create_square(cls, side):
        return cls(side, side)

    @staticmethod
    def is_large_rectangle(rect):
        return rect.width * rect.height > 50

# Creating instances of the Rectangle class
rectangle1 = Rectangle(5, 10)
rectangle2 = Rectangle.create_square(7)

# Using instance methods
area1 = rectangle1.calculate_area()
area2 = rectangle2.calculate_area()

# Using static method
is_large1 = Rectangle.is_large_rectangle(rectangle1)
is_large2 = Rectangle.is_large_rectangle(rectangle2)

print(f"Area of rectangle1: {area1}")
print(f"Area of rectangle2: {area2}")
```

```
print(f"Is rectangle1 large? {is_large1}")
print(f"Is rectangle2 large? {is_large2}")
```

In Code Snippet 7, an instance method (`calculate_area`), a class method (`create_square`), and a static method (`is_large_rectangle`) are defined within the `Rectangle` class. Instances of the class are created, and these methods are utilized for various operations.

Figure 8.7 depicts the output of this code when executed through PyCharm.

The image shows a PyCharm terminal window with the following output:

```
Area of rectangle1: 50
Area of rectangle2: 49
Is rectangle1 Large? False
Is rectangle2 Large? False
Process finished with exit code 0
```

Figure 8.7: Output of Code Snippet 7

Getter and Setter Methods

Setter and getter methods, also known as setter and getter functions, are methods used to manage the access and modification of attributes (instance variables) in a class.

They facilitate control over attribute setting and retrieval, enhancing encapsulation and supporting data validation. These methods are commonly used in OOP to enforce proper data encapsulation and maintain a clear interface for interacting with object attributes.

Accessor Method
A getter method is used to retrieve the value of an attribute. It provides controlled access to the attribute, allowing you to implement validation or additional logic by returning the value.

Mutator Method
A setter method is used to set the value of an attribute. It provides controlled assignment of the attribute, allowing you to implement validation or other operations by actually setting the value.

Code Snippet 8 shows the use of getter and setter methods in Python.

Code Snippet 8:

```
class Person:
    def __init__(self, name, age):
        self._name = name
    # Prefixing with underscore indicates a "protected" attribute
        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, new_name):
        if isinstance(new_name, str):
            self._name = new_name
        else:
            print("Invalid name format.")

    def set_age(self, new_age):
        if isinstance(new_age, int) and new_age >= 0:
            self._age = new_age
        else:
            print("Invalid age format.")

    def display_info(self):
        print(f"Name: {self._name}")
        print(f"Age: {self._age}")

# Create an instance of the Person class
person = Person("Alice", 25)

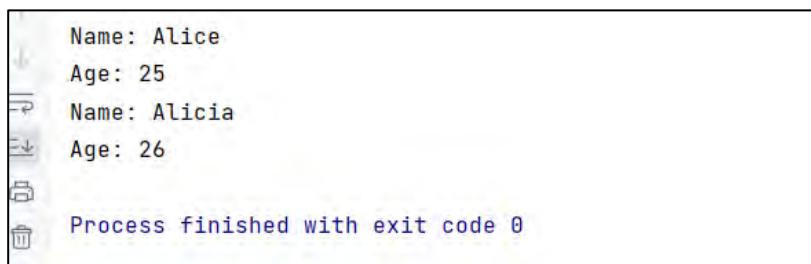
# Using getter methods
print(f"Name: {person.get_name()}")
print(f"Age: {person.get_age()}")

# Using setter methods
person.set_name("Alicia")
person.set_age(26)
```

```
# Display information using display_info method
person.display_info()
```

In Code Snippet 8, getter methods (`get_name` and `get_age`) and setter methods (`set_name` and `set_age`) are defined for the `Person` class. These methods allow controlled access to the attributes `_name` and `_age`, enforcing validation rules by modifying or retrieving the attribute values.

Figure 8.8 depicts the output of this code when executed through PyCharm.



```
Name: Alice
Age: 25
Name: Alicia
Age: 26
Process finished with exit code 0
```

Figure 8.8: Output of Code Snippet 8

8.6 Static Field

In Python, the term static field is often used interchangeably with class attribute. A static field or class attribute is a variable that belongs to a class rather than an instance of the class. It is shared among all instances of that class and is accessible using the class name itself. Class attributes are used to store data that is common to all instances of the class.

Code Snippet 9 shows an example to illustrate static fields or class attributes in Python.

Code Snippet 9:

```
class Dog:
    species = "Canine" # This is a class attribute

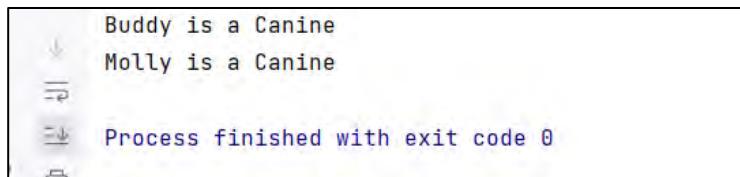
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Create instances of the Dog class
dog1 = Dog("Buddy", 3)
dog2 = Dog("Molly", 5)
```

```
# Access class attribute using the class name
print(f"{dog1.name} is a {Dog.species}")
print(f"{dog2.name} is a {Dog.species}")
```

Code Snippet 9 defines a `Dog` class with a class attribute `species` set to `Canine`. Instances of the class are created for two dogs, each with a `name` and `age`. The class attribute `species` is accessed using the class name `Dog`, and it is printed along with each dog's name. This demonstrates that the `species` attribute is shared among all instances of the `Dog` class.

Figure 8.9 depicts the output of this code when executed through PyCharm.



```
Buddy is a Canine
Molly is a Canine
Process finished with exit code 0
```

Figure 8.9: Output of Code snippet 9

Here, both `dog1` and `dog2` instances share the same `species` value because it is a class attribute. Modifying the class attribute will affect all instances.

Code Snippet 10 shows the modification of class attribute.

Code Snippet 10:

```
Dog.species = "Canis familiaris" # Modify the class attribute
print(f"{dog1.name} is a {Dog.species}")
print(f"{dog2.name} is a {Dog.species}")
```

Code Snippet 10 modifies the class attribute `species` of the `Dog` class to `Canis familiaris`. This demonstrates that modifying the class attribute affects all instances of the class, as both `dog1` and `dog2` now have the updated `species` value.

Figure 8.10 depicts the output of this code when executed through PyCharm.

```
Buddy is a Canis familiaris
Molly is a Canis familiaris
Process finished with exit code 0
```

Figure 8.10: Output of Code Snippet 10

8.7 Inner Class

In Python, an inner class is a class that is defined within the scope of another class. Inner classes are also sometimes referred to as nested classes. The outer class containing the inner class is often called the outer class or enclosing class.

Inner classes are used to logically group classes together when one class is closely related to another and would not make sense to be used outside of that context.

Code Snippet 11 shows an example of how to define and use an inner class in Python.

Code Snippet 11:

```
class Outer:
    def __init__(self):
        self.outer_attr = "Outer attribute"
        self.inner_instance = self.Inner()
    # Creating an instance of Inner class

    def outer_method(self):
        print("This is an outer method")

    class Inner:
        def __init__(self):
            self.inner_attr = "Inner attribute"

        def inner_method(self):
            print("This is an inner method")

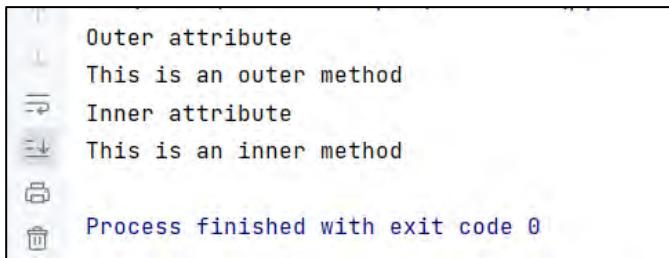
    # Creating an instance of the Outer class
    outer_instance = Outer()

    # Accessing outer class attributes and methods
    print(outer_instance.outer_attr)
    outer_instance.outer_method()
```

```
# Accessing inner class attributes and methods #through the
#inner instance
print(outer_instance.inner_instance.inner_attr)
outer_instance.inner_instance.inner_method()
```

In Code Snippet 11, `Inner` is an inner class within the `Outer` class. The `Inner` class is defined inside the scope of the `Outer` class, and it can access attributes and methods of the outer class. The outer class can also create instances of the inner class.

Figure 8.11 depicts the output of this code when executed through PyCharm.



The screenshot shows a PyCharm terminal window with the following output:

```
Outer attribute
This is an outer method
Inner attribute
This is an inner method
Process finished with exit code 0
```

Figure 8.11: Output of Code Snippet 11

8.8 Garbage Collection

Garbage collection in Python refers to the automatic management of memory by the Python interpreter to reclaim memory occupied by objects that are no longer required or referenced in the program. Python uses a built-in garbage collector to keep track of objects and free up memory that is no longer in use, helping to prevent memory leaks and manage memory efficiently.

Python's garbage collector works using called `reference counting` technique combined with cycle detection algorithm. The explanation of its functioning:

Reference Counting

Each object in Python has a reference count associated with it. A reference count represents the number of references (variables, attributes, and so on) pointing to that object. When an object's reference count drops to zero, it means there are no more references to that object and it becomes eligible for garbage collection.

Cycle Detection

While reference counting is effective in most cases, it cannot handle circular references, where objects reference each other in a cycle. To deal with circular references, Python's garbage collector uses a cycle detection algorithm that identifies and collects such cycles of objects.

Mark and Sweep

The garbage collector performs a process called mark and sweep. It starts by marking all objects that are reachable from the root objects (usually global variables and objects in the call stack). Then, it sweeps through the memory, freeing up objects that were not marked as reachable.

Code Snippet 12 shows an example of garbage collection in python.

Code Snippet 12:

```
import gc

class Person:
    def __init__(self, name):
        self.name = name
        print(f"Created instance for {self.name}")

    def __del__(self):
        print(f"Deleted instance for {self.name}")

# Create instances of the Person class
person1 = Person("Alice")
person2 = Person("Bob")

# Manually break reference to person2
person2 = None

# Force garbage collection
print("Collecting garbage...")
gc.collect()

# The program might have a slight pause here as #garbage
# collection is performed
print("Garbage collection done.")
```

Code Snippet 12 demonstrates manual garbage collection in Python using the `gc` module. Instances of the `Person` class are created, references are manually broken, and garbage collection is triggered to reclaim memory.

Figure 8.12 depicts the output of this code when executed through PyCharm.

```
Created instance for Alice
Created instance for Bob
Deleted instance for Bob
Collecting garbage...
Garbage collection done.
Deleted instance for Alice

Process finished with exit code 0
```

Figure 8.12: Output of Code Snippet 12

This output demonstrates the lifecycle of the `Person` instances and the working of the garbage collection process. Remember that this is just an example to illustrate how manual garbage collection can be triggered using the `gc` module. In most cases, Python's automatic garbage collection handles memory management effectively without manual intervention.

8.9 Summary

- A class in Python is a blueprint for creating objects that encapsulate data attributes and methods.
- An object in Python is an instance of a class, representing a specific entity with its own unique attributes and behaviors defined by the class.
- A constructor in Python is a special method within a class. It is used to initialize attributes and perform setup operations when creating an object instance.
- Methods in Python are functions defined within a class.
- A static field in Python, also known as a class attribute, is a variable shared among all instances of a class.
- An inner class in Python is a class defined within another class, allowing logical grouping of related functionality.
- Garbage collection in Python is an automatic memory management process.

8.10 Test Your Knowledge

1. Which of following statements accurately describes a class in Python?
 - a) A class is a built-in data type in Python used for storing collections of items
 - b) A class is a function in Python used to perform mathematical operations
 - c) A class is a blueprint that defines the structure and behavior of objects
 - d) A class is a special module used for importing external libraries in Python

2. What is an object in Python?
 - a) A reserved keyword used for defining variables
 - b) A built-in function used for input/output operations
 - c) A data type used for storing numerical values
 - d) An instance of a class that encapsulates data attributes and behaviors

3. What is the purpose of a constructor in Python?
 - a) To define a method for changing the data type of a variable
 - b) To create a new instance of a class with default attributes
 - c) To initialize the attributes of an object when it is created
 - d) To perform mathematical calculations within a class

4. What is a method in Python?
 - a) A built-in function that performs basic mathematical operations
 - b) A type of variable used for storing data within a class
 - c) A reserved keyword used for defining loops and conditional statements
 - d) A function defined within a class that operates on class attributes and provides specific functionality

5. What is a static field (class attribute) in Python?
 - a) A variable used for storing temporary data within a function
 - b) A reserved keyword used for defining static methods in a class
 - c) A variable shared among instances of a class, holding common data
 - d) A type of loop used to iterate through elements in a collection

6. What is an inner class in Python?
 - a) A class defined outside the scope of any other class
 - b) A class that inherits attributes from its outer class
 - c) A class defined within the scope of another class
 - d) A class that can only be accessed from within a function

7. What is the purpose of garbage collection in Python?
- a) To manually delete objects that are no longer required
 - b) To automatically release system resources used by Python programs
 - c) To remove unused code segments from the source code
 - d) To optimize the performance of mathematical calculations

8.10.1 Answers to Test Your Knowledge

1. A class is a blueprint that defines the structure and behavior of objects.
2. An instance of a class that encapsulates data attributes and behaviors.
3. To initialize the attributes of an object when it is created.
4. A function defined within a class that operates on class attributes and provides specific functionality.
5. A variable shared among instances of a class, holding common data.
6. Class defined within the scope of another class.
7. To automatically release system resources used by Python programs.

Try It Yourself

1. Create a Python class called `Person` with following characteristics:
 - a) It should have a constructor (`__init__`) that takes two parameters: `name` and `age` and initializes instance variables with these values.
 - b) It should have a method called `introduce` that prints a message such as 'Hello, my name is [name] and I am [age] years old.'

Create an instance of the `Person` class and call the `introduce` method to introduce the person.
2. Enhance the `Person` class from Question 1 following methods:
 - a) A method called `have_birthday` that increments the person's age by 1.
 - b) A method called `change_name` that takes a new name as a parameter and updates the person's name.

Create an instance of the `Person` class, call `introduce`, have a few birthdays using the `have_birthday` method, change the person's name using `change_name`, and then call `introduce` again to see the updated information.
3. Add a static field called `total_people` to the `Person` class that keeps track of the total number of `Person` instances created. Update the constructor to increment this field each time a new `Person` instance is created. Print the `total_people` field to see how many `Person` instances have been created.
4. Create a class called `car` with a constructor that takes a make and model as parameters and initializes instance variables. Add a method called `start_engine` that prints 'Engine started.' Now, create several instances of the `car` class and store them in a list. Once done, remove some of the instances from the list. Observe how Python's garbage collection works when objects are no longer referenced.



SESSION 09

ADVANCED OOPS AND EXCEPTIONAL HANDLING

Learning Objectives

In this session, students will learn to:

- ◆ Explain the concept of Encapsulation
- ◆ Explain briefly about Inheritance
- ◆ Outline the concept of Polymorphism
- ◆ Describe about Abstraction in Python
- ◆ Identify Exception Handling
- ◆ Describe the concepts of Assertions and Logging in Python

9.1 Introduction to Encapsulation

Encapsulation refers to the bundling of data (attributes) and methods (functions) that operate on the data into a single unit, known as a class. The main goal of encapsulation is to hide the internal details of an object's implementation from the outside world and provide a clear interface for interacting with the object.

Key aspects of encapsulation include:

Data Hiding

Encapsulation provides the capability to control and oversee how external entities engage with the internal data stored within an object. By designating specific attributes as private, access and modification are restricted to within the class. This restriction prevents external code from directly altering the object's internal state, thereby upholding data integrity and security.

Getter and Setter

Encapsulation promotes the use of getter and setter methods to manage an object's attributes, ensuring controlled access and modification. Getters offer controlled access to private attribute values, while setters allow controlled changes, enabling validation, constraints, and logic for data interaction.

Information Hiding

Encapsulation promotes the idea of information hiding, which means that the internal implementation details of an object should not be exposed to the outside world. This provides a clear separation between the interface (public methods) that users of the class interact with, and the implementation details that can change without affecting the class's users.

Flexibility and Maintenance

Encapsulating the internal details of an object enables alterations to the implementation without impacting the code utilizing the object. This makes it easier to maintain and extend the codebase over time, as long as the public interface remains consistent.

By encapsulating attributes and providing controlled access through getter and setter methods, encapsulation ensures that the internal state of an object remains consistent and secure. This practice promotes code maintainability and makes it easier to debug and maintain the application over a period of time. Additionally, encapsulation enhances data integrity, as changes to attributes are made through methods that can implement checks and safeguards against incorrect data manipulation.

Code Snippet 1 shows an example of encapsulation in Python.

Code Snippet 1:

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

    def get_balance(self):
        return self.__balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

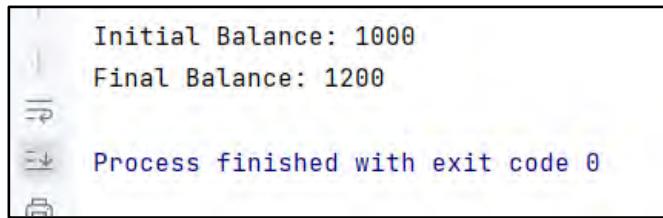
    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount

# Create an instance of the BankAccount class
account = BankAccount("123456", 1000)

# Accessing attributes indirectly through methods
print("Initial Balance:", account.get_balance())
account.deposit(500)
account.withdraw(300)
print("Final Balance:", account.get_balance())
```

In Code Snippet 1, the `account_number` and `balance` attributes are encapsulated by making them private (prefixed with double underscores). Access to these attributes is provided through getter and setter methods (`get_balance`, `deposit`, `withdraw`), enforcing controlled interaction with the object's data.

Figure 9.1 depicts the output of this code when executed through PyCharm.



A screenshot of a PyCharm terminal window. The output shows three lines of text: "Initial Balance: 1000", "Final Balance: 1200", and "Process finished with exit code 0". The terminal has a light gray background with dark gray text. There are small icons at the bottom left of the window.

Figure 9.1: Output of Code Snippet 1

9.2 Inheritance

Inheritance allows a new class (subclass or derived class) to inherit attributes and behaviors (methods and fields) from an existing class (superclass or base class). Inheritance promotes code reusability, extensibility, and the creation of specialized classes based on existing ones.

Key aspects of inheritance are as follows:

Superclass and Subclass Relationship: The superclass is the class being inherited from and the subclass is the class inheriting from the superclass. The subclass can access all public attributes and methods of the superclass, effectively extending its functionality.

Syntax: Creating a subclass involves defining it with the `class` keyword, followed by the subclass name and the superclass name enclosed in parentheses. Its syntax is given below.

```
class SubclassName(SuperclassName):  
    # class definition
```

Attributes and Methods Inheritance: Subclasses inherit attributes (data members) and methods (functions) from the superclass. Subclasses can also define their own attributes and methods, potentially overriding or extending those inherited from the superclass.

Method Overriding: Subclasses have the option to override (modify) methods inherited from the superclass by providing a new implementation. This allows subclasses to customize behavior while maintaining the same method signature.

Inheritance Hierarchy: Inheritance can create a hierarchy of classes with multiple levels of inheritance. Subclasses can themselves become superclasses for further subclasses, forming a chain of inheritance.

Access Control: Subclasses can access public attributes and methods of the superclass. However, private attributes are not inherited, as they are meant to be encapsulated within the defining class.

Code Snippet 2 shows an example of Inheritance.

Code Snippet 2:

```
class Vehicle:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def get_info(self):
        return f"Make: {self.make}, Model: {self.model}"

class Car(Vehicle):
    def __init__(self, make, model, year):
        super().__init__(make, model)
        self.year = year

    def get_info(self):
        return f"Year: {self.year}, {super().get_info()}""

class ElectricCar(Car):
    def __init__(self, make, model, year, battery_capacity):
        super().__init__(make, model, year)
        self.battery_capacity = battery_capacity

    def get_info(self):
        return f"Battery Capacity: {self.battery_capacity}, {super().get_info()}""

# Creating instances of the classes
car1 = Car("Toyota", "Camry", 2022)
electric_car1 = ElectricCar("Tesla", "Model 3", 2022, "75 kWh")

# Accessing the information using overridden methods
print(car1.get_info())
print(electric_car1.get_info())
```

In Code Snippet 2, the `Vehicle` class is the superclass with a `make` and `model` attribute and a method to get basic information.

The `Car` class is a subclass of `Vehicle`, with an added `year` attribute and an overridden `get_info` method.

The `ElectricCar` class is a subclass of `Car`, adding a `battery_capacity` attribute and overriding the `get_info` method.

Figure 9.2 depicts the output of this code when executed through PyCharm.

```
Year: 2022, Make: Toyota, Model: Camry
Battery Capacity: 75 kWh, Year: 2022, Make: Tesla, Model: Model 3
Process finished with exit code 0
```

Figure 9.2: Output of Code Snippet 2

9.3 Polymorphism

Polymorphism describes the capacity of various classes or objects to react to a shared method name in unique manners. This facilitates the treatment of objects from different classes as instances of a shared superclass. It fosters adaptability, reusability of code, and the concept of abstraction.

9.3.1 Compile-time (or Static) Polymorphism

Compile-time polymorphism in Python, often referred to as method overloading, is a concept where multiple methods in a class share the same name, but have different parameter lists. The appropriate method to be executed is determined by the number or types of arguments provided during the method call. This allows for selection of the most suitable method at compile time based on the method signature, providing flexibility in method invocation.

Code Snippet 3 shows an example that demonstrates a form of method overloading using default arguments.

Code Snippet 3:

```
class MathOperations:
    def add(self, a=None, b=None, c=None):
```

```

        if c is not None:
            return a + b + c
        elif b is not None:
            return a + b
        elif a is not None:
            return a
        else:
            return 0

# Create an instance of the MathOperations class
math = MathOperations()

# Method overloading using default arguments
print(math.add())          # Output: 0
print(math.add(5))         # Output: 5
print(math.add(5, 10))      # Output: 15
print(math.add(5, 10, 20))   # Output: 35

```

Code Snippet 3 demonstrates method overloading using default arguments in Python. The `add()` method of the `MathOperations` class can accept 0, 1, 2, or 3 arguments. When called with different numbers of arguments, it returns the sum of those arguments.

9.3.2 Runtime (or Dynamic) Polymorphism

Runtime polymorphism, also known as dynamic polymorphism or method overriding, is a key concept in object-oriented programming. It refers to the ability of objects of different classes to respond to the same method name in a way that is specific to their individual class. This behavior is determined at runtime, based on the actual type of the object invoking the method.

In Python, runtime polymorphism is achieved through method overriding. When a subclass provides its own implementation of a method that is already defined in its superclass, the subclass's method overrides the superclass's method. This allows objects of different classes to use the same method name while exhibiting behavior that is tailored to their specific class.

Some of the important factors about Runtime or Method Overriding are as follows:

Method Overriding: In Python, runtime polymorphism is achieved through method overriding. When a subclass provides its own implementation of a method that is already defined in its superclass, the subclass's method overrides

the superclass's method. This allows objects of different classes to use the same method name while exhibiting behavior that is tailored to their specific class.

super() Function: The `super()` function is often used in the subclass's overridden method to call the superclass's version of the method before adding or modifying behavior.

Inheritance Requirement: For runtime polymorphism to occur, there should be a relationship between classes where a subclass inherits from a superclass.

Runtime polymorphism enables the creation of generic code, with behavior dictated by subclass instances during runtime. This feature enhances the reuse of code, simplifies maintenance, and promotes abstraction within the realm of object-oriented programming.

Code Snippet 4 shows an example of method overriding in Python.

Code Snippet 4:

```
class Country:
    def official_language(self):
        return "No official language specified"

class USA(Country):
    def official_language(self):
        return "English"

class China(Country):
    def official_language(self):
        return "Mandarin"

class Germany(Country):
    def official_language(self):
        return "German"

# Creating instances of the subclasses
country = Country()
usa = USA()
china = China()
germany = Germany()

# Calling the overridden official_language method
```

```
print(country.official_language()) # Output: No official  
language specified  
print(usa.official_language())      # Output: English  
print(china.official_language())    # Output: Mandarin  
print(germany.official_language())  # Output: German
```

Code Snippet 4 defines a base class `Country` with a method `official_language()` that returns 'No official language specified.' Subclasses `USA`, `China` and `Germany` inherit from the `Country` class and override the `official_language()` method to return the respective official languages of each country. Instances of the subclasses are created, and the overridden `official_language()` method is called for each instance to print the official language of each country.

9.4 Abstraction

In Python programming, abstraction is a fundamental principle of OOP where intricate real-world entities are depicted using simplified representations within software. This approach concentrates on delineating the vital attributes and functionalities of an object while concealing extraneous intricacies. Abstraction facilitates the creation of a distinct division between the user-facing interface and the concealed implementation specifics.

Abstraction holds significant importance in Python due to its role in simplifying complex systems and promoting efficient code design. It allows developers to focus on the essential aspects of an object's behavior, shielding them from unnecessary implementation intricacies. By creating a clear separation between the interface and underlying details, abstraction enhances code readability, maintainability, and reusability. This aids in managing the complexity of large-scale applications, facilitating collaborative development, and minimizing errors.

Listed are some of the important aspects of abstraction in Python.

Simplified Representation: Abstraction involves creating simplified models of real-world entities, focusing only on the essential attributes and behaviors while omitting irrelevant details.

Hiding Complexity: Abstraction allows developers to hide the intricate implementation details of an object, exposing only the necessary interfaces and functionalities.

Code Reusability: By providing a concise and standardized interface, abstraction enhances code reusability across different parts of an application or even across different projects.

Clear Interface: It promotes the creation of clear and well-defined interfaces that users interact with, reducing confusion and making the code more user-friendly.

High-Level Design: It plays a crucial role in the high-level design of applications, allowing developers to focus on the overall architecture without getting bogged down in low-level details.

Efficiency: By simplifying complex entities, abstraction aids in improving performance and memory usage by minimizing unnecessary overhead.

Code Snippet 5 shows an example of Abstraction in Python.

Code Snippet 5:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

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

    def area(self):
        return 3.14 * self.radius * self.radius

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

```

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

# Creating instances of the subclasses
circle = Circle(5)
rectangle = Rectangle(4, 6)

# Calculating and printing the areas
print("Area of Circle:", circle.area())
print("Area of Rectangle:", rectangle.area())

```

Code Snippet 5 defines an abstract base class `Shape` with an abstract method `area()`, which must be implemented by its subclasses.

Subclasses `Circle` and `Rectangle` inherit from `Shape` and implement the `area()` method to calculate the area of a circle and a rectangle, respectively.

Instances of the `Circle` and `Rectangle` classes are created, and their `area()` methods are called to calculate and print the areas of a circle and a rectangle.

Figure 9.3 depicts the output of this code when executed through PyCharm.

The screenshot shows a PyCharm terminal window. It displays two lines of text: "Year: 2022, Make: Toyota, Model: Camry" and "Battery Capacity: 75 kWh, Year: 2022, Make: Tesla, Model: Model 3". Below these lines, there are three small icons: a double arrow, a plus sign, and a minus sign. At the bottom of the window, the text "Process finished with exit code 0" is visible.

Figure 9.3: Output of Code Snippet 5

In this example, the `Shape` class is defined as an Abstract Base Class (ABC) with an abstract method `area()`.

The `Circle` and `Rectangle` classes are subclasses of `Shape` and provide their own implementations of the `area()` method.

9.5 Exception Handling

Exception handling in Python is a mechanism that facilitates the management of errors and exceptional situations that may arise during program execution. When an error occurs in a Python program, it can lead to termination of the program if not properly handled. Exception handling provides a structured way to catch, handle, and recover from errors, ensuring that the program continues executing even in the presence of unexpected issues.

Key concepts of exception handling in Python are as follows:

- i. **Exception:** An exception is an event that occurs during program execution, resulting in the interruption of the normal flow of code. Common exceptions include `ZeroDivisionError`, `TypeError`, `FileNotFoundException`, and so on.

Code Snippet 6 shows an example of basic exception in Python.

Code Snippet 6:

```
def divide(a, b):
    if b == 0:
        return "Error: Division by zero"
    return a / b

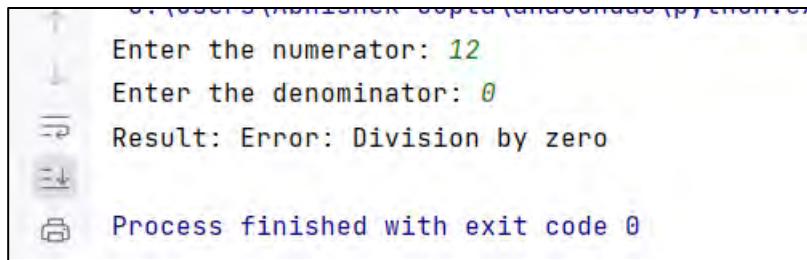
# Main program
numerator = input("Enter the numerator: ")
denominator = input("Enter the denominator: ")

if not numerator.isdigit() or not denominator.isdigit():
    print("Error: Invalid input. Please enter valid numbers.")
else:
    numerator = float(numerator)
    denominator = float(denominator)

    result = divide(numerator, denominator)
    print("Result:", result)
```

Code Snippet 6 shows a Python script which prompts the user to input a numerator and denominator. It then divides the numerator by the denominator, handling division by zero and invalid input gracefully, and prints the result or error message accordingly.

Figure 9.4 depicts the output of this code when executed through PyCharm.



```
Enter the numerator: 12
Enter the denominator: 0
Result: Error: Division by zero
Process finished with exit code 0
```

Figure 9.4: Output of Code Snippet 6

- ii. **Try-Except Block:** The try block is used to enclose the code that might raise an exception. If an exception occurs within the try block, the program flow is immediately transferred to the corresponding except block.

Code Snippet 7 shows a basic example of try-except block in Python.

Code Snippet 7:

```
def divide(a, b):
    try:
        result = a / b
        return result
    except ZeroDivisionError:
        return "Error: Division by zero"
    except TypeError:
        return "Error: Invalid data types for division"

# Main program
try:
    numerator = float(input("Enter the numerator: "))
    denominator = float(input("Enter the denominator: "))

    result = divide(numerator, denominator)
    print("Result:", result)
except ValueError:
    print("Error: Invalid input. Please enter valid numbers.")
except KeyboardInterrupt:
    print("\nProgram terminated by user.")
except Exception as e:
    print(f"An error occurred: {e}")
```

Code Snippet 7 shows Python script that divides two numbers provided by the user, handling division by zero and invalid input. It then prints the result or appropriate error messages. Exception handling is utilized for specific error cases such as invalid input or keyboard interrupts.

Figure 9.5 depicts the output of this code when executed through PyCharm.

The screenshot shows a terminal window from PyCharm. The command line path is 'C:\Users\ADMINISTER\Desktop\PycharmProjects\pytut'. The terminal displays the following text:
Enter the numerator: 23
Enter the denominator: 0
Result: Error: Division by zero
Process finished with exit code 0

Figure 9.5: Output of Code Snippet 7

- iii. **Except Clause:** The except block follows the try block and contains the code that should execute when a specific exception occurs. It is feasible to employ multiple except blocks for the purpose of dealing with different sorts of exceptions. This technique permits the handling of various types of errors that could arise within a try block, facilitating specific and targeted error management.

Code Snippet 8 shows a basic example on except clause in Python.

Code Snippet 8:

```
def read_and_divide(filename):
    try:
        with open(filename, 'r') as file:
            numerator = float(file.readline())
            denominator = float(file.readline())
            result = numerator / denominator
            return result
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found.")
    except ZeroDivisionError:
        print("Error: Division by zero is not allowed.")
    except ValueError:
```

```

        print("Error: Invalid data in the file. Please ensure
numeric values.")
    except Exception as e:
        print(f"An error occurred: {e}")

# Main program
file_name = input("Enter the name of the file: ")

result = read_and_divide(file_name)
if result is not None:
    print("Result:", result)

```

Code Snippet 8 shows Python script that reads two numbers from a file specified by the user, divides them, and prints the result. It handles various exceptions such as file not found, division by zero, invalid data in the file, and general exceptions. If successful, it prints the result of the division.

num.txt:

```

5
2

```

Figure 9.6 depicts the output of this code when executed through PyCharm.

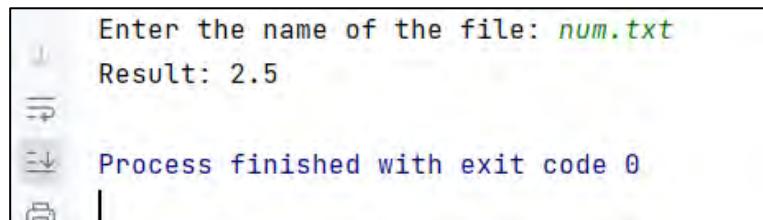


Figure 9.6: Output of Code Snippet 8

iv. **Handling Multiple Exceptions:** You can use multiple except blocks to handle different exceptions individually. This prevents your program from crashing due to unhandled exceptions.

Code Snippet 9 shows a basic example on Handling Multiple Exceptions in Python.

Code Snippet 9:

```
class BookInventory:
    def __init__(self):
        self.books = {"Python Basics": 10, "Data Science Handbook": 5, "Algorithms Unlocked": 3}

    def borrow_book(self, book_title):
        try:
            if self.books[book_title] > 0:
                self.books[book_title] -= 1
                print(f"You have borrowed '{book_title}' .")
            else:
                print(f"Sorry, '{book_title}' is out of stock .")
        except KeyError:
            print(f"Sorry, '{book_title}' is not available in our inventory .")
        except Exception as e:
            print(f"An error occurred: {e} ")

# Main program
inventory = BookInventory()

while True:
    book_title = input("Enter the title of the book you want to borrow: ")
    if book_title.lower() == "exit":
        break
    inventory.borrow_book(book_title)
```

Code Snippet 9 shows class BookInventory maintains a dictionary of book titles and their quantities. The borrow_book() method decrements the quantity when a book is borrowed and prints a success message or an appropriate error message if the book is out of stock or not in the inventory. In the main program, the user inputs a book_title to borrow. The loop continues until the user enters exit, allowing continuous borrowing from the inventory.

Figure 9.7 depicts the output of this code when executed through PyCharm.

```

Enter the title of the book you want to borrow: DBMS
Sorry, 'DBMS' is not available in our inventory.
Enter the title of the book you want to borrow: Python
Sorry, 'Python' is not available in our inventory.
Enter the title of the book you want to borrow: Algorithms Unlocked
You have borrowed 'Algorithms Unlocked'.
Enter the title of the book you want to borrow: Python Basics
You have borrowed 'Python Basics'.
Enter the title of the book you want to borrow: Data Science Handbook
You have borrowed 'Data Science Handbook'.
Enter the title of the book you want to borrow: exit

Process finished with exit code 0

```

Figure 9.7: Output of Code Snippet 9

- v. **Finally Block:** Optionally, you can include a finally block after try and except blocks. The code in the finally block is executed regardless of whether an exception was raised or not.

Code Snippet 10 shows a basic example on Finally Block in Python.

Code Snippet 10:

```

class BikeRental:
    def __init__(self, total_bikes):
        self.total_bikes = total_bikes
        self.available_bikes = total_bikes

    def rent_bike(self):
        try:
            if self.available_bikes > 0:
                print("You have rented a bike.")
                self.available_bikes -= 1
            else:
                print("Sorry, no bikes available for rent.")
        except Exception as e:
            print(f"An error occurred: {e}")
        finally:
            print("Thank you for using our bike rental service!")

# Main program
rental_shop = BikeRental(total_bikes=10)

```

```

while True:
    user_input = input("Press 'r' to rent a bike or 'q' to quit:
    ")
    if user_input.lower() == 'q':
        break
    elif user_input.lower() == 'r':
        rental_shop.rent_bike()
    else:
        print("Invalid input. Please enter 'r' or 'q'.")

```

Code Snippet 10 shows a Python class `BikeRental` manages the rental of bikes, with the ability to rent a bike if available and decrement the available bikes count. It ensures availability is checked before renting and provides appropriate feedback. The main program interacts with users, allowing them to rent bikes (`r`) or quit (`q`).

The try-except-finally block ensures error handling and concludes with a thank you message regardless of the outcome.

Figure 9.8 depicts the output of this code when executed through PyCharm.

```

main
C:\Users\Username\PycharmProjects\pythonProject\venv\Scripts\python.exe
Press 'r' to rent a bike, 'q' to quit, or 'c' to return a bike: r
You have rented a bike.
Press 'r' to rent a bike, 'q' to quit, or 'c' to return a bike: c
Thank you for returning the bike.
Press 'r' to rent a bike, 'q' to quit, or 'c' to return a bike: q

```

Figure 9.8: Output of Code Snippet 10

vi. Keyboard Interrupt: When a keyboard interrupt is triggered, the Python interpreter raises a built-in exception called `KeyboardInterrupt`. This exception can be caught and handled just as any other exception using a try and except block, allowing the program to perform some cleanup or take specific actions before terminating.

The example shows a basic syntax on keyboard interrupt in Python.

```

try:
    while True:
        pass # This loop will run indefinitely until
             # a Keyboard Interrupt is encountered

```

```
except KeyboardInterrupt:  
    print("\nKeyboard Interrupt detected. Exiting.")
```

Code Snippet 11 shows an example on exception handling in Python.

Code Snippet 11:

```
def divide(a, b):  
    try:  
        result = a / b  
        return result  
    except ZeroDivisionError:  
        return "Error: Division by zero"  
    except TypeError:  
        return "Error: Invalid data types for division"  
    except Exception as e:  
        return f"An error occurred: {e}"  
  
def read_integer():  
    try:  
        num = int(input("Enter an integer: "))  
        return num  
    except ValueError:  
        print("Invalid input. Please enter a valid integer.")  
        return None  
  
def open_file(filename):  
    try:  
        file = open(filename, 'r')  
        content = file.read()  
        file.close()  
        return content  
    except FileNotFoundError:  
        return "Error: File not found"  
    except Exception as e:  
        return f"An error occurred while reading the file: {e}"  
  
# Main program  
try:  
    numerator = read_integer()  
    denominator = read_integer()  
  
    if numerator is None or denominator is None:  
        exit()  
  
    result = divide(numerator, denominator)
```

```

print("Division result:", result)

filename = input("Enter the name of the file to read: ")
file_content = open_file(filename)
print("File content:", file_content)
except KeyboardInterrupt:
    print("\nProgram terminated by user.")
except Exception as e:
    print(f"An error occurred: {e}")
finally:
    print("Program execution completed.")

```

Code Snippet 11 shows a Python script that defines functions for division, reading an integer, and opening a file. It handles various exceptions such as division by zero, invalid data types, and file not found. In the main program, it prompts users for input, performs division, reads a file, and displays results or errors. Finally, it prints a completion message after execution.

num.txt:

```

5
2

```

Figure 9.9 depicts the output of this code when executed through PyCharm.

```

Enter an integer: 12
Enter an integer: 2
Division result: 6.0
Enter the name of the file to read: num.txt
File content: 5
2
Program execution completed.

Process finished with exit code 0

```

Figure 9.9: Output of Code Snippet 11

9.6 Assertion and Logging

9.6.1 Assertion in Python

Assertions in Python are a debugging aid that performs a sanity check in your code. An assertion is a statement that asserts a condition to be true. If the

condition is false, an exception (`AssertionError`) is raised, indicating that something is wrong with your code and requires attention.

Assertions are typically used during development and testing to catch logical errors and ensure that assumptions about the code's behavior are valid. They help to identify problems early in the development process.

Code Snippet 12 will explain how assertion can be used in Python.

Code Snippet 12:

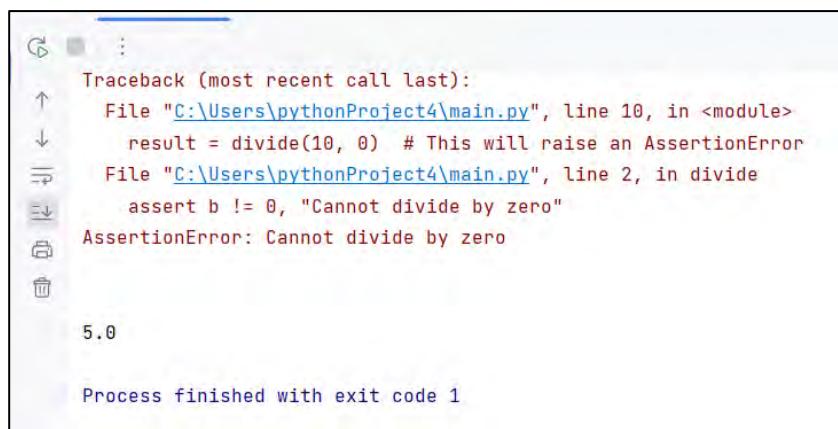
```
def divide(a, b):
    assert b != 0, "Cannot divide by zero"
    return a / b

result = divide(10, 2)  # This is valid
print(result)

result = divide(10, 0)  # This will raise an AssertionError
print(result)
```

Code Snippet 12 defines a `divide` function performs division of two numbers, `a` and `b`, ensuring `b` is not zero with an assertion. Upon calling `divide(10, 0)`, an `AssertionError` is raised due to division by zero, halting execution before printing any result.

Figure 9.10 depicts the output of this code when executed through PyCharm.



The screenshot shows a PyCharm interface with a terminal window. The terminal displays the following output:

```
Traceback (most recent call last):
  File "C:\Users\pythonProject4\main.py", line 10, in <module>
    result = divide(10, 0)  # This will raise an AssertionError
  File "C:\Users\pythonProject4\main.py", line 2, in divide
    assert b != 0, "Cannot divide by zero"
AssertionError: Cannot divide by zero

5.0

Process finished with exit code 1
```

Figure 9.10: Output of Code Snippet 12

9.6.2 Logging in Python

Logging in Python involves recording events, messages, and other relevant information during the execution of a program. Instead of printing messages to the console, logging provides a more structured and versatile approach to capturing information about a program's behavior.

The logging module in Python provides a robust logging framework. Configurable log levels such as DEBUG, INFO, WARNING, ERROR, and CRITICAL can be utilized to classify the significance of logged messages. This enables the regulation of the level of detail to be recorded in the logs.

Code Snippet 13 shows an example for Logging in Python.

Code Snippet 13:

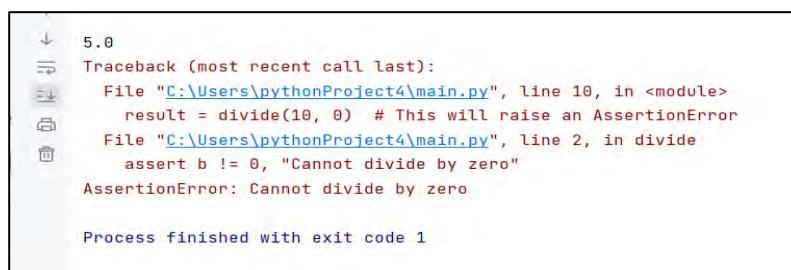
```
import logging

# Configure logging
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')

def divide(a, b):
    logging.debug(f"Dividing {a} by {b}")
    result = a / b
    logging.info(f"Result: {result}")
    return result

result = divide(10, 2)
result = divide(10, 0)      # This will generate a
#ZeroDivisionError, and the log messages will capture it
```

Figure 9.11 depicts the output of this code when executed through PyCharm.



```
5.0
Traceback (most recent call last):
  File "C:\Users\pythonProject4\main.py", line 10, in <module>
    result = divide(10, 0) # This will raise an AssertionError
  File "C:\Users\pythonProject4\main.py", line 2, in divide
    assert b != 0, "Cannot divide by zero"
AssertionError: Cannot divide by zero

Process finished with exit code 1
```

Figure 9.11: Output of Code Snippet 13

In this instance, the logging messages incorporate timestamps, log levels, and the actual content of the messages. The logging level can be modified to regulate which messages are captured. For instance, configuring the level to `logging.INFO` will display only the info and higher-level messages in the output.

Assertions and logging serve distinct purposes in Python development. Assertions aid in identifying errors during development and testing. Logging offers a structured method for recording information regarding your program's behavior, serving purposes such as debugging and monitoring.

9.7 Summary

- Encapsulation is the principle of bundling data and methods that operate on that data into a single unit.
- Inheritance involves creating subclasses that inherit attributes and methods from a superclass.
- Polymorphism allows objects of different classes to respond to the same method name in distinct ways.
- Abstraction simplifies complex entities by focusing on essential attributes and behaviors.
- Exception handling is a mechanism to catch and handle errors during program execution. It prevents crashes, and allows graceful recovery.
- Assertions are checks for logical correctness during development while logging captures runtime information in a structured manner.

9.8 Test Your Knowledge

1. Which concept in Python involves bundling data and methods that operate on that data into a single unit, while restricting direct access to the internal details from outside the unit?
A) Inheritance
B) Polymorphism
C) Abstraction
D) Encapsulation

2. What is polymorphism in object-oriented programming?
A) A design pattern used for creating classes
B) A process of converting data types
C) The ability of objects to take on multiple forms
D) A technique to define private methods in a class

3. What is abstraction in object-oriented programming?
A) A technique for creating objects from classes
B) The process of hiding complex implementation details and showing only the necessary features of an object
C) A mechanism for defining multiple methods with the same name but different implementations
D) The process of converting data from one type to another

4. Which of the following is not a standard exception in Python?
a) ValueError
b) TypeError
c) NumberError
d) FileNotFoundError

5. What are assertions and logging used for in Python?

- A) Assertions are used to create custom error messages, while logging is used to handle input/output operations
- B) Assertions are used to print debug information, while logging is used for controlling program flow
- C) Assertions are used to terminate the program when an error occurs, while logging is used for recording information and debugging messages
- D) Assertions are used to optimize code execution, while logging is used for handling file operations

9.8.1 Answers to Test Your Knowledge

1. Encapsulation
2. The ability of objects to take on multiple forms.
3. The process of hiding complex implementation details and showing only the necessary features of an object.
4. NumberEfigurerror
5. Assertions are used to terminate the program when an error occurs, while logging is used for recording information and debugging messages.

Try It Yourself

1. Create a program that models a simple zoo.
 - a) Create a base class called `Animal` with following attributes: `name`, `species`, and `sound`.
 - b) Create a constructor for the `Animal` class that initializes these attributes.
 - c) Create a method in the `Animal` class called `make_sound` that prints the sound of the animal.
 - d) Create two subclasses: `Mammal` and `Bird`, which inherit from the `Animal` class.
 - e) Add an additional attribute to the `Mammal` class called `num_legs`.
 - f) Add an additional attribute to the `Bird` class called `can_fly`.
 - g) Create constructors for both `Mammal` and `Bird` classes that initialize their respective attributes and call the constructor of the base class (`Animal`).
 - h) Override the `make_sound` method in both the `Mammal` and `Bird` classes to print an appropriate sound for each animal type.
 - i) Create instances of both `Mammal` and `Bird` classes and demonstrate the use of the `make_sound` method.
2. Create a program that calculates the average of a list of numbers.
 - a) Prompt the user to enter a list of numbers (separated by spaces).
 - b) Split the user input into individual numbers and store them in a list.
 - c) Use a `try` block to handle exceptions that may occur during the input process, such as `invalid input` or `empty input`.
 - d) Write a function called `calculate_average` that takes the list of numbers as an argument and calculates the average.
 - e) Handle exceptions that may occur during the calculation, such as `division by zero`.
 - f) Display the calculated average to the user.



SESSION 10

SERIALIZATION AND DESERIALIZATION

Learning Objectives

In this session, students will learn to:

- ◆ Explain the concept of serialization in Python
- ◆ Describe the purpose of deserialization in Python
- ◆ Distinguish between serialization and deserialization processes
- ◆ Identify scenarios where serialization and deserialization are used in Python
- ◆ Explain the importance of exceptions and error handling in serialization and deserialization

10.1 Introduction to Serialization

Serialization is the process of converting complex data structures, objects, or data types into a structured format that can be easily stored, transmitted, or reconstructed. In Python, serialization is commonly used to transform data into a format such as JavaScript Object Notation (JSON) or pickle, which allows data to be saved to files, transferred over networks, or stored in databases.

Serialization is essential when it is required to persistently store data or exchange information between different systems, programming languages, or components. It ensures that the data remains intact and can be reconstructed accurately.

10.1.1 Serialization Data Transformation into Structured Format

Data Structure Conversion: When serializing data, the process begins with a complex data structure, such as dictionaries, lists, classes, or objects. This data structure contains various data types, nested structures, and references.

Encoding Data: The data is encoded or transformed into a format that is easy to represent and transmit. Common serialization formats in Python include JSON, pickle, XML, and YAML. Among these, JSON and pickle are widely used.

JSON Serialization: JSON is a human-readable text-based format that is also easy for machines to parse. When you serialize data into JSON, Python dictionaries, lists, strings, numbers, and boolean values are translated into their JSON equivalents. JSON represents data using key-value pairs and arrays.

Pickle Serialization: Pickle is a Python-specific serialization module. It can handle more complex Python-specific objects, including custom classes and their instances. Pickle serializes data into a binary format that can be used to recreate the original Python objects.

Structural Format: The serialized data now exists in a structured format, represented as a string (in case of JSON) or binary data (in case of pickle). This format is compact and suitable for storage in files or transmission over networks.

Storage and Transmission: The serialized data can be saved to a file, sent over a network, or stored in a database. JSON files, for example, can be easily read by other programming languages as well. It makes it a popular choice for cross-language communication.

Serialization provides a way to maintain data integrity. It facilitates data sharing, and enable communication between different components of a system.

However, it is important to choose the appropriate serialization format based on factors such as human readability, cross-language compatibility, and security.

Code Snippet 1 shows a basic example that demonstrates serialization using the JSON format.

Code Snippet 1:

```
import json

# Sample data in a Python dictionary
data = {
    "name": "John",
    "age": 30,
    "city": "New York"
}

# Serialize the data to JSON format
serialized_data = json.dumps(data, indent=4)

# Print the serialized data
print("Serialized Data:")
print(serialized_data)

# Save the serialized data to a file
with open("data.json", "w") as file:
    file.write(serialized_data)

print("Data saved to 'data.json' file.")
```

In Code Snippet 1, the code converts a Python dictionary into JSON format using `json.dumps()`. It then prints the serialized data and saves it to a file named `data.json`. This Snippet demonstrates the process of serializing Python data to JSON and saving it for storage or transmission.

Figure 10.1 depicts the output of this code when executed through PyCharm.

```
Serialized Data:  
{  
    "name": "John",  
    "age": 30,  
    "city": "New York"  
}  
Data saved to 'data.json' file.  
  
Process finished with exit code 0
```

Figure 10.1: Output of Code Snippet 1

10.2 Introduction to Deserialization

Deserialization in Python serves the purpose of reversing the process of serialization. It involves transforming serialized data, which is usually in a structured format such as `JSON` or `pickle`, back into its original, complex data structure. The primary goal of deserialization is to restore the data to a usable state, allowing it to be manipulated and processed as it was before serialization.

Structured Data Retrieval: The serialized data, which will be in the form of a string (for `JSON`) or binary data (for `pickle`), is obtained from a file, network transmission, or any storage medium.

Format Recognition: Depending on the serialization format used, the deserialization process requires recognizing the format's structure. For instance, `JSON` data is typically organized in key-value pairs or arrays, while `pickle` data contains binary-encoded representations of Python objects.

Data Reconstruction: Deserialization involves reassembling the serialized data into its original form. This means converting `JSON` strings back into dictionaries, lists, strings, and numbers, or recreating Python objects from `pickle` binary data.

Custom Object Handling: In case of more complex serialization formats such as `pickle`, deserialization must take into account how custom Python objects were serialized. The deserialization process requires the presence of original class definitions and methods for these objects to be accurately reconstructed.

Data Restoration: The deserialized data is now in a state that closely resembles the original data structure that was initially serialized. This data can now be utilized, processed, and manipulated as required within the Python program.

Code Snippet 2 shows a basic example that demonstrates deserialization using the JSON format.

Code Snippet 2:

```
import json

# Sample serialized data in JSON format
serialized_data = '''
{
    "name": "John",
    "age": 30,
    "city": "New York"
}
'''

# Deserialize the JSON data back into a Python dictionary
deserialized_data = json.loads(serialized_data)

# Print the deserialized data
print("Deserialized Data:")
print(deserialized_data)

# Access specific values in the dictionary
print("Name:", deserialized_data["name"])
print("Age:", deserialized_data["age"])
print("City:", deserialized_data["city"])
```

Code Snippet 2 deserializes JSON data into a Python dictionary using `json.loads()`. It prints the deserialized data and accesses specific values such as name, age, and city from the dictionary. Essentially, it demonstrates converting JSON into a Python dictionary for easy data manipulation and access.

Figure 10.2 depicts the output of this code when executed through PyCharm.

```

Deserialized Data:
{'name': 'John', 'age': 30, 'city': 'New York'}
Name: John
Age: 30
City: New York
Process finished with exit code 0

```

Figure 10.2: Output of Code Snippet 2

10.3 Comparison between Serialization and Deserialization

Table 10.1 displays the comparison between serialization and deserialization.

Aspect	Serialization	Deserialization
Purpose	Converts complex data into a structured, storable, or transferable format.	Reverses the serialization process, restoring data to its original form.
Primary Goal	Prepare data for storage, transmission, or sharing.	Restore serialized data to its original structure for manipulation.
Input Data	Complex data structures, objects, or data types in memory.	Serialized data in a structured format (example, JSON, pickle) from storage or transmission.
Output Data	Serialized data in a structured format (example, JSON, pickle).	Original data structure, matching what was serialized.
Data Integrity	Ensures that data remains intact, but loses specific object references (example, in JSON).	Restores data integrity, ensuring that objects and references are correctly reconstructed.
Usage Scenarios	<ul style="list-style-type: none"> - Data persistence (example, saving to files, databases). - Network communication (example, sending data between systems). - Cross-language interaction. 	<ul style="list-style-type: none"> - Retrieving previously serialized data. - Processing data received from external sources. - Data restoration for further manipulation.

Aspect	Serialization	Deserialization
Serialization Formats	<ul style="list-style-type: none"> - JSON, XML, YAML (for human-readable structured data). - Pickle (for Python-specific objects). - Protocol Buffers, Avro, and so on (for various use cases). 	Depends on the serialization format used. Formats such as JSON, pickle, XML, and so on have their deserialization methods.
Error Handling	Can raise exceptions during serialization (example, due to incompatible data types).	Can raise exceptions during deserialization (example, due to malformed data or missing objects). Proper error handling is essential.
Security Considerations	Be cautious when deserializing data from untrusted sources to prevent code execution vulnerabilities.	Verify and validate deserialized data to prevent security risks such as injection attacks.

Table 10.1: Comparison between Serialization and Deserialization

10.4 Utilization of Serialization and Deserialization

Serialization and deserialization are commonly used in various scenarios in Python applications.

Table 10.2 displays the common scenarios where serialization and deserialization are utilized.

Applications	Serialization	Deserialization
Data Persistence	Saving program data, configurations, or user preferences to files or databases for later retrieval.	Retrieving and loading previously saved data from storage.
Networking	Preparing data for transmission over networks, such as sending JSON payloads in HTTP requests or responses.	Receiving and processing data received over a network, converting it back into usable data structures.

Applications	Serialization	Deserialization
Inter-process Communication	Preparing data to be passed between different processes or components of an application.	Accepting and interpreting data received from other processes or components.
Web APIs	Converting data into JSON or XML formats when building RESTful APIs or other Web services.	Parsing incoming JSON or XML data from Web requests to extract and manipulate information.
Caching	Storing computed or fetched data in a cache to improve performance.	Retrieving and using cached data when required, restoring it to its original format.
Message Queues	Preparing messages for transmission in message queues such as RabbitMQ or Kafka.	Consuming and processing messages received from message queues.
Database Interaction	Preparing data to be stored in databases, especially NoSQL databases that accept JSON or similar formats.	Reading data from databases and converting it into usable objects or data structures.
Cross-Language Communication	Facilitating communication between Python applications and applications written in other programming languages by using common interchange formats such as JSON or protocol buffers.	Receiving data from external systems and converting it into Python data structures.
Object Persistence	Saving complex Python objects, including custom classes and their instances.	Loading previously saved Python objects, reconstructing them for use within the program.
Testing and Mocking	Preparing test data or mock data for unit testing and integration testing.	Parsing test data or mock data within test cases to simulate real data scenarios.

Table 10.2: Common Scenarios where Serialization and Deserialization are utilized

10.5 Standardized Data Formats for Serialization and Deserialization

Using standardized data formats such as `JSON` and Extensible Markup Language (`XML`) for serialization and deserialization offers several benefits, especially in terms of interoperability between different systems, programming languages, and platforms.

Cross-Platform Compatibility:

Standardized formats such as `JSON` and `XML` are platform-agnostic. It means they can be used across various operating systems and hardware architectures. This compatibility ensures that serialized data can be exchanged seamlessly between systems running on different platforms.

Language Neutrality:

`JSON` and `XML` are not tied to a specific programming language. They are supported by a wide range of programming languages, making it easier to share data between applications developed in different languages. This language neutrality promotes interoperability.

Human-Readable and Self-Descriptive:

Both `JSON` and `XML` are human-readable and self-descriptive. Developers can easily understand the data structure and content just by looking at the serialized data. This readability makes it simpler to troubleshoot and debug interoperability issues.

Robust Data Validation:

Standardized formats often come with built-in mechanisms for data validation and schema definition (a, `JSON Schema`, `XML Schema`). This ensures that serialized data adheres to a predefined structure and constraints, reducing the risk of data corruption or misinterpretation.

Versatility:

`JSON` and `XML` are versatile and can represent a wide range of data structures, including nested objects, arrays, and primitive data types. This flexibility allows for the serialization of diverse data formats, making them suitable for a broad spectrum of use cases.

10.6 Role of Exception and Error Handling in Serialization and Deserialization

The role of exceptions and error handling in serialization and deserialization processes is crucial for managing unexpected situations and ensuring the reliability of data conversion.

Handling Unexpected Data: During deserialization, unexpected data can cause errors, such as missing fields or incompatible data types. Error handling mechanisms, such as `try-catch` blocks, allow the program to gracefully respond to these issues by providing alternative actions or reporting errors to prevent crashes.

Version Compatibility: When dealing with serialized data, version mismatches between the serialized data and the deserialization code can occur due to updates or changes in data structures. Error handling helps manage versioning issues by detecting and accommodating differences to ensure a smooth transition.

Data Validation: Serialization and deserialization should include data validation steps to ensure the integrity of the data being processed. If validation fails, exceptions can be raised to indicate that the data is corrupted or does not meet the expected criteria.

Resource Management: Exception handling is vital for resource management, such as file handling during serialization and deserialization. It ensures that resources such as files are properly closed and released. Even if errors occur, there are no resource leaks.

Security Concerns: Exception handling is also important for security. It can help detect and respond to potential security threats. For instance, attempts to inject malicious data or exploit vulnerabilities in the deserialization process.

10.7 Summary

- Serialization is the process of converting data structures or objects into a format that can be easily stored, transmitted, or reconstructed.
- Deserialization is the process of reconstructing data structures or objects from a serialized format back into their original form.
- Serialization and deserialization enable efficient data storage, transmission, and interoperability by converting complex data structures into a portable format.
- Serialization and deserialization are utilized in various aspects such as data persistence, networking, inter-process communication, cross-language communication, Web API, and so on.
- Standardized data formats in serialization and deserialization are predefined structures that establish a consistent and universally recognized way of organizing and representing data.

10.8 Test Your Knowledge

1. What is serialization in Python?
 - A) The process of converting data into a format suitable for storage or transmission
 - B) The process of executing code serially
 - C) The process of converting a string into an integer
 - D) The process of encrypting data

2. Which Python module is commonly used for deserialization, especially for handling JSON data?
 - A) pickle
 - B) deserialize
 - C) json
 - D) unpickle

3. Serialization and deserialization primarily aim to make data storage, transmission, and interoperability more efficient and enable data exchange between systems.
 - A) True
 - B) False

4. What are the primary uses of Serialization and Deserialization in programming?
 - A) Serialization is used to execute code, while Deserialization is used for encryption
 - B) Serialization is used to convert data into a portable format, while Deserialization is used for database querying
 - C) Serialization is used for data storage, transmission, and interoperability, while Deserialization is used for code compilation
 - D) Serialization and Deserialization have no specific use in programming

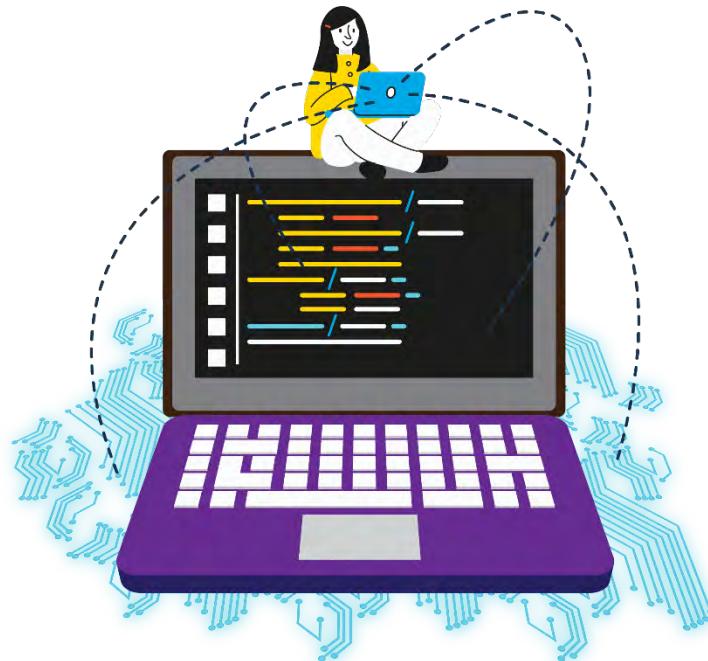
5. What is the primary purpose of using standardized data formats?
 - A) To make data more complex and difficult to interpret
 - B) To improve data security and encryption
 - C) To enable data exchange between different systems and ensure data consistency
 - D) To decrease data interoperability and restrict data sharing

10.8.1 Answers to Test Your Knowledge

1. The process of converting data into a format suitable for storage or transmission
2. json
3. True
4. Serialization is used for data storage, transmission, and interoperability, while Deserialization is used for code compilation
5. To enable data exchange between different systems and ensure data consistency

Try It Yourself

1. Create a program that serializes and deserializes a Python object using the `pickle` module.
 - a) Create a Python class called `Person` with attributes for a person's name and age.
 - b) Write a function called `serialize_person` that takes a `Person` object as input and serializes it using the `pickle` module. Save the serialized object to a file named '`person.pickle`'.
 - c) Write a function called `deserialize_person` that reads the serialized object from '`person.pickle`' and deserializes it, returning the `Person` object.
 - d) Prompt the user to enter a person's name and age.
 - e) Create a `Person` object with the provided information.
 - f) Use the `serialize_person` function to serialize the `Person` object.
 - g) Use the `deserialize_person` function to read and deserialize the object from the file.
 - h) Display the deserialized `Person` object's attributes to verify that the serialization and deserialization were successful.



SESSION 11

FILES AND THREADS

Learning Objectives

In this session, students will learn to:

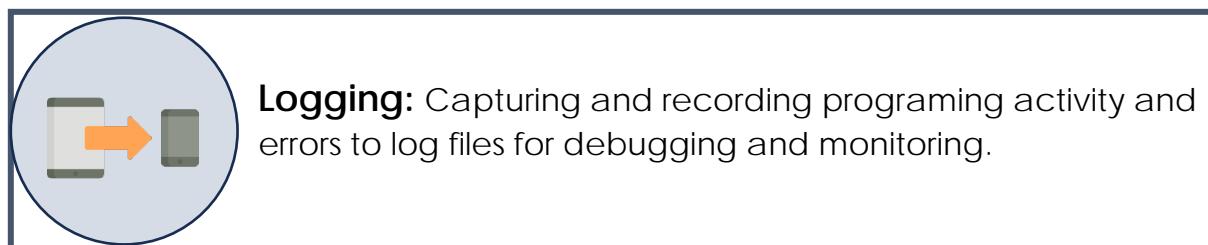
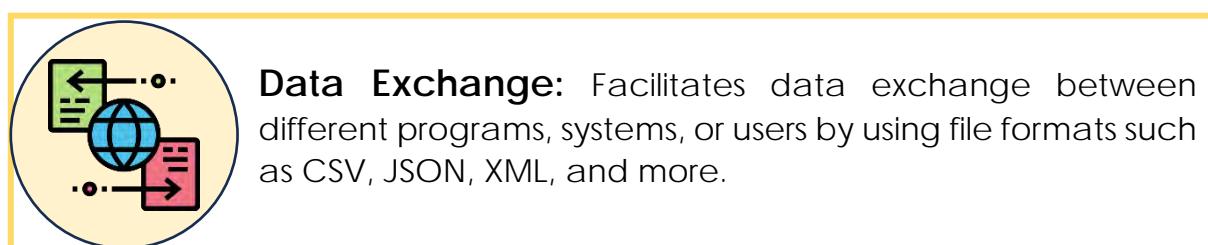
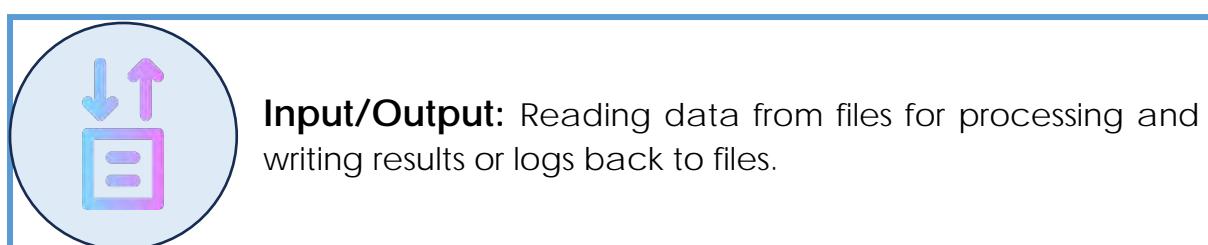
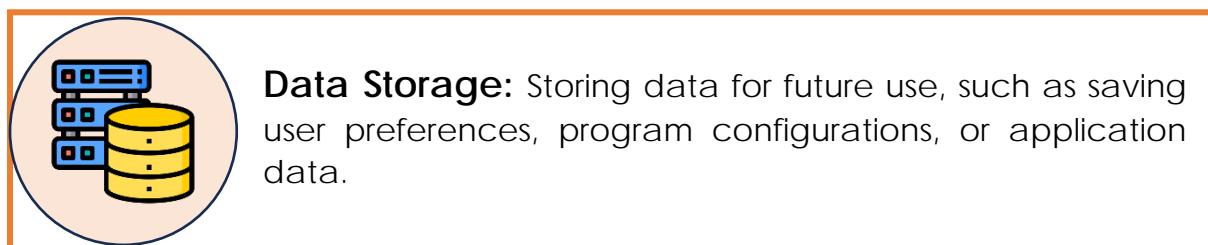
- ◆ Define File in Python programming
- ◆ Identify steps involved in reading and writing string in a File
- ◆ Explain the concept of pickle in Python
- ◆ Describe creating a Thread using both a Class and a Function
- ◆ Explain the concept of Multithreading in Python
- ◆ Identify the utilization of Thread in communication

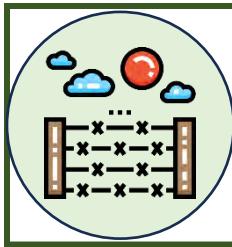
11.1 Introduction to File

In Python programming, a file is a named collection of data or information that is stored on computer's storage medium, such as a hard drive, solid-state drive, or network storage. Files can contain various types of data, including text, binary data, configuration settings, code, and so on.

11.1.1 Application of Files

Python provides several built-in functions and libraries for working with files, allowing programmers to perform various operations such as reading, writing, appending, and manipulating file contents. Files are a fundamental component of data storage and retrieval in Python and are often utilized for tasks such as:





Persistence: Storing and retrieving complex data structures, such as dictionaries, lists, and objects, for long-term use or sharing between program executions.

11.1.2 Classification of Files

Python categorizes files into different types depending on their content and access methods.



Text Files

Contain human-readable text and are opened using text mode (t). Common examples include .txt files and .csv files.



Binary Files

Contain non-textual data, such as images, audio, or serialized objects. These files are opened using binary mode (b).

Python provides built-in functions such as `open()`, `read()`, `write()`, `close()`, and libraries such as `os` and `shutil` to work with files efficiently. Proper file management and error handling are essential practices when dealing with files in Python to ensure data integrity and prevent issues such as data loss or corruption.

Code Snippet 1 shows a basic example of files using the built-in functions.

Code Snippet 1:

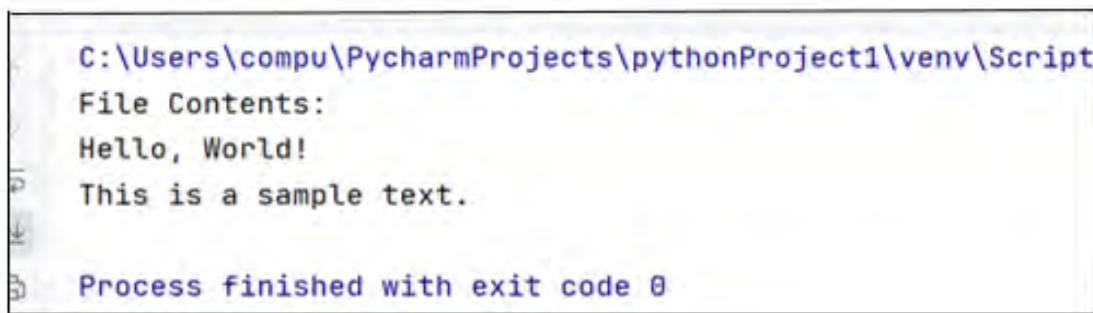
```
# Create and write to a text file
with open("example.txt", "w") as file:
    file.write("Hello, World!\n")
    file.write("This is a sample text.")

# Read from the text file
```

```
with open("example.txt", "r") as file:  
    file_contents = file.read()  
  
# Print the contents of the file  
print("File Contents:")  
print(file_contents)
```

In Code Snippet 1, the code creates a text file named 'example.txt' and writes two lines of text into it. Then, it reads the contents of the file and stores them in a variable called `file_contents`. Finally, it prints the contents of the file to the console.

Figure 11.1 depicts the output of this code when executed through PyCharm.



The screenshot shows the PyCharm terminal window. The command `C:\Users\compu\PycharmProjects\pythonProject1\venv\Script` is entered at the top. Below it, the output of the script is displayed:
File Contents:
Hello, World!
This is a sample text.
At the bottom, the message `Process finished with exit code 0` is shown.

Figure 11.1: Output of Code Snippet 1

11.2 Reading and Writing a String in a File

Reading and writing strings to a file in Python involves several steps. Following are the key steps for both reading from and writing to a file:

11.2.1 Writing a String to a File

Opening a File:

Utilize the `open()` function to open the file you want to write in. Specify the file path and the mode as `w` (write) to create a new file or overwrite an existing file, or `a` (append) to add content to an existing file.

Following example can be referred to open the file in write mode.

Example:

```
file_path = "example.txt"  
with open(file_path, 'w') as file:  
    # Write operations go here
```

Writing Data:

Utilize the `write()` method of the file object to write the string data to the file.

Following example can be referred to write into the file.

Example:

```
with open(file_path, 'w') as file:  
    file.write("Hello, World!\n")  
    file.write("This is a sample text.")
```

Closing a File:

Always close the file after writing to ensure that changes are saved and resources are freed. This can be done automatically using the 'with' statement, as shown in the previous code.

Code Snippet 2 shows an example to write a string to a file in Python.

Code Snippet 2:

```
# Open a file in write mode ('w')  
with open('example.txt', 'w') as file:  
    # Write a string to the file  
    file.write('This is a string that will be written to the  
file.')
```

In Code Snippet 2, it opens a file named 'example.txt' in write mode (`w`) and writes the specified string to it. The 'with' statement ensures that the file is properly closed after writing, even if an error occurs during the process.

11.2.2 Reading String from a File:

Opening the File:

Utilize the `open()` function to open the file you want to read from. Specify the file path and the mode as `r` (read).

Following example can be referred to open the file in read mode.

Example:

```
file_path = "example.txt"  
with open(file_path, 'r') as file:  
    # Read operations go here
```

Reading Data:

You can read the file contents using various methods such as `read()`, `readline()`, or by iterating over the file object.

To read the entire file as a string, refer to example given below.

Example:

```
with open(file_path, 'r') as file:  
    file_contents = file.read()
```

To read the file content line by line, following example can be referred.

Example:

```
with open(file_path, 'r') as file:  
    for line in file:  
        # Process each line  
        print(line)
```

Closing a File:

Similar to the process of writing, it is important to close the file after reading to release system resources. The 'with' statement handles this automatically.

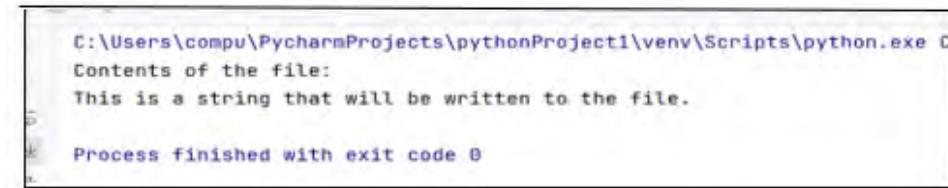
Code Snippet 3 shows how to read String from a File.

Code Snippet 3:

```
# Specify the name of the file to read from  
file_name = "example.txt"  
  
try:  
    with open(file_name, 'r') as file:  
        # Read the contents of the file into a string  
        file_contents = file.read()  
        print("Contents of the file:")  
        print(file_contents)  
except FileNotFoundError:  
    print(f"The file {file_name} does not exist.")  
except IOError:  
    print(f"An error occurred while reading from {file_name}.")
```

In Code Snippet 3, the code attempts to open a file named 'example.txt' for reading. If the file exists, it reads its contents into a string and prints them. If the file does not exist, it raises a `FileNotFoundError`, and if there is an error while reading the file, it raises an `IOError`. In both error cases, it prints an appropriate error message. The `try-except` blocks handle potential exceptions that may occur during the file operation.

Figure 11.2 depicts the output of this code when executed through PyCharm.



The screenshot shows a terminal window from PyCharm. The command entered is 'python <file>'. The output displays the contents of a file named 'contents.txt' which contains the string 'This is a string that will be written to the file.' followed by a message indicating the process finished successfully with an exit code of 0.

```
C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:\Users\compu\PycharmProjects\pythonProject1\contents.txt
Contents of the file:
This is a string that will be written to the file.

Process finished with exit code 0
```

Figure 11.2: Output of Code Snippet 3

11.3 Introduction to Pickle

In Python, Pickle refers to a built-in module called `pickle` that provides a way to serialize and deserialize Python objects. Serialization is the process of converting complex data structures, such as dictionaries or custom objects, into a format that can be easily stored or transmitted. Deserialization is the reverse process, where the serialized data is restored back into its original form. The `pickle` module converts Python objects into a binary format that can be saved to a file or transmitted over a network. It then reconstructs these objects back into Python objects when required.

Following are some key points about the pickle module:

Serialization and Deserialization: Pickle allows you to serialize (convert to binary) and deserialize (convert back to Python objects) Python data types, including dictionaries, lists, tuples, custom objects, and more.

Binary Format: The data produced by pickle is in a binary format. It makes it more compact, efficient for storage and transmission compared to plain text formats such as JSON or XML. This binary format is not human-readable.

Cross-Version Compatibility: Pickle is version-specific, meaning that you should be cautious when using pickle files created in one Python version with a different Python version. It is recommended to utilize the same Python version for serialization and deserialization.

Security Considerations: Pickle files can execute arbitrary code during deserialization, making them a potential security risk when loading data from

untrusted sources. To mitigate this risk, it is essential to avoid unpickling data from untrusted or unauthenticated sources.

Here is a simple example of how to utilize the pickle module to serialize and deserialize a Python object:

Code Snippet 4 shows how to utilize `Pickle` module.

Code Snippet 4:

```
import pickle

# Sample data (a Python dictionary)
data = {"name": "Alice", "age": 30}

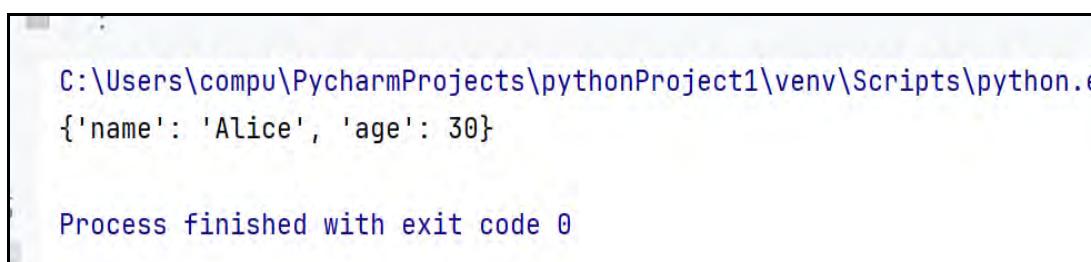
# Serialize the data and save it to a file
with open("data.pickle", "wb") as file:
    pickle.dump(data, file)

# Deserialize the data from the file
with open("data.pickle", "rb") as file:
    loaded_data = pickle.load(file)

# Print the deserialized data
print(loaded_data)
```

In Code Snippet 4, the code demonstrates serialization and deserialization using Python's `pickle` module. It begins by defining sample data as a Python dictionary. Then, it serializes the data and saves it to a file named `data.pickle`. Next, it deserializes the data from the file using `pickle.load()`, storing the result in `loaded_data`. Finally, it prints the deserialized data.

Figure 11.3 depicts the output of this code when executed through PyCharm.



The screenshot shows the PyCharm terminal window. The command entered is `C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe`. The output displayed is `{'name': 'Alice', 'age': 30}`. At the bottom, the message `Process finished with exit code 0` is shown.

Figure 11.3: Output of Code Snippet 4

In this example, `pickle.dump()` has been utilized to serialize the data dictionary into a binary file called `data.pickle` and `pickle.load()` to deserialize it back into the `loaded_data` variable.

11.4 Introduction to Threads in Python

In Python, a thread refers to a lightweight, independent, and separate flow of control within a program. Threads are a way to achieve concurrent execution in a Python program. It allows multiple tasks to run concurrently, which potentially speeds up the execution of CPU-bound or I/O-bound operations. Threads in Python are implemented using the `threading` module, which provides a high-level interface for working with threads.

11.4.1 Key Aspects of Threads in Python

Concurrency: Threads enable concurrent execution, which means that multiple threads can run simultaneously within a single program. This is particularly useful for tasks that can be performed independently and concurrently.

Global Interpreter Lock (GIL): Python has a GIL that allows only one thread to execute Python bytecode at a time. This limitation can affect multi-core processors and may not provide full parallelism for CPU-bound tasks. However, threads can still be beneficial for I/O-bound operations.

Threading Module: Python's `threading` module provides a high-level, object-oriented interface for creating and managing threads. It simplifies the creation and synchronization of threads, making it easier to work with concurrency.

Thread Safety: When multiple threads access shared data, it is essential to ensure thread safety to prevent data corruption or race conditions. Python provides tools such as locks, semaphores, and conditions to help manage thread synchronization.

Parallelism vs. Concurrency: Threads primarily provide concurrency, not necessarily parallelism. Parallelism involves executing multiple threads or processes simultaneously on multiple CPU cores. Concurrency involves

managing multiple tasks concurrently, including tasks that are interleaved on a single CPU core.

Code Snippet 5 shows creating and running a thread in Python using the `threading` module.

Code Snippet 5:

```
import threading
def print_numbers():
    for i in range(1, 6):
        print(f"Number {i}")

def print_letters():
    for letter in "abcde":
        print(f"Letter {letter}")

# Create two threads
thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_letters)

# Start the threads
thread1.start()
thread2.start()

# Wait for both threads to finish
thread1.join()
thread2.join()

print("Both threads have finished.")
```

In Code Snippet 5, the code demonstrates multithreading in Python using the `threading` module. It defines two functions, `print_numbers()` and `print_letters()`, each printing numbers and letters respectively. Two threads are created, each targeting one of these functions. The threads are started using `start()` and then joined back using `join()` to wait for their completion. Finally, a message is printed indicating that both threads have finished.

Figure 11.4 depicts the output of this code when executed through PyCharm.

```
C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:\Users\compu\PycharmProjects\pythonProject1\main.py
Number 1
Number 2
Number 3
Number 4
Number 5
Letter a
Letter b
Letter c
Letter d
Letter e
Both threads have finished.
```

Figure 11.4: Output of Code Snippet 5

In this program, two threads are created, each with a specific function (`print_numbers` and `print_letters`) so that they execute concurrently. The `start()` method initiates the execution of each thread, and `join()` is utilized to wait for both threads to finish before continuing with the main program.

Python threads facilitate introducing concurrency into applications, which enables simultaneous task execution and enhances program efficiency, notably for I/O-bound operations. However, for CPU-bound tasks that require true parallelism, Python's `multiprocessing` module or other parallel programming techniques may be more suitable.

11.4.2 Main Thread in Python

In Python, the main thread serves as the primary execution thread for a Python program. Whenever you initiate the execution of a Python script or program, the code begins executing within the main thread by default. The main thread executes top-level statements in your script and plays a central role in managing the execution of other threads. This is crucial when employing multi-threading or multi-processing techniques in your program.

Following are some fundamental points regarding the main thread in Python:

Single Entry Point: The main thread is the initial entry point for your Python program. When you invoke a Python script, the code within the main thread commences execution from the beginning of the script.

Sequential Execution: By default, Python operates in a single-threaded manner, implying that the main thread executes your code in a sequential fashion. It processes statements one at a time, adhering to the order they appear in your script.

Global Scope: The main thread operates within the global scope of your script. Variables and objects created in the global scope are accessible from the main thread and also from any other threads that you may create.

11.4.3 MainThread Objects

Python's `threading` module offers the capability to obtain a reference to the main thread through the `threading.main_thread()` function. This feature can be valuable if you are required to inspect or manipulate the main thread's behavior from within other threads.

Code Snippet 6 shows a Python program featuring the main thread.

Code Snippet 6:

```
import threading
def main():
    print("This is the main thread.")

if __name__ == "__main__":
    main_thread = threading.main_thread()
    print(f"Main thread name: {main_thread.name}")
    main()
```

In Code Snippet 6, it demonstrates how to obtain and print information about the main thread in a Python program using the `threading` module. It defines a `main()` function which simply prints a message indicating it is the main thread. The `if __name__ == __main__:` block ensures that the `main()` function is called only when the script is executed directly. Within this block, it retrieves the main thread object using `threading.main_thread()` and prints its name. Finally, it calls the `main()` function.

Figure 11.5 depicts the output of this code when executed through PyCharm.

```
C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.
Main thread name: MainThread
This is the main thread.

Process finished with exit code 0
```

Figure 11.5: Output of Code Snippet 6

Python supports multi-threading and multi-processing, which allows to create additional threads or processes for concurrent tasks. Nevertheless, the main thread retains its role as the primary starting point and often serves as the central controller for these auxiliary threads or processes.

11.5 Threads Using Class and Functions in Python

Creating threads in Python can be done using either functions or classes.

11.5.1 Using a Function

To create a thread using a function, you will typically utilize the `threading` module's `Thread` class and provide the function you want to run in the new thread as the target.

Code Snippet 7 shows the creation of a Thread using a function.

Code Snippet 7:

```
import threading

def my_function():
    for i in range(5):
        print(f"Function: {i}")

# Create a thread using a function
my_thread = threading.Thread(target=my_function)

# Start the thread
my_thread.start()

# Wait for the thread to finish (optional)
my_thread.join()

print("Main thread continues.")
```

In Code Snippet 7, the code demonstrates how to create and start a new thread in Python using the `threading` module. It defines a function `my_function()` that prints numbers from 0 to 4. Then, it creates a new thread `my_thread` targeting `my_function()` function. The thread is started using `start()` and then optionally joined back using `join()` to wait for its completion. Finally, a message indicating the continuation of the main thread is printed.

Figure 11.6 depicts the output of this code when executed through PyCharm.

```
C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:\Users  
Function: 0  
Function: 1  
Function: 2  
Function: 3  
Function: 4  
Main thread continues.  
  
Process finished with exit code 0
```

Figure 11.6: Output of Code Snippet 7

11.5.2 Using a Class

To create a thread using a class, you will define a custom class that inherits from `threading.Thread` and override the `run()` method within the class. The `run()` method contains the code you want to execute in the thread.

Code Snippet 8 shows how to create a thread using a class.

Code Snippet 8:

```
import threading  
  
class MyThread(threading.Thread):  
    def run(self):  
        for i in range(5):  
            print(f"Class: {i}")  
  
    # Create an instance of the custom thread class  
my_thread = MyThread()  
  
    # Start the thread  
my_thread.start()  
  
    # Wait for the thread to finish (optional)  
my_thread.join()  
  
print("Main thread continues.")
```

In Code Snippet 8, the code demonstrates how to create a custom thread class in Python by sub-classing `threading.Thread`. The subclass `MyThread` overrides the `run()` method, where the actual thread logic is defined. In this case, it prints numbers from 0 to 4. An instance of `MyThread` class named `my_thread` is created,

and then the thread is started using `start()`. Optionally, `join()` is used to wait for the thread to finish execution. Finally, a message indicating the continuation of the main thread is printed.

Figure 11.7 depicts the output of this code when executed through PyCharm.

```
C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:\  
Class: 0  
Class: 1  
Class: 2  
Class: 3  
Class: 4  
Main thread continues.
```

Figure 11.7: Output of Code Snippet 8

Code Snippet 9 demonstrates the creation of a thread using class and function in Python.

Code Snippet 9:

```
import threading  
  
# Define a function to run in a thread  
def function_thread():  
    for i in range(5):  
        print(f"Function Thread: {i}")  
  
# Define a custom thread class  
class MyThreadClass(threading.Thread):  
    def run(self):  
        for i in range(5):  
            print(f"Class Thread: {i}")  
  
# Create and start a thread using a function  
function_thread = threading.Thread(target=function_thread)  
function_thread.start()  
  
# Create and start a thread using a custom class  
class_thread = MyThreadClass()  
class_thread.start()
```

```
# Wait for both threads to finish (optional)
function_thread.join()
class_thread.join()

print("Main thread continues.")
```

In Code Snippet 9, the provided code demonstrates Python's multithreading capabilities through both function-based and class-based approaches. It defines a function and a class, each representing a task for threads to execute, printing numbers from 0 to 4. Two threads are then created and started, one for each task. The main thread optionally waits for both threads to finish before proceeding, and finally, it prints a message indicating its continuation.

Figure 11.8 depicts the output of this code when executed through PyCharm.

```
C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:\Users\compu\PycharmP
Function Thread: 0
Function Thread: 1
Function Thread: 2
Function Thread: 3
Function Thread: 4
Class Thread: 0
Class Thread: 1
Class Thread: 2
Class Thread: 3
Class Thread: 4
Main thread continues.
```

Figure 11.8: Output of Code Snippet 9

11.6 Multithreading in Python

Multithreading in Python refers to the concurrent execution of multiple threads within a single Python process. A thread is the smallest unit of a program's execution and represents an independent flow of control.

Multithreading allows a Python program to perform multiple tasks or operations concurrently, potentially improving the program's efficiency and responsiveness.

11.6.1 Import the threading Module

Start by importing the `threading` module, which allows you to create and manage threads.

```
import threading
```

11.6.2 Define a Thread Function

Create a function that represents the task you want to execute in parallel. This function will be run by multiple threads. Refer to exemplary function given below.

Example:

```
def worker_thread(thread_id):
    # Your code for the task goes here
```

11.6.3 Create Thread Objects

Create instances of the `Thread` class from the `threading` module, passing the thread function and any required arguments as arguments to the `Thread` constructor. Refer to example given below.

Example:

```
thread1 = threading.Thread(target=worker_thread, args=(1,))
thread2 = threading.Thread(target=worker_thread, args=(2,))
# Create more threads as required
```

11.6.4 Start Threads

Utilize the `start()` method to initiate the execution of each thread as shown in example given below.

Example:

```
thread1.start()
thread2.start()
# Start more threads as required
```

Wait for Threads to Finish

You can utilize the `join()` method to wait for a thread to complete its execution before proceeding further. This ensures that the main thread does not exit prematurely. Refer exemplary format given below.

Example:

```
thread1.join()
thread2.join()
# Use join() for other threads as required
```

Code Snippet 10 shows an example of Multithreading.

Code Snippet 10:

```
import threading
```

```

def worker_thread(thread_id):
    print(f"Thread {thread_id} started.")
    # Your task code here
    print(f"Thread {thread_id} finished.")

# Create thread objects
thread1 = threading.Thread(target=worker_thread, args=(1,))
thread2 = threading.Thread(target=worker_thread, args=(2,))

# Start threads
thread1.start()
thread2.start()

# Wait for threads to finish
thread1.join()
thread2.join()

print("Main thread continues.")

```

In Code Snippet 10, the code demonstrates the use of threading in Python. It defines a function `worker_thread()` representing the task to be executed by each thread. Two threads are created with unique identifiers, and they are started to execute the `worker_thread()` function. The main thread waits for both worker threads to finish their tasks using `join()`, and finally, it prints a message indicating its continuation.

Figure 11.9 depicts the output of this code when executed through PyCharm.

```

C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:/Users/compu/Pychar
Thread 1 started.
Thread 1 finished.
Thread 2 started.
Thread 2 finished.
Main thread continues.

Process finished with exit code 0

```

Figure 11.9: Output of Code Snippet 10

11.7 Communication Through Threads in Python

In Python, threads enable concurrent execution, enabling multiple parts of a program to run simultaneously. Their usage spans various purposes. It includes

facilitating communication between threads which is crucial for data sharing or coordinating activities within a multi-threaded application.

One common way to achieve communication between threads in Python is by using the `threading` module, which provides a high-level interface for working with threads. Here is a simple explanation of how to utilize threads for communication in Python.

Following are some common ways to achieve thread communication in Python:

11.7.1 Shared Variables

Threads can communicate by sharing variables in memory. However, this approach requires proper synchronization to prevent race conditions. You can utilize locks (example `threading.Lock()`) to protect shared resources and ensure that only one thread can access them at a time.

Code Snippet 11 shows a sample of Shared Variables in Threads.

Code Snippet 11:

```
import threading

shared_variable = 0
lock = threading.Lock()

def thread1():
    global shared_variable
    with lock:
        shared_variable += 1

def thread2():
    global shared_variable
    with lock:
        shared_variable -= 1
```

In Code Snippet 11, the code demonstrates the use of threading with a shared variable in Python. Two threads are defined, each modifying the shared variable `shared_variable` within a critical section protected by a lock. `thread1()` increments the shared variable while `thread2()` decrements it. The `threading.Lock()` object ensures that only one thread can access the shared variable at a time, preventing potential race conditions.

11.7.2 Queue (Thread-Safe)

The `queue` module provides thread-safe data structures such as `Queue` and `PriorityQueue`. These queues can be utilized for safe communication between threads, allowing one thread to put data into the queue and another thread to retrieve it.

Code Snippet 12 shows a sample of queue module in Threads.

Code Snippet 12:

```
import threading
import queue

q = queue.Queue()

def producer():
    data = "Hello, World!"
    q.put(data)

def consumer():
    data = q.get()
    print(data)
```

In Code Snippet 12, the code illustrates a producer-consumer pattern using `threading` and a `queue` in Python. The `producer()` function adds data into a shared queue '`q`' using `put()`, while the `consumer()` function retrieves data from the queue using `get()` and prints it. By using a queue, threads can safely communicate and share data without the risk of data corruption or race conditions.

Condition Variables:

Condition variables (example, `threading.Condition`) allow threads to synchronize their actions based on certain conditions. Threads can wait for a condition to be met and signal when the condition changes.

Code Snippet 13 shows a sample of conditional variables in Threads.

Code Snippet 13:

```
import threading
condition = threading.Condition()
def thread1():
```

```

        with condition:
            condition.wait()
            print("Thread 1 woke up!")

def thread2():
    with condition:
        condition.notify()

```

In Code Snippet 13, threading condition objects are utilized for synchronization between threads. The function `thread1()` enters a critical section protected by the condition, where it waits for a notification using `condition.wait()`. Once it receives the notification, it prints 'Thread 1 woke up!' On the other hand, the `thread2()` function, upon execution, notifies other threads waiting on the condition using `condition.notify()`. Threading condition objects enable threads to coordinate their execution based on certain conditions, ensuring efficient synchronization and communication between them.

Event Objects:

Event objects (example, `threading.Event`) can be utilized to signal between threads. One thread sets the event, and other threads can wait for the event to be set before proceeding.

Code Snippet 14 shows a sample of event variables in Threads.

Code Snippet 14:

```

import threading

event = threading.Event()

def thread1():
    event.wait()
    print("Thread 1 received the event!")

def thread2():
    event.set()

```

In Code Snippet 14, the code showcases the usage of threading events in Python. An event object is created using `threading.Event()`. In this scenario, `thread1()` waits for the event to be set using `event.wait()`. Meanwhile, `thread2()` sets the event using `event.set()`. Once the event is set by `thread2()`, `thread1()` proceeds to print 'Thread 1 received the event!'

Threading events are synchronization primitives that allow threads to coordinate their execution based on the state of the event.

Code Snippet 15 shows a program on multithreading in Python.

Code Snippet 15:

```
import threading
import time

# Define a function to run in each thread
def worker_thread(thread_id):
    print(f"Thread {thread_id} started.")
    for i in range(5):
        print(f"Thread {thread_id}: Count {i}")
        time.sleep(1) # Simulate some work
    print(f"Thread {thread_id} finished.")

# Create and start multiple threads
threads = []
for i in range(3):
    thread = threading.Thread(target=worker_thread, args=(i,))
    threads.append(thread)
    thread.start()

# Wait for all threads to finish
for thread in threads:
    thread.join()

print("All threads have finished.")
```

In Code Snippet 15, the code showcases the implementation of multithreading to execute multiple instances of a function concurrently. The function `worker_thread()` is defined to simulate work by printing a count within a loop, representing a thread's activity. Three threads are created and started within a loop, each executing the `worker_thread()` function with a unique identifier. The main thread waits for all created threads to finish their execution using `join()`. Upon completion of all threads, a message is printed indicating that all threads have finished their tasks.

Figure 11.10 depicts the output of this code when executed through PyCharm.

```
C:\Users\compu\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:\Users\compu\PycharmProje
Thread 0 started.
Thread 0: Count 0
Thread 1 started.
Thread 1: Count 0
Thread 2 started.
Thread 2: Count 0
Thread 2: Count 1Thread 1: Count 1
Thread 0: Count 1

Thread 0: Count 2Thread 2: Count 2
Thread 1: Count 2

Thread 1: Count 3Thread 0: Count 3
Thread 2: Count 3

Thread 1: Count 4Thread 2: Count 4
Thread 0: Count 4

Thread 1 finished.Thread 0 finished.
Thread 2 finished.

All threads have finished.
```

Figure 11.10: Output of Code Snippet 15

11.8 Summary

- A file is data stored on a medium, accessible for reading, writing, and manipulation within a computer's file system.
- For Python file tasks with strings: open in read or write mode using file methods, and close for data integrity.
- Pickle in Python serializes objects to binary for storage/transmission and restores them to their original form.
- The main thread in Python is the starting execution thread responsible for running the program's primary code.
- Class-based threading involves inheriting from the `Thread` class and overriding the `run()` method. Function-based threading entails utilizing the `threading` module and setting the target for execution upon start.
- Multithreading in Python enables concurrent execution within a single process.
- Threads exchange data and synchronize using shared variables, thread-safe structures, and condition variables for coordinated execution.

11.9 Test Your Knowledge

1. Which of the following methods in Python is utilized to open a file for both reading and writing, creating the file if it does not exist?
 - A) `open("file.txt", "r")`
 - B) `open("file.txt", "w")`
 - C) `open("file.txt", "a")`
 - D) `open("file.txt", "r+")`
2. Which of the following statements is true regarding the Python pickle module?
 - A) Pickle is utilized for compressing files in Python
 - B) Pickle is utilized for converting Python objects into a byte stream
 - C) Pickle is utilized for sorting data in Python lists
 - D) Pickle is utilized for creating database connections in Python
3. In Python, the main thread is:
 - A) The thread responsible for garbage collection
 - B) The thread that runs the `__main__` module
 - C) The thread utilized for handling user interface elements
 - D) The thread created by the threading module
4. In Python, you can create a thread using both a class (by sub-classing `threading.Thread`) and a function (by passing the function as a target to `threading.Thread`).
 - A) True
 - B) False
5. In Python, what is the primary advantage of using multithreading?
 - A) Multithreading allows for parallel execution of multiple Python programs
 - B) Multithreading is utilized to run Python code on multiple processors simultaneously
 - C) Multithreading can improve the responsiveness of a program by allowing multiple tasks to run concurrently
 - D) Multithreading always results in faster program execution compared to multiprocessing

6. Threads in Python can communicate and share data with each other without any requirement for synchronization mechanisms such as locks or semaphores.

- A) True
- B) False

11.9.1 Answers to Test Your Knowledge

1. `open("file.txt", "r+")`
2. Pickle is utilized for converting Python objects into a byte stream.
3. The thread that runs the `__main__` module.
4. True
5. Multithreading can improve the responsiveness of a program by allowing multiple tasks to run concurrently.
6. False

Try It Yourself

1. Create a Python program that reads a text file and counts the number of words in it.
2. Write a program that reads a CSV file containing student names and their scores, calculates the average score, and displays the names of students who scored above the average.
3. Create a Python program that utilizes two threads to download multiple images concurrently from the Internet. Save each downloaded image to a local directory.
4. Build a program that simulates a simple ticket booking system with multiple booking agents (threads). Each booking agent should try to book tickets concurrently, and you should ensure that tickets are not overbooked.



SESSION 12

NETWORKING AND DATABASE HANDLING

Learning Objectives

In this session, students will learn to:

- ◆ Identify steps involved in accessing HTML, Images, and Server
- ◆ Explain how to install and setup MySQL Database
- ◆ Explain how to create a Database in Python
- ◆ Define CREATE, READ, UPDATE and DELETE commands
- ◆ Describe how to download and use tools in Python

12.1 Accessing HTML, Images, and Server

Various libraries and modules are available in Python to access HTML, images, and interact with a server. Following are some common libraries and approaches for each task:

12.1.1 Accessing HTML

Accessing HTML content from a Website is possible through libraries such as requests for making HTTP requests and beautiful soup for parsing HTML. Ensure to install the libraries if you have not already:

```
pip install requests beautifulsoup4
```

Code Snippet 1 shows an example of how to retrieve HTML content from a Website.

Code Snippet 1:

```
import requests
from bs4 import BeautifulSoup

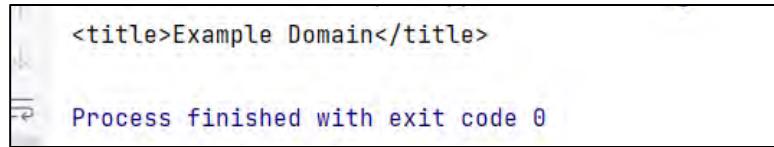
# Make an HTTP GET request to the Website
response = requests.get("https://example.com")

# Check if the request was successful
if response.status_code == 200:
    # Parse the HTML content using BeautifulSoup
    soup = BeautifulSoup(response.text, "html.parser")

    # You can now work with the parsed HTML, example, #extract
    # data or manipulate it
    print(soup.title) # Print the title of the Webpage
else:
    print("Failed to retrieve HTML:", response.status_code)
```

Code Snippet 1 shows Python script that fetches the HTML content of `https://example.com` using a HTTP GET request. If the request is successful (status code 200), it parses the HTML using BeautifulSoup and prints the Webpage title. Otherwise, it prints a failure message with the status code.

Figure 12.1 depicts the output of this code when executed through PyCharm.



```
<title>Example Domain</title>
Process finished with exit code 0
```

Figure 12.1: Output of Code Snippet 1

12.1.2 Accessing Images

Downloading and handling images involves using the `requests` library to retrieve image content, which can then be saved to your local machine.

Code Snippet 2 shows an example to download image.

Code Snippet 2:

```
import requests

# URL of the image to download
image_url = "https://example.com/image.jpg"

# Send an HTTP GET request to the image URL
response = requests.get(image_url)

# Check if the request was successful
if response.status_code == 200:
    # Save the image to a local file
    with open("image.jpg", "wb") as file:
        file.write(response.content)
else:
    print("Failed to download image:", response.status_code)
```

Code Snippet 2 shows Python script that downloads an image from the URL `https://example.com/image.jpg` using an HTTP GET request with the `requests` library. If the request is successful (status code 200), it saves the image to a local file named `image.jpg`. If the request fails, it prints a failure message along with the status code.

12.1.3 Interacting with a Server

Interacting with a server involves utilizing the `requests` library to transmit HTTP requests to the server and manage the corresponding responses.

Code Snippet 3 shows an example of making a POST request.

Code Snippet 3:

```
import requests

# URL of the server endpoint
server_url = "
https://www.themealdb.com/api/json/v1/1/filter.php?i=bread"

# Data to send in the request (can be JSON, form data, #and so
on.)
data = {"key": "value"}

# Send an HTTP POST request to the server
response = requests.post(server_url, json=data)

# Check if the request was successful
if response.status_code == 200:
    # Process the server response
    server_data = response.json()
    print("Server response:", server_data)
else:
    print("Failed to send POST request:", response.status_code)
```

Code Snippet 3 shows Python script that sends an HTTP POST request to the server endpoint

`https://www.themealdb.com/api/json/v1/1/filter.php?i=bread` with data `{"key": "value"}`. If successful (status code 200), it prints the server response in JSON format; otherwise, it prints a failure message with the status code.

Figure 12.2 depicts the output of this code when executed through PyCharm.



```
Server response: {"meals": [{"strMeal": "BBQ Pork Sloppy Joes", "strMealThumb": "https://www.themealdb.com/images/media/means/af0d5sh1583188467.jpg", "idMeal": '52995'}, {"strMeal": 'Bread and Butter Pudding', "strMealThumb": "https://www.themealdb.com/images/media/means/xqawpy14398497.jpg", "idMeal": '52792'}, {"strMeal": 'Bread omelette', "strMealThumb": "https://www.themealdb.com/images/media/means/hqae11695738655.jpg", "idMeal": '53076'}, {"strMeal": 'Chilli prawn linguine', "strMealThumb": "https://www.themealdb.com/images/media/means/usypp1511189717.jpg", "idMeal": '52839'}, {"strMeal": 'Chivito uruguayo', "strMealThumb": "https://www.themealdb.com/images/media/means/n7qnbk1630444129.jpg", "idMeal": '53063'}, {"strMeal": 'English Breakfast', "strMealThumb": "https://www.themealdb.com/images/media/means/utrryn1511721587.jpg", "idMeal": '52895'}, {"strMeal": 'French Onion Soup', "strMealThumb": "https://www.themealdb.com/images/media/means/xvrrrx1511783685.jpg", "idMeal": '52903'}, {"strMeal": 'Full English Breakfast', "strMealThumb": "https://www.themealdb.com/images/media/means/sortwu1511721265.jpg", "idMeal": '52896'}, {"strMeal": 'Lasagna Sandwiches', "strMealThumb": "https://www.themealdb.com/images/media/means/xr0n4r1576788363.jpg", "idMeal": '52987'}, {"strMeal": 'Soy-Glazed Meatloaves with Wasabi Mashed Potatoes & Roasted Carrots', "strMealThumb": "https://www.themealdb.com/images/media/means/o2ab6p1581005243.jpg", "idMeal": '52992'}, {"strMeal": 'Split Pea Soup', "strMealThumb": "https://www.themealdb.com/images/media/means/xxtavx1511814683.jpg", "idMeal": '52925'}, {"strMeal": 'Summer Pudding', "strMealThumb": "https://www.themealdb.com/images/media/means/rsqwsu1511648214.jpg", "idMeal": '52889'}]}
```

Figure 12.2: Output of Code Snippet 3

12.2 Installing and Setting up MySQL Database

To install and set up a MySQL database for Python, following steps are required to be followed.

12.2.1 Install MySQL Server

The first step is to install MySQL Server on your system. The installation process varies depending on your operating system. Following are some common methods:

Linux (Ubuntu/Debian)

```
sudo apt-get update  
sudo apt-get install mysql-server
```

Linux (CentOS/RHEL)

```
sudo yum install mysql-server
```

macOS (using Homebrew)

```
brew install mysql
```

Windows

Download the MySQL Installer for Windows from the official MySQL Website and follow the installation wizard.

12.2.2 Start MySQL Server

After installation, start the MySQL server. The method to start MySQL varies by operating system. On Linux, you can use:

```
sudo systemctl start mysql
```

On macOS or Windows, you can start it using the services or system preferences.

12.2.3 Install the MySQL Connector for Python

To interact with the MySQL database from Python, you must install the `mysql-connector-python` library. You can install it using `pip`:

```
pip install mysql-connector-python
```

12.2.4 Connect to MySQL from Python

Now that you have MySQL installed and the Python library in place, you can connect to the MySQL server from your Python script. Here is an example:

Code Snippet 4 shows an example of connecting MySQL from Python.

Code Snippet 4:

```
import mysql.connector

# Replace these values with your MySQL server
#configuration
config = {
    "host": "localhost",
    "user": "your_username",
    "password": "your_password",
    "database": "your_database_name"
}

# Establish a connection to the MySQL server
try:
    connection = mysql.connector.connect(**config)
    if connection.is_connected():
        print("Connected to MySQL database")

# Perform database operations here
except mysql.connector.Error as e:
    print(f"Error: {e}")

finally:
    # Close the connection when done
    if connection.is_connected():
        connection.close()
        print("Connection closed")
```

Code Snippet 4 shows Python script that establishes a connection to a MySQL database using the provided server configuration. It then performs database operations within a `try-except` block. If an error occurs during the database operations, it prints the error message. Finally, it closes the database connection in the `finally` block.

Figure 12.3 depicts the output of this code when executed through PyCharm.

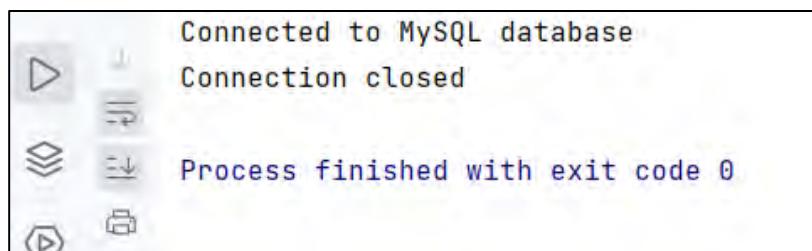


Figure 12.3: Output of Code Snippet 4

12.2.5 Execute SQL Queries

With the connection established, you can execute SQL queries using a `cursor` object.

Code Snippet 5 shows execution of SQL queries using a `cursor` object.

Code Snippet 5:

```
cursor = connection.cursor()

# Example: Create a table
create_table_query = """
CREATE TABLE IF NOT EXISTS employees (
    id INT AUTO_INCREMENT PRIMARY KEY,
    first_name VARCHAR(255),
    last_name VARCHAR(255),
    e-mail VARCHAR(255)
)
"""
cursor.execute(create_table_query)

# Commit changes to the database
connection.commit()

# Close the cursor
cursor.close()
```

Code Snippet 5 demonstrates how to create a table named `employees` in a MySQL database using the `cursor` object. It executes an SQL query to create the table with columns for employee ID, first name, last name, and email address. After executing the query, it commits the changes to the database and closes the `cursor`.

12.2.6 Perform Other Database Operations

Various database operations such as inserting, updating, and querying data are achievable using SQL queries executed via the `cursor`.

Remember to replace placeholders with your actual database credentials and modify the database operations as required for your specific application.

12.3 Creating a Database from Python

To create a database from Python, use the MySQL connector and execute SQL commands to create the database.

A step-by-step guide to create a database from Python is as follows:

12.3.1 Install MySQL Connector

First, make sure that the `mysql-connector-python` library installed. You can install it using `pip`:

```
pip install mysql-connector-python
```

12.3.2 Connect to the MySQL Server

Next, establish a connection to the MySQL server by providing your MySQL credentials (`host`, `username`, and `password`). Replace these placeholders with your MySQL server details as shown in given example.

Example:

```
import mysql.connector

# Replace with your MySQL server details
config = {
    "host": "localhost",
    "user": "your_username",
    "password": "your_password",
}

try:
    connection = mysql.connector.connect(**config)
    if connection.is_connected():
        print("Connected to MySQL server")
except mysql.connector.Error as e:
    print(f"Error: {e}")
```

12.3.3 Create a Database

To create a new database, you can execute a SQL `CREATE DATABASE` statement using a `cursor` object.

Replace `your_database_name` with the desired name of your new database as shown in given example.

Example:

```
try:
```

```

cursor = connection.cursor()

# Replace "your_database_name" with your desired database name
create_database_query = "CREATE DATABASE your_database_name"

cursor.execute(create_database_query)
print("Database created successfully")

except mysql.connector.Error as e:
    print(f"Error: {e}")
finally:
    if connection.is_connected():
        cursor.close()

```

12.3.4 Close the Connection

Do not forget to close the connection once you are done. Refer to example given below to close the connection.

Example:

```

if connection.is_connected():
    connection.close()
    print("Connection closed")

```

This code will create a new database with the specified name. Ensure to replace `your_database_name` with the desired name for your database.

Code Snippet 6 shows a complete Python program to create a database.

Code Snippet 6:

```

import mysql.connector

# Replace with your MySQL server details
config = {
    "host": "localhost",
    "user": "your_username",
    "password": "your_password",
}

# Replace with your desired database name
database_name = "your_database_name"

try:
    # Connect to the MySQL server
    connection = mysql.connector.connect(**config)
    if connection.is_connected():

```

```

print("Connected to MySQL server")

# Create a cursor
cursor = connection.cursor()

# Create the database
create_database_query = f"CREATE DATABASE IF NOT EXISTS {database_name}"
cursor.execute(create_database_query)
print(f"Database '{database_name}' created successfully")

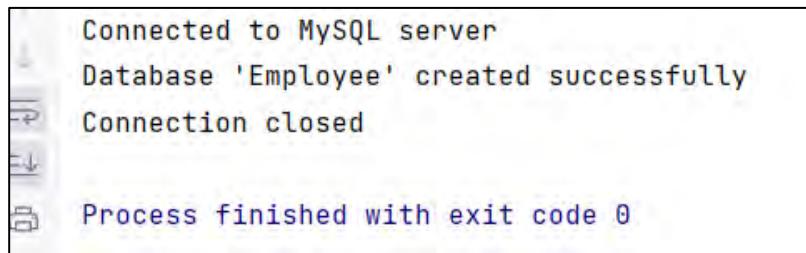
except mysql.connector.Error as e:
    print(f"Error: {e}")

finally:
    if connection.is_connected():
        cursor.close()
        connection.close()
        print("Connection closed")

```

Code Snippet 6 shows Python script that connects to a MySQL server using the provided server details and attempts to create a database with the specified name if it does not already exist. It prints a confirmation message if the database is created successfully. If an error occurs during the process, it prints the error message. Finally block closes the connection and cursor.

Figure 12.4 depicts the output of this code when executed through PyCharm.



```

Connected to MySQL server
Database 'Employee' created successfully
Connection closed
Process finished with exit code 0

```

Figure 12.4: Output of Code Snippet 6

12.4 CREATE, READ and DELETE commands in Python for MySQL

To perform CREATE, READ, UPDATE and DELETE (CRUD) operations in Python with a MySQL database, use the MySQL connector.

After setting up a MySQL database and establishing a connection, proceed with following steps:

12.4.1 CREATE (Insert Data)

This command is used to insert data into MySQL database.

Code Snippet 7 demonstrates the steps to use this command in MySQL database.

Code Snippet 7:

```
import mysql.connector

# Replace with your MySQL server details
config = {
    "host": "localhost",
    "user": "your_username",
    "password": "your_password",
    "database": "your_database_name",
}

try:
    connection = mysql.connector.connect(**config)
    if connection.is_connected():
        print("Connected to MySQL server")

        # Create a cursor
        cursor = connection.cursor()

        # Define an INSERT query
        insert_query = "INSERT INTO your_table_name (column1,
column2) VALUES (%s, %s)"

        # Data to be inserted
        data = ("value1", "value2")

        # Execute the INSERT query
        cursor.execute(insert_query, data)

        # Commit the changes to the database
        connection.commit()
        print("Data inserted successfully")

except mysql.connector.Error as e:
    print(f"Error: {e}")

finally:
    if connection.is_connected():
        cursor.close()
```

```
connection.close()
print("Connection closed")
```

Replace `your_table_name` and the column names with your actual table and column names, and `value1` and `value2` with the data you want to insert.

Code Snippet 7 shows Python script that connects to a MySQL database and inserts data into a specified table. It defines an `INSERT` query with placeholders for values to be inserted and executes it with the provided data. Finally, it commits the changes to the database and closes the connection.

Figure 12.5 depicts the output of this code when executed through PyCharm.

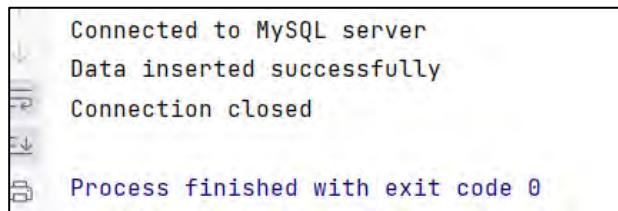


Figure 12.5: Output of Code Snippet 7

12.4.2 READ (Retrieve Data)

This command is used to retrieve data from a MySQL database.

Code Snippet 8 shows how to use this command in MySQL database.

Code Snippet 8:

```
import mysql.connector

# Replace with your MySQL server details
config = {
    "host": "localhost",
    "user": "your_username",
    "password": "your_password",
    "database": "your_database_name",
}

try:
    connection = mysql.connector.connect(**config)
    if connection.is_connected():
        print("Connected to MySQL server")

        # Create a cursor
        cursor = connection.cursor()
```

```

# Define a SELECT query
select_query = "SELECT * FROM your_table_name"

# Execute the SELECT query
cursor.execute(select_query)

# Fetch all rows of data
records = cursor.fetchall()

# Process and print the retrieved data
for record in records:
    print(record)

except mysql.connector.Error as e:
    print(f"Error: {e}")

finally:
    if connection.is_connected():
        cursor.close()
        connection.close()
        print("Connection closed")

```

Replace `your_table_name` with the name of the table from which you want to retrieve data.

Code snippet 8 shows Python script that connects to a MySQL database and retrieves all records from a specified table. It defines a `SELECT` query to fetch data from the table, executes the query, and fetches all rows of data using the `fetchall()` method. It then processes and prints the retrieved data. Finally, it closes the connection to the database.

Figure 12.6 depicts the output of this code when executed through PyCharm.

```

Connected to MySQL server
('Andrew', 'Jhonson')
('Dom', 'Jhonson')
('Emilye', 'Jhonson')
Connection closed

Process finished with exit code 0

```

Figure 12.6: Output of Code Snippet 8

12.4.3 UPDATE

To perform an UPDATE command in Python, use the MySQL connector (or the appropriate database library for your chosen database system). Code Snippet 9 demonstrates the process to update data in a MySQL database using the `mysql-connector-python` library. Make sure the library is installed. If it is not installed, install it using the `pip` command given below.

```
pip install mysql-connector-python
```

Code Snippet 9 shows an example that update data in a MySQL database.

Code Snippet 9:

```
import mysql.connector

# Replace with your MySQL server details
config = {
    "host": "localhost",
    "user": "your_username",
    "password": "your_password",
    "database": "your_database_name",
}

try:
    # Connect to the MySQL server
    connection = mysql.connector.connect(**config)
    if connection.is_connected():
        print("Connected to MySQL server")

    # Create a cursor
    cursor = connection.cursor()

    # Define an UPDATE query
    update_query = "UPDATE your_table_name SET column1 = %s
WHERE column2 = %s"

    # Data for the update (replace with your data)
    update_data = ("new_value", "value_to_identify_row")

    # Execute the UPDATE query
    cursor.execute(update_query, update_data)

    # Commit the changes to the database
    connection.commit()
    print("Data updated successfully")
```

```

except mysql.connector.Error as e:
    print(f"Error: {e}")

finally:
    if connection.is_connected():
        cursor.close()
        connection.close()
        print("Connection closed")

```

Code Snippet 9 shows Python script that connects to a MySQL database and updates records in a specified table. It defines an `UPDATE` query to set a column's value based on a condition. It then executes the query with the provided data, and commits the changes to the database. Finally, it closes the connection.

Replace the placeholders in the config dictionary with your MySQL server details. Replace `your_table_name` with the name of your table, `column1` with the column you want to update, `column2` with the column by which you want to identify the row to update, `new_value` with the new value, and `value_to_identify_row` with the value that should identify the row to update.

Figure 12.7 depicts the output of this code when executed through PyCharm.

```

Connected to MySQL server
Data updated successfully
Connection closed
Process finished with exit code 0

```

Figure 12.7: Output of Code Snippet 9

12.4.4 `DELETE` (Remove Data)

This command is used to delete or remove data from MySQL database.

Code Snippet 10 shows how to use `DELETE` command in MySQL database.

Code Snippet 10:

```

import mysql.connector

# Replace with your MySQL server details
config = {
    "host": "localhost",

```

```

"user": "your_username",
"password": "your_password",
"database": "your_database_name",
}

try:
    connection = mysql.connector.connect(**config)
    if connection.is_connected():
        print("Connected to MySQL server")

        # Create a cursor
        cursor = connection.cursor()

        # Define a DELETE query
        delete_query = "DELETE FROM your_table_name WHERE
column_name = %s"

        # Data to identify the row to delete
        data = ("value_to_delete",)

        # Execute the DELETE query
        cursor.execute(delete_query, data)

        # Commit the changes to the database
        connection.commit()
        print("Data deleted successfully")

except mysql.connector.Error as e:
    print(f"Error: {e}")

finally:
    if connection.is_connected():
        cursor.close()
        connection.close()
        print("Connection closed")

```

Code Snippet 10 shows Python script that connects to a MySQL database and deletes records from a specified table. It defines a `DELETE` query to remove rows based on a condition. It executes the query with the provided data, and commits the changes to the database. Finally, it closes the `connection`.

Replace `your_table_name` with the name of the table from which you want to delete data, `column_name` with the column by which you want to identify the row to delete, and `value_to_delete` with the value that should trigger the deletion.

Figure 12.8 depicts the output of this code when executed through PyCharm.

```
Connected to MySQL server
Data deleted successfully
Connection closed

Process finished with exit code 0
```

Figure 12.8: Output of Code Snippet 10

Database handling in Python often involves using a Database Management System (DBMS) such as MySQL, PostgreSQL, SQLite, or others.

A general outline of a Python program that demonstrates database handling using the SQLite database is shown as an example in Code Snippet 11. You can adapt this program for other databases by changing the database connection details and SQL queries accordingly.

Create a simple SQLite database, create a table, and perform basic CRUD (Create, Read, Update, Delete) operations.

Code Snippet 11 shows the process of database handling using SQLite database.

Code Snippet 11:

```
import sqlite3

# Create or connect to an SQLite database file
connection = sqlite3.connect("mydatabase.db")

# Create a cursor object to interact with the database
cursor = connection.cursor()

# CREATE TABLE - Create a table if it does not
# exist
create_table_query = """
CREATE TABLE IF NOT EXISTS students (
    id INTEGER PRIMARY KEY,
    name TEXT,
    age INTEGER
)
"""

cursor.execute(create_table_query)
print("Table 'students' created or already exists.")

# INSERT (CREATE) - Insert data into the table
```

```
insert_query = "INSERT INTO students (name, age) VALUES (?, ?)"
data_to_insert = [("Alice", 25), ("Bob", 22), ("Charlie", 28)]
cursor.executemany(insert_query, data_to_insert)
connection.commit()
print("Data inserted successfully.")

# SELECT (READ) - Retrieve data from the table
select_query = "SELECT * FROM students"
cursor.execute(select_query)
students = cursor.fetchall()
print("Retrieved data:")
for student in students:
    print(student)

# UPDATE - Update data in the table
update_query = "UPDATE students SET age = ? WHERE name = ?"
update_data = (30, "Alice")
cursor.execute(update_query, update_data)
connection.commit()
print("Data updated successfully.")

# SELECT (READ) - Retrieve updated data
cursor.execute(select_query)
updated_students = cursor.fetchall()
print("Updated data:")
for student in updated_students:
    print(student)

# DELETE - Delete data from the table
delete_query = "DELETE FROM students WHERE name = ?"
delete_data = ("Bob",)
cursor.execute(delete_query, delete_data)
connection.commit()
print("Data deleted successfully.")

# SELECT (READ) - Retrieve remaining data
cursor.execute(select_query)
remaining_students = cursor.fetchall()
print("Remaining data:")
for student in remaining_students:
    print(student)

# Close the cursor and connection
cursor.close()
connection.close()
```

Code Snippet 11 demonstrates various database operations using SQLite. It creates or connects to a SQLite database file, creates a table if it does not exist, inserts data into the table, retrieves data, updates records, deletes records, and finally closes the cursor and connection.

Figure 12.9 depicts the output of this code when executed through PyCharm.

```
Data inserted successfully.  
Retrieved data:  
(1, 'Alice', 25)  
(2, 'Bob', 22)  
(3, 'Charlie', 28)  
Data updated successfully.  
Updated data:  
(1, 'Alice', 30)  
(2, 'Bob', 22)  
(3, 'Charlie', 28)  
Data deleted successfully.  
Remaining data:  
(1, 'Alice', 30)  
(3, 'Charlie', 28)
```

Figure 12.9: Output of Code Snippet 11

12.5 Downloading and Using Tools

Working with databases in Python commonly involves employing database-specific libraries or tools that offer higher-level abstractions, streamlining database interactions. The selection of tools relies on the utilized database system. Following are some common tools and libraries for different databases:

12.5.1 MySQL

MySQL Connector/Python: This is the official Python driver for MySQL. This can be utilized for connecting to a MySQL database, executing queries, and handling data.

Connect the installed MySQL

```
pip install mysql-connector-python
```

Import MySQL

```
import mysql.connector
```

Establish a connection

```
# Replace these with your actual database credentials
host = "your_host"
user = "your_username"
password = "your_password"
database = "your_database"

# Create a connection
connection = mysql.connector.connect(
    host=host,
    user=user,
    password=password,
    database=database
)
```

Code Snippet 12 shows a simple example to insert data into a table.

Code Snippet 12:

```
# Inserting data
insert_query = "INSERT INTO users (username, e-mail) VALUES (%s, %s)"
user_data = ("JohnDoe", "johndoe@example.com")

cursor = connection.cursor()
cursor.execute(insert_query, user_data)

connection.commit()
cursor.close()
connection.close()
```

Code Snippet 12 shows Python script that establishes a connection to a MySQL database using provided credentials. It then inserts data into the `users` table using parameterized queries to avoid SQL injection vulnerabilities. Finally, it commits the transaction and closes the cursor and connection.

`SQLAlchemy` is an Object-Relational Mapping (ORM) library that supports multiple database systems, including MySQL. It provides an abstraction layer over the database, allowing you to work with Python objects.

```
pip install sqlalchemy
```

12.5.2 PostgreSQL

This is the most popular PostgreSQL adapter for Python. This allows connection to a PostgreSQL database for query execution.

```
pip install psycopg2
```

12.5.3 MongoDB

PyMongo is the official Python driver for MongoDB. It allows you to interact with MongoDB databases.

```
pip install pymongo
```

After installing the suitable library for the database system, it enables connection to the database, query execution, data insertion, retrieval, and additional database operations.

Code Snippet 13 demonstrates an instance of using SQLAlchemy for interactions with a PostgreSQL database.

Code Snippet 13:

```
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base

# Create a database engine (replace with your PostgreSQL
#connection string)
engine = create_engine('postgresql://username:password@localhost:5432/my
database')

# Create a session
Session = sessionmaker(bind=engine)
session = Session()

# Define a SQLAlchemy model
Base = declarative_base()

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String)
# Create the table (if it does not exist)
Base.metadata.create_all(engine)

# Insert data
new_user = User(name='John')
session.add(new_user)
session.commit()
```

```

# Query data
users = session.query(User).all()
for user in users:
    print(user.name)

# Update data
user_to_update =
session.query(User).filter_by(name='John').first()
user_to_update.name = 'Jane'
session.commit()

# Delete data
user_to_delete =
session.query(User).filter_by(name='Jane').first()
session.delete(user_to_delete)
session.commit()

# Close the session
session.close()

```

Code Snippet 13 demonstrates SQLAlchemy usage with PostgreSQL. It creates a database engine and session, defines a SQLAlchemy model, creates a table if it does not exist, inserts data, queries data, updates data, deletes data, and closes the session.

Figure 12.10 depicts the output of this code when executed through PyCharm.

```

C:\Users\Abhishek Gupta\PycharmProjects\pythonProject5\main.py:13: MovedIn20Warning: Deprecated API features detected! These feature(s) are not compatible with SQLAlchemy 2.0. To prevent incompatible upgrades prior to updating applications, ensure requirements files are pinned to "sqlalchemy<2.0". Set environment variable SQLALCHEMY_WARN_20=1 to show all deprecation warnings. Set environment variable SQLALCHEMY_SILENCE_UBER_WARNING=1 to silence this message. (Background on SQLAlchemy 2.0 at: https://sqlalche.me/e/b8d9)
  Base = declarative_base()
John
Process finished with exit code 0

```

Figure 12.10: Output of Code Snippet 13

12.6 Summary

- Python utilizes libraries for HTML retrieval, image downloading, and server interaction via `HTTP` requests.
- Installing and setting up a MySQL database comprises installing MySQL Server, launching it, and configuring the database via MySQL client or command-line interface.
- Creating a database via Python involves establishing a connection to the MySQL server.
- Performing CRUD operations in Python involves using database-specific libraries, enabling data manipulation and management.
- Downloading and using tools in Python involves installing and importing relevant libraries or modules that extend the functionality of Python for specific tasks.

12.7 Test Your Knowledge

1. Which Python library is commonly used to access and parse HTML content from Web pages?
 - A. `json`
 - B. `requests`
 - C. `matplotlib`
 - D. `pandas`
2. What is the primary purpose of installing MySQL Server and configuring it during setup?
 - A. To create and manage databases
 - B. To write Python code
 - C. To access HTML content
 - D. To install additional Python libraries
3. Which SQL statement is typically used in Python to create a database from within a Python script?
 - A. `CREATE TABLE`
 - B. `INSERT INTO`
 - C. `CREATE DATABASE`
 - D. `SELECT FROM`
4. Which of the following SQL statements is used to read data from a database in Python when implementing CRUD operations?
 - A. `CREATE`
 - B. `UPDATE`
 - C. `INSERT`
 - D. `SELECT`
5. What is the primary purpose of using external libraries or tools in Python?
 - A. To increase code complexity
 - B. To make the code shorter
 - C. To extend functionality and simplify development
 - D. To improve code security

12.7.1 Answers to Test Your Knowledge

1. requests
2. To create and manage databases
3. CREATE DATABASE
4. SELECT
5. To extend functionality and simplify development

Try It Yourself

1. Create a Python program that reads a text file and counts the number of words in it.
2. Write a program that reads a csv file containing student names and their scores, calculates the average score, and displays the names of students who scored above the average.
3. Create a Python program that uses two threads to download multiple images concurrently from the Internet. Save each downloaded image to a local directory.
4. Build a program that simulates a simple ticket booking system with multiple booking agents (threads). Each booking agent should try to book tickets concurrently, and you should ensure that tickets are not overbooked.

Appendix

Sr. No	Case Studies
1	<p>Project 1: Task Manager</p> <p>Scenario: Create a task manager program with user-friendly features:</p> <ol style="list-style-type: none"> 1. Add tasks to the list. 2. View current task list. 3. Delete completed tasks. <p>Hints:</p> <ol style="list-style-type: none"> 1. Use lists to store the tasks. 2. Implement flow control statements to handle user input and execute operations. 3. Use functions to modularize the code and improve reusability. <p>Technology to be used:</p> <ol style="list-style-type: none"> 1. Python SDK 3.9.17 2. Tkinter 3. Pycharm 2023.1
2	<p>Project 2: File Encryption/Decryption</p> <p>Scenario: Create a program that can encrypt and decrypt files using a simple encryption algorithm. The program should perform following features.</p> <ol style="list-style-type: none"> 1. Prompt user to choose: Encrypt, Decrypt, or View encrypted file. 2. Encrypt files securely with selected algorithm. 3. Decrypt and display original contents effortlessly. <p>Hints:</p> <ol style="list-style-type: none"> 1. Encrypt/Decrypt files using American Standard Code for Information Interchange (ASCII) code data type conversions. 2. Utilize file handling for reading and writing. 3. Implement flow control for user input and operations. 4. Modularize code with functions for reusability. <p>Technology to be used:</p> <ol style="list-style-type: none"> 1. Python SDK 3.9.17 2. Pycharm 2023.1

Sr. No	Case Studies
3	<p>Project 3: Banking System</p> <p>Scenario: Develop a Banking Account Management System that should support following functionalities.</p> <ol style="list-style-type: none"> 1. Enable users to create new accounts by providing personal details. 2. Validate information and assign unique account numbers. 3. Allow users to check their account balances. 4. Facilitate deposits and withdrawals. 5. Maintain detailed transaction histories, including date, time, and transaction type. <p>Hints:</p> <ol style="list-style-type: none"> 1. Manage accounts and transactions using data structures. 2. Calculate balances and perform transfers with operators. 3. Handle scenarios such as insufficient funds with flow control. 4. Use functions for deposits and withdrawals. 5. Implement iterators for transaction history. 6. Log transactions and generate statements via file handling. 7. Manage errors with exception handling. 8. Ensure data persistence with serialization. <p>Technology to be used:</p> <ol style="list-style-type: none"> 1. Python SDK 3.9.17 2. Tkinter 3. MySQL 8.0.33 or higher 4. Pycharm 2023.1
4	<p>Project 4: Employee Attendance Tracker</p> <p>Scenario: Develop a system for company management that should include following features.</p> <ol style="list-style-type: none"> 1. Register employees and track their attendance. 2. Manage leave requests and approvals. 3. Generate reports on attendance, leaves, and overtime. 4. Calculate payroll based on attendance records. <p>Hints:</p> <ol style="list-style-type: none"> 1. Manage employee data and leave tracking with various data structures.

Sr. No	Case Studies
	<p>2. Control user input for attendance and leave requests. 3. Organize code with functions for reuse. 4. Store and retrieve data with serialization. 5. Log attendance and generate reports using files operations. 6. Store data in a relational database. 7. Ensure reliability with error handling. 8. Handle multiple requests with threading.</p> <p>Technology to be used:</p> <ol style="list-style-type: none"> 1. Python SDK 3.9.17 2. Tkinter 3. MySQL 8.0.33 or higher 4. Pycharm 2023.1
5	<p>Project 5: Expense Tracker</p> <p>Scenario: Develop an expense tracker application to help users manage their finances, that should include features listed as follows:</p> <ol style="list-style-type: none"> 1. User registration and profile management. 2. Record and categorize expenses. 3. Set budget limits for different categories. 4. Generate reports and visualizations of spending habits. 5. Provide alerts for exceeding budget limits. <p>Hints:</p> <ol style="list-style-type: none"> 1. Store expense data with various data structures. 2. Organize code with modular functions and parameters. 3. Use a relational database for data storage. 4. Employ serialization for user and expense data. 5. Implement exception handling for errors. <p>Technology to be used:</p> <ol style="list-style-type: none"> 1. Python SDK 3.9.17 2. Tkinter 3. MySQL 8.0.33 or higher 4. Pycharm 2023.1