



IBM®

# Innovation Centre for Education

## Python Programming



### Student Guide

Course code GAI02SG170

## April 2024 edition

### Notices

© Copyright International Business Machines Corporation 2024.

This document may not be reproduced in whole or in part without the prior written permission of IBM.

US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

*IBM Director of Licensing, IBM Corporation, North Castle Drive, MD-NC119, Armonk, NY 10504-1785, US*

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

### Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at [www.ibm.com/legal/copytrade.shtml](http://www.ibm.com/legal/copytrade.shtml).

# Contents

<b>Trademarks . . . . .</b>	<b>viii</b>
<b>Course description . . . . .</b>	<b>ix</b>
<b>Agenda . . . . .</b>	<b>xi</b>
<b>Unit 1. Introduction to Python Programming . . . . .</b>	<b>1-1</b>
Unit objectives . . . . .	1-2
Python fundamentals . . . . .	1-3
Python syntax and structure . . . . .	1-7
Writing and executing python programs . . . . .	1-12
Python basics . . . . .	1-15
Python variables . . . . .	1-17
Python reserved words/keywords . . . . .	1-19
Python data types . . . . .	1-22
Operators and expressions . . . . .	1-34
Input and output operations . . . . .	1-41
Self evaluation: Exercise 1 . . . . .	1-44
Self evaluation: Exercise 2 . . . . .	1-45
Self evaluation: Exercise 3 . . . . .	1-46
Self evaluation: Exercise 4 . . . . .	1-47
Self evaluation: Exercise 5 . . . . .	1-48
Checkpoint (1 of 2) . . . . .	1-49
Checkpoint (2 of 2) . . . . .	1-50
Question bank (1 of 2) . . . . .	1-51
Question bank (2 of 2) . . . . .	1-52
Unit summary . . . . .	1-53
<b>Unit 2. Control Flow and Loops. . . . .</b>	<b>2-1</b>
Unit objectives . . . . .	2-2
Introduction . . . . .	2-3
Logical operators and conditions . . . . .	2-8
Switch-case (if-elif-else) . . . . .	2-11
Loops . . . . .	2-17
For loops . . . . .	2-22
Loop control statements . . . . .	2-25
Sorted vs. Sort in python . . . . .	2-29
Self evaluation: Exercise 6 . . . . .	2-32
Self evaluation: Exercise 7 . . . . .	2-33
Self evaluation: Exercise 8 . . . . .	2-34
Self evaluation: Exercise 9 . . . . .	2-35
Self evaluation: Exercise 10 . . . . .	2-36
Checkpoint (1 of 2) . . . . .	2-37
Checkpoint (2 of 2) . . . . .	2-38
Question bank (1 of 2) . . . . .	2-39
Question bank (2 of 2) . . . . .	2-40
Unit summary . . . . .	2-41
<b>Unit 3. Object-Oriented Programming (OOP) in Python. . . . .</b>	<b>3-1</b>
Unit objectives . . . . .	3-2

Advantages of Object-Oriented Programming (OOP) .....	3-3
OOP principles .....	3-5
Defining classes and objects .....	3-7
Constructor in python .....	3-10
Destructor in python .....	3-13
Types of methods .....	3-15
Class methods .....	3-17
Static methods .....	3-19
Type of variables .....	3-22
Advanced OOP concepts .....	3-27
Inheritance types .....	3-29
Polymorphism .....	3-34
Polymorphism with abstract classes .....	3-39
Encapsulation and abstraction .....	3-41
Achieving abstraction through interfaces .....	3-46
Self evaluation: Exercise 11 .....	3-48
Self evaluation: Exercise 12 .....	3-49
Self evaluation: Exercise 13 .....	3-50
Self evaluation: Exercise 14 .....	3-51
Self evaluation: Exercise 15 .....	3-52
Checkpoint (1 of 2) .....	3-53
Checkpoint (2 of 2) .....	3-54
Question bank (1 of 2) .....	3-55
Question bank (2 of 2) .....	3-56
Unit summary .....	3-57
<b>Unit 4. Error Handling and Exception Handling .....</b>	<b>4-1</b>
Unit objectives .....	4-2
Error handling and exception handling .....	4-3
Understanding exceptions in python .....	4-5
NameError .....	4-11
Handling exceptions with try and except .....	4-14
Handling multiple exception .....	4-16
Finally block .....	4-18
Custom exceptions .....	4-20
Raising exceptions .....	4-22
File handling (I/O) .....	4-24
Reading and writing files .....	4-25
Reading and writing text files .....	4-28
Working with text and binary files .....	4-31
Text encoding and decoding .....	4-35
File handling best practices .....	4-37
Error handling in file operations .....	4-39
Self evaluation: Exercise 16 .....	4-42
Self evaluation: Exercise 17 .....	4-43
Self evaluation: Exercise 18 .....	4-44
Self evaluation: Exercise 19 .....	4-45
Self evaluation: Exercise 20 .....	4-46
Checkpoint (1 of 2) .....	4-47
Checkpoint (2 of 2) .....	4-48
Question bank (1 of 2) .....	4-49
Question bank (2 of 2) .....	4-50
Unit summary .....	4-51
<b>Unit 5. Introduction to Data &amp; Business Analytics .....</b>	<b>5-1</b>
Unit objectives .....	5-2

Advanced s .....	5-3
Pattern matching and text processing (1 of 2) .....	5-6
Pattern matching and text processing (2 of 2) .....	5-12
Regex functions and methods .....	5-17
Modules and libraries .....	5-22
Standard library modules .....	5-26
Third-party libraries and packages .....	5-34
Self evaluation: Exercise 21 .....	5-37
Self evaluation: Exercise 22 .....	5-38
Self evaluation: Exercise 23 .....	5-39
Self evaluation: Exercise 24 .....	5-40
Checkpoint (1 of 2) .....	5-41
Checkpoint (2 of 2) .....	5-42
Question bank (1 of 2) .....	5-43
Question bank (2 of 2) .....	5-44
Unit summary .....	5-45
<b>Unit 6. Graphical User Interfaces (GUI) and Web programming.....</b>	<b>6-1</b>
Unit objectives .....	6-2
Introduction .....	6-3
GUI development .....	6-4
GUI vs. Command line interfaces .....	6-6
GUI frameworks in python .....	6-7
Widgets and event handling .....	6-9
Creating widgets (1 of 2) .....	6-11
Creating widgets (2 of 2) .....	6-17
Handling user events (clicks, input) .....	6-22
GUI development in python: Example codes .....	6-25
Web programming with CGI .....	6-29
What is CGI and its purpose .....	6-31
Handling HTTP requests .....	6-34
Receiving and processing requests .....	6-36
Generating HTTP responses .....	6-42
Building interactive web applications .....	6-50
Form handling with CGI .....	6-51
Implementing data processing .....	6-54
Self evaluation: Exercise 25 .....	6-58
Self evaluation: Exercise 26 .....	6-59
Self evaluation: Exercise 27 .....	6-60
Self evaluation: Exercise 28 .....	6-61
Self evaluation: Exercise 29 .....	6-62
Checkpoint (1 of 2) .....	6-63
Checkpoint (2 of 2) .....	6-64
Question bank (1 of 2) .....	6-65
Question bank (2 of 2) .....	6-66
Unit summary .....	6-67
<b>Unit 7. Python Applications .....</b>	<b>7-1</b>
Unit objectives .....	7-2
Introduction .....	7-3
Networking and serialization .....	7-4
Networking basics in python .....	7-5
Introduction to networking protocols .....	7-6
Creating client and server sockets .....	7-9
Socket programming .....	7-12
Building networked applications .....	7-13

Data transfer and communication .....	7-16
An example code .....	7-18
Serialization (JSON and Pickle) .....	7-21
Serialization overview .....	7-23
JSON and Pickle for data serialization (1 of 2) .....	7-25
JSON and Pickle for data serialization (2 of 2) .....	7-28
Data processing and analysis .....	7-31
Introduction to NumPy and pandas .....	7-33
What is NumPy? .....	7-34
Overview of pandas .....	7-36
Data manipulation with NumPy .....	7-38
Array operations and manipulation .....	7-42
Broadcasting .....	7-44
Numpy operations .....	7-47
Data analysis with pandas .....	7-50
Working with DataFrame .....	7-52
Creating pandas dataframe .....	7-54
Retrieving labels and data .....	7-57
Data cleaning and exploration .....	7-60
Handling missing values .....	7-61
Dealing with duplicates .....	7-67
Database applications .....	7-69
Connecting to databases .....	7-71
SQL queries and database operations .....	7-73
Structured Query Language (SQL) .....	7-75
Executing SQL queries from python (1 of 2) .....	7-77
Executing SQL queries from python (2 of 2) .....	7-79
Building database-driven applications .....	7-82
Integrating python with databases .....	7-84
CRUD operations in database applications .....	7-86
Self evaluation: Exercise 30 .....	7-89
Self evaluation: Exercise 31 .....	7-90
Self evaluation: Exercise 32 .....	7-91
Self evaluation: Exercise 33 .....	7-92
Self evaluation: Exercise 34 .....	7-93
Self evaluation: Exercise 35 .....	7-94
Self evaluation: Exercise 36 .....	7-95
Self evaluation: Exercise 37 .....	7-96
Self evaluation: Exercise 38 .....	7-97
Self evaluation: Exercise 39 .....	7-98
Checkpoint (1 of 2) .....	7-99
Checkpoint (2 of 2) .....	7-100
Question bank (1 of 2) .....	7-101
Question bank (2 of 2) .....	7-102
Unit summary .....	7-103
<b>Unit 8. Advanced Cloud Topics and Case Studies.....</b>	<b>8-1</b>
Unit objectives .....	8-2
Solution: Exercise 1 .....	8-3
Solution: Exercise 2 .....	8-5
Solution: Exercise 3 .....	8-7
Solution: Exercise 4 .....	8-9
Solution: Exercise 5 .....	8-11
Solution: Exercise 6 .....	8-13
Solution: Exercise 7 .....	8-15
Solution: Exercise 8 .....	8-17

Solution: Exercise 9 . . . . .	8-19
Solution: Exercise 10 . . . . .	8-21
Solution: Exercise 11 . . . . .	8-23
Solution: Exercise 12 . . . . .	8-25
Solution: Exercise 13 . . . . .	8-27
Solution: Exercise 14 . . . . .	8-29
Solution: Exercise 15 . . . . .	8-31
Solution: Exercise 16 . . . . .	8-33
Solution: Exercise 17 . . . . .	8-35
Solution: Exercise 18 . . . . .	8-37
Solution: Exercise 19 . . . . .	8-39
Solution: Exercise 20 . . . . .	8-41
Solution: Exercise 21 . . . . .	8-43
Solution: Exercise 22 . . . . .	8-45
Solution: Exercise 23 . . . . .	8-47
Solution: Exercise 24 . . . . .	8-49
Solution: Exercise 25 . . . . .	8-51
Solution: Exercise 26 . . . . .	8-56
Solution: Exercise 27 . . . . .	8-58
Solution: Exercise 28 . . . . .	8-60
Solution: Exercise 29 . . . . .	8-63
Solution: Exercise 30 . . . . .	8-66
Solution: Exercise 31 . . . . .	8-70
Solution: Exercise 32 . . . . .	8-72
Solution: Exercise 33 . . . . .	8-75
Solution: Exercise 34 . . . . .	8-77
Solution: Exercise 35 . . . . .	8-79
Solution: Exercise 36 . . . . .	8-82
Solution: Exercise 37 . . . . .	8-84
Solution: Exercise 38 . . . . .	8-86
Solution: Exercise 39 . . . . .	8-88
Solution: Exercise 40 . . . . .	8-91
Unit summary . . . . .	8-93

<b>Appendix A. Review answers . . . . .</b>	<b>A-1</b>
---	------------

---

# Trademarks

The reader should recognize that the following terms, which appear in the content of this training document, are official trademarks of IBM or other companies:

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide.

The following are trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide:

Bluemix®  
SPSS®

Cognitive Era™  
Watson Avatar®

Cognos®  
Worklight®

Windows is a trademark of Microsoft Corporation in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other product and service names might be trademarks of IBM or other companies.

# Course description

## Python

### Purpose

The course "Python Programming" provides a comprehensive introduction to the Python programming language. It covers various topics, including Python fundamentals, control flow, object-oriented programming, error handling, file handling, regular expressions, modules, GUI development, CGI programming, and Python applications. Students will gain practical experience in writing and executing Python programs, handling exceptions, working with files, and building interactive applications. This course is designed for learners looking to develop a strong foundation in Python programming.

### Audience

B.E engineering students.

### Prerequisites

Basic understanding of programming concepts like variables, data types, control flow, and functions.

### Objectives

- Understand the fundamentals of the Python programming language and its historical development
- Master Python basics, including data types, operators, and input/output operations
- Implement control statements and loops to make decisions and perform iterative tasks
- Explore object-oriented programming (OOP) concepts and apply them to create reusable and modular code
- Handle exceptions and errors effectively to ensure robustness in Python programs
- Work with files and perform input/output (I/O) operations to read and write data
- Learn regular expressions and use them for text processing and pattern matching
- Gain proficiency in working with modules to modularize Python code and enhance reusability
- Develop graphical user interfaces (GUI) using various widgets and events
- Understand the basics of Common Gateway Interface (CGI) and create interactive web applications
- Explore Python applications, including networking, serialization, and data processing with NumPy and Pandas, along with database applications

### Course outcomes

- Upon completion of the course, learners will be able to:
  - Demonstrate a solid understanding of Python programming language and its syntax
  - Implement Python basics and effectively work with variables, data types, and collections

- Apply control statements and loops to solve complex problems and execute iterative tasks
- Develop object-oriented Python code, encapsulating data and behavior in classes and objects
- Handle exceptions and errors gracefully to ensure smooth program execution
- Perform input/output (I/O) operations to read from and write to files and streams
- Utilize regular expressions for efficient text processing and pattern matching
- Modularize Python code using modules and libraries to enhance code maintainability and reusability
- Create graphical user interfaces (GUI) with various widgets and handle user events
- Build interactive web applications using CGI in Python for data processing and form handling
- Implement Python applications, including networking, serialization, and data processing with NumPy and Pandas, along with database applications

## References

- "Python Crash Course" by Eric Matthes
- "Automate the Boring Stuff with Python" by Al Sweigart
- "Learning Python, 5th Edition" by Mark Lutz
- "Python Programming: An Introduction to Computer Science" by John Zelle
- "Python for Data Analysis" by Wes McKinney
- "Fluent Python" by Luciano Ramalho
- "Python Cookbook" by David Beazley and Brian K. Jones
- "Python Pocket Reference" by Mark Lutz
- "Python GUI Programming Cookbook" by Burkhard A. Meier

---

# Unit 1. Introduction to Python Programming

## Overview

This unit will provide an overview of Understand the fundamentals of the Python programming language and its historical development. Master Python basics, including data types, operators, and input/output operations.

## How you will check your progress

- Checkpoint

## References

IBM Knowledge Center



IBM ICE (Innovation Centre for Education)

## Unit objectives

- After completing this unit, you should be able to:
  - Understand the fundamentals of the Python programming language and its historical development
  - Master Python basics, including data types, operators, and input/output operations
- Learning outcomes:
  - Demonstrate a solid understanding of Python programming language and its syntax
  - Implement Python basics and effectively work with variables, data types, and collections

---

Figure 1-1. Unit objectives

Unit objectives and outcomes are as stated above.

## Python fundamentals



IBM ICE (Innovation Centre for Education)

- Python fundamentals encompass syntax simplicity, dynamic typing, and support for variables, data types, and operators.
- Control flow structures like conditional statements and loops, along with functions and modular design, contribute to code organization and readability.



Figure 1: Hardware and software

Figure 1-2. Python fundamentals

Python's versatile data structures, including lists, tuples, and dictionaries, enhance data manipulation. With object-oriented programming support, extensive libraries, and a robust community, Python is a powerful language for diverse applications, from scripting and web development to data science and artificial intelligence.

### Introduction to Python

Python was created by Guido van Rossum and first released in 1991. It has since become one of the most popular programming languages globally, known for its simplicity and versatility.

Hardware is the visible, physical components of the computer, such as the keyboard, mouse, processor, and RAM. Whereas software comprises logical components, software is not visible but is essential for making the hardware components work, as depicted in figure 1. There has been significant development in both hardware and software over the last few decades, resulting in substantial restructuring and improved performance. Python, as a programming language, falls under the category of software. Software can be broadly classified into three categories:

- Utility software: Commonly used for specific activities within a computer. Examples include antivirus programs, Microsoft Office, and Tally.
- Application software: This category encompasses programming languages that assist in building other software; examples include C, C++, Java, and Python.
- System software: This category facilitates communication between computer hardware and users, managing the working of the hardware and handling interfacing among components. Example: Operating system, Compiler, Linker, Loader.

## What is Python?

Python, a versatile, high-level programming language renowned for its readability and ease of use, is suitable for a wide range of applications, from web development to data science. Designed for simplicity and code readability, python's syntax is straightforward, allowing programmers to express concepts without restrictions in code length. It serves as the primary programming language for executing projects on Raspberry pi, and renowned companies such as Google, Nokia, Disney, Yahoo, and IBM incorporate python in their operations. Beyond being a scripting language, python is open-source and supports both web development and database connectivity, functioning as a versatile language that is object-oriented, procedural, and functional. Dynamic type systems and automatic memory management are some of the remarkable qualities of python. Python also has a very large collection of standard library routines, along with its capacity to incorporate multiple external packages. With various features and practices followed in python, it is one of the most preferred programming languages in multiple areas nowadays.

Python is both a compiled and interpreted language. When a python script is run, it is first compiled into an intermediate form called bytecode. The python interpreter then interprets this bytecode, and the instructions are executed. Python employs a combination of compilation and interpretation to achieve a balance between performance and flexibility. The bytecode is created during the compilation step, and this bytecode is executed on the Python Virtual Machine (PVM) during the interpretation step. The dynamic and flexible features of python are maintained through this hybrid approach while still ensuring a level of performance.

## Python's popularity and use cases

Python has witnessed an unprecedented surge in popularity, becoming one of the most widely used and versatile programming languages. Several factors contribute to python's widespread adoption:

- Ease of learning and readability: Python's syntax is clear and readable, making it an excellent language for beginners. Its simplicity accelerates the learning curve and facilitates collaboration among developers.
- Extensive standard library: Python comes with a comprehensive standard library that provides ready-to-use modules and packages for various tasks. This minimizes the need for developers to build functionalities from scratch, saving time and effort.
- Community support: Python boasts a vibrant and supportive community. The python community actively contributes to the language's development, offers assistance through forums, and shares a wealth of resources, making it easier for developers to find solutions to their queries.
- Versatility: Python is a versatile language that supports various programming paradigms, including object-oriented, procedural, and functional programming. This flexibility makes it suitable for a wide range of applications, from web development to data science and artificial intelligence.
- Web development: Python frameworks such as Django and flask have gained popularity in web development. They provide robust tools for building scalable and maintainable web applications, making python a preferred choice for web developers.
- Automation and scripting: Python's simplicity and readability make it an ideal choice for automation and scripting tasks. Many system administrators and DevOps professionals use python to automate repetitive tasks and streamline workflows.
- Education: Python's beginner-friendly nature has made it a staple in educational institutions. Many universities and coding boot camps use python as the introductory language for programming courses.
- Artificial Intelligence (AI) and Internet Of Things (IoT): Python's adaptability has made it a language of choice for ai development and IoT projects. Its compatibility with raspberry pi and other IoT devices has fueled its use in this burgeoning field.
- Data science and analytics: Python is a dominant language in the field of data science. Libraries like Pandas, NumPy, and Scikit-learn are widely used for data manipulation, analysis, and machine learning. For instance, a data scientist might use python to analyze customer behavior data and develop predictive models to optimize marketing strategies.
- Scientific computing: Python is widely employed in scientific computing for tasks like numerical simulations and data visualization.

- The scientific community often uses python along with libraries such as SciPy and matplotlib. For instance, researchers might use python to model and simulate the behavior of complex systems in physics or biology.
- Network programming: Python is commonly used for network programming tasks, including the development of network applications and protocols. The simplicity of the language makes it suitable for tasks such as socket programming. For example, a network engineer might use python to create a network monitoring tool that analyzes network traffic and identifies potential issues.
- Game development: Python is employed in the gaming industry for scripting, automation, and developing game prototypes. PyGame is a popular library for game development in python. Game developers might use python to create game scripts or design prototypes to test game mechanics before full-scale development.

### **Python use cases:**

The case studies given below highlight python's widespread adoption in industry-leading companies and organizations, showcasing its effectiveness in solving complex challenges across diverse sectors.

### **Desktop applications:**

Python's popularity in desktop application development stems from its simplicity, cross-platform compatibility, and strong GUI frameworks like Tkinter, PyQt, and Kivy. Developers can quickly prototype applications, design visually appealing interfaces, and seamlessly integrate with other technologies. Python's versatility is evident in its use across a range of desktop applications, including Microsoft Office, Adobe Photoshop, Skype, Discord, Slack, and more. Leveraging Python's rapid prototyping and integration capabilities, developers create feature-rich, cross-platform desktop solutions tailored to diverse industries and needs.

### **Mobile applications:**

Python's versatility extends beyond desktop applications to mobile app development, where it is compatible with various platforms, including Android and iOS. With frameworks like Kivy, BeeWare, and PyQt, developers can leverage Python's simplicity and cross-platform capabilities to create mobile applications that run seamlessly on different devices and operating systems. Python's extensive libraries and community support further enhance its utility in mobile app development, enabling developers to build feature-rich, user-friendly applications for a wide range of purposes and industries.

### **Robotics:**

Python's utility extends significantly in the realm of robotics, primarily due to its capability for rapid prototyping, facilitating the swift testing of novel ideas and algorithms. With the proliferation of robotics applications, Python has emerged as a straightforward language to learn. Moreover, Python Remote Objects (Pyro) serves as a valuable library for constructing applications, enabling seamless communication between objects over networks. Additionally, Pybotics stands as an open-source Python toolbox dedicated to robot kinematics and calibration, further augmenting Python's role in advancing robotics technology.

### **Instagram:**

- Challenge: Dealing with massive image and video data uploads.
- Solution: Instagram employs python extensively for backend development. Django, a python web framework, is utilized to manage the high volume of content and facilitate seamless interactions on the platform.

### **Nasa:**

- Challenge: Analyzing vast amounts of data from space missions.
- Solution: Python is used for scientific computing and data analysis at nasa. Libraries like numpy and scipy assist scientists in processing and interpreting data from telescopes and satellites.

### **Dropbox:**

- Challenge: Building a reliable and scalable cloud storage service.

- Solution: Dropbox's server-side components are primarily written in python. The simplicity and readability of python contributed to faster development cycles and efficient maintenance of the cloud storage infrastructure.

### **Reddit:**

Challenge: Managing a large-scale social news aggregation platform.

Solution: Reddit's backend is powered by python. The use of the pylons framework and other python libraries allows reddit to handle a massive number of user interactions and content submissions effectively.

### **Netflix:**

- Challenge: Enhancing content recommendations for users.
- Solution: Netflix employs python for its machine learning algorithms, especially for content recommendation. Python's machine learning libraries, such as sci-kit-learn, enable Netflix to personalize user recommendations based on viewing history.

### **Spotify:**

- Challenge: Delivering personalized music recommendations to users.
- Solution: Python is utilized in Spotify for data analysis and algorithm development. The company uses python's data science libraries to analyze user preferences and behaviors, creating tailored playlists and recommendations.

### **Tesla:**

- Challenge: Developing software for electric vehicles.
- Solution: Tesla utilizes python for various aspects, including vehicle software and automation in manufacturing processes. Python's versatility allows tesla engineers to integrate and deploy software updates efficiently.



Figure 2. Python applications

Source: [<https://www.tech-act.com/blog/data-science/what-is-python/>]



IBM ICE (Innovation Centre for Education)

# Python syntax and structure

- Python is known for its clear and readable syntax, which contributes to its popularity and ease of learning.
- The syntax and structure of Python are designed to be clear and readable, facilitating ease of understanding and code maintenance.
- Python's syntax is enforced by the interpreter, ensuring consistency in coding practices. Code blocks are defined by indentation, promoting a clean and organized structure.
- Python's indentation and block structure:
  - In Python, indentation and block structure are crucial elements that determine the grouping of statements and the scope of code.
  - Unlike many programming languages that use braces or keywords to define code blocks, python relies on indentation to signify the beginning and end of blocks of code.
- Comments:
  - Comments play a crucial role in programming languages by providing insights into the thought process behind the code, aiding in understanding, debugging, and code reuse.
  - Well-commented code contributes to code improvement and error identification.

*Figure 1-3. Python syntax and structure*

## Python syntax and structure

Variables and objects are dynamically typed, allowing for flexibility in programming. Functions and methods are defined using the def keyword, and python supports both procedural and Object-Oriented Programming paradigms. Importantly, the use of colons and whitespace in python enhances code readability and enforces a structured approach to coding conventions. Overall, python's syntax and structure are crafted to prioritize simplicity and readability, contributing to the language's popularity and ease of use in various domains.

## Python's indentation and block structure

The primary purpose of indentation in python is to improve code readability. Key points about python indentation and block structure are:

- **Indentation:** Indentation is the number of spaces or tabs at the beginning of a line. Consistent indentation is essential for proper code execution. The python community conventionally uses four spaces for each level of indentation, although tabs can be used as well.
- **Code blocks:** Code blocks are groups of statements that are executed together. The beginning of a code block is indicated by indentation. The end of a code block is denoted by a decrease in indentation or the end of the file.
- **Colon (:) usage:** A colon (:) is used to signal the start of an indented code block.

For example:

```
If condition:  
  # Indented block
```

```
Statement1
```

```
Statement2
```

- **Consistency is key:** All statements within the same block must have the same level of indentation. Mixing spaces and tabs for indentation can lead to errors, so it's recommended to use one or the other consistently.
- **Nested blocks:** Blocks of code can be nested within each other by adding additional levels of indentation. For example:

```
If condition:
```

```
    # Outer block
```

```
    Statement1
```

```
    If nested_condition:
```

```
        # Nested block
```

```
        Statement2
```

- **Whitespace sensitivity:** Python is whitespace-sensitive, meaning that the indentation is not just for readability but is part of the syntax. Incorrect indentation can result in syntax errors or unexpected behavior. Example of indentation in a function:

```
Def example_function():
```

```
    # Indented block
```

```
    Print("this is part of the function")
```

```
    If condition:
```

```
        # Nested block
```

```
        Print("this is part of a nested block")
```

```
    # Back to the outer block
```

```
    Print("back to the outer block")
```

In summary, python indentation and block structure are fundamental for organizing code and defining the scope of statements within the language. Proper indentation enhances code readability and is a key aspect of python's design philosophy. Notepad++ and Visual Studio are widely used editors with robust support for automatic indentation. In Notepad++, the auto-indent feature preserves the indentation level of the line above it. Therefore, when you indent a line while typing, Notepad++ retains that same indentation for subsequent lines until you manually adjust the indent level.

## Comments

Comments are essential in projects of any scale, but it is crucial to provide them appropriately. Poorly written comments hinder code comprehension, emphasizing the importance of clear and purposeful commenting.

- **Single-line comment:** Python supports both single-line and multi-line comments. Single-line comments in python start with a hash character (#) followed by the explanatory text.

```
num = 10 # initialize number as 10
```

- **Multi-line comment:** It's recommended to limit the size of a single-line comment to 79 characters, and if it exceeds this limit, it's advisable to split the text into multiple lines. Multi-line comments in python are achieved by grouping single-line comments. Although python does not have explicit syntax for multi-line comments, developers utilize consecutive single-line comments for inline documentation.

```

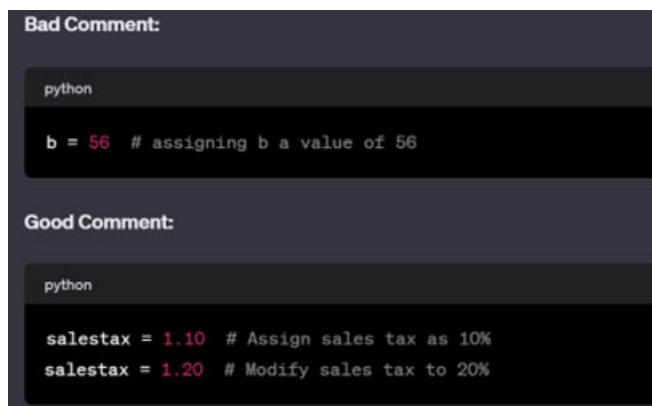
# This is a multiline comment in Python
# The comments are placed one after another
# to provide detailed explanations or documentation.

def example_function():
    """
    This is another way to create a multiline comment
    using triple double-quotes. This is often used for
    docstrings, which are used to document functions, classes, etc.
    """
    print("Hello, World!")

# Call the example function
example_function()

```

**Note:** regardless of the type, comments should always be clear and understandable. A comment that fails to describe the purpose of the code is considered a bad comment. Here's a comparison:

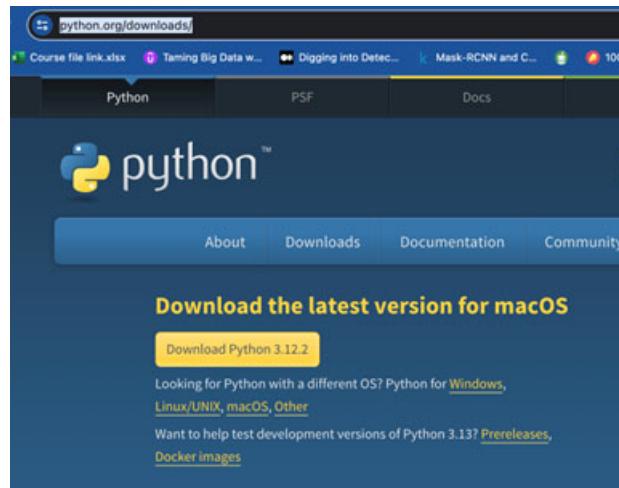


## Writing your first python program

Writing the first python program involves a straightforward and engaging process. The following step-by-step instructions outline the creation of a basic python program, commonly known as the "hello, world!" Program:

### Step 1: Install Python

- Visit the official Python website (<https://www.python.org/downloads/>)
- Download and install the latest version of Python (3.12) for your operating system.



## Step 2: Choose a text editor or IDE

- You can use any basic text editor like Notepad (Windows) orTextEdit (Mac).
- Consider using an Integrated Development Environment (IDE) like Visual Studio Code, PyCharm, or IDLE for a more advanced and user-friendly experience.

## Step 3: Open the editor or IDE

Launch your chosen text editor or IDE.

```

IDLE Shell 3.12.1
Python 3.12.1 (v3.12.1:2305ca5144, Dec 7 2023, 17:23:38) [Clang 13.0.0 (clang-1300.0.29.30)]
Type "help", "copyright", "credits" or "license()" for more information.

>>> |

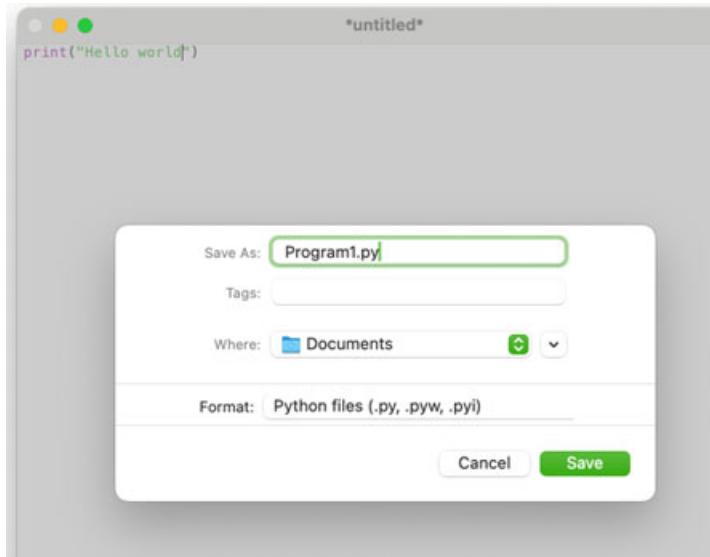
```

## Step 4: Write the Python code

- Type the following code into your editor
- `print("Hello, World!")`

## Step 5: Save the file

- Save the file with a name ending in ".py" (e.g., "Program1.py") to indicate it's a Python script.



## Step 6: Run the program

### Option 1: Using a terminal or command prompt

- Open a terminal or command prompt window.
- Navigate to the directory where you saved the script using the "cd" command (e.g., "cd Desktop").

- Type the command `python hello_world.py` and press Enter.

Option 2: Using an IDE

- Most IDEs have a "Run" or "Execute" button to run the code directly.

**Expected output:**

The program should print "Hello, World!" on the screen.



IBM ICE (Innovation Centre for Education)

# Writing and executing python programs

- Executing Python programs can be achieved in various ways, depending on the environment and the nature of the program.
- Here are a few common methods:
  - [Using python interactive mode \(REPL\)](#):
    - Python interactive mode, often referred to as REPL (Read-Eval-Print Loop), is a powerful tool for quickly testing code snippets, exploring language features, and debugging.
    - An interactive environment is provided where Python code can be entered, and the interpreter immediately executes it, displaying the results.
  - [Running python scripts from command line](#):
    - Running Python scripts from the command line is a common practice in software development and automation.
  - [Exploring python idle](#):
    - The Python IDLE (Integrated Development and Learning Environment) is a simple yet powerful integrated development environment that comes bundled with the Python programming language.
    - It provides a user-friendly interface for writing, running, and testing Python code. The IDLE environment can be launched by searching for "IDLE" in the start menu or by typing idle in the command prompt or terminal.

*Figure 1-4. Writing and executing python programs*

## Using python interactive mode (REPL)

Here's a brief note on using Python interactive mode:

- Accessing REPL: Open a terminal or command prompt. Type python or python3 and press enter to enter the interactive mode.
- The prompt: Once in the interactive mode, the python prompt (>>>) appears, indicating that python is ready to receive commands.
- Executing code: Enter python code directly at the prompt and press enter.

```
(base) deepajoshi@Deepas-MacBook-Air ~ % python
Python 3.11.5 (main, Sep 11 2023, 08:31:25) [Clang 14.0.6 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 10+20
30
>>> print("hello world")
hello world
>>>
```

- Exiting the interactive mode: To exit the interactive mode, type exit(), quit(), or press ctrl + d (on Unix/Linux) or ctrl + z (on windows).

- Considerations: While REPL is excellent for quick experimentation, it might not be suitable for large projects. For extensive development, consider using a code editor or Integrated Development Environment (IDE).

Valuable for learning, testing, and debugging, python interactive mode serves as a resource. Whether Python syntax is being explored by a beginner or ideas are being prototyped by an experienced developer, a quick and convenient way to interact with Python in real-time is provided by the REPL environment.

### **Running python scripts from command line**

The steps to run a python script using the command line are as follows:

- A command prompt or terminal should be opened, with the command prompt or powershell used on windows and the terminal used on unix-based systems (Linux or MacOs)
- The cd command (change directory) is used to navigate to the directory where the python script is located.
- The python script is executed using the python command followed by the script's filename.

Python script.py

- Command-line arguments, if accepted by the script, can be passed after the script's filename.

Python script.py arg1 arg

- For users employing virtual environments, it is optional to activate the virtual environment before running the script.

```
(base) deepajoshi@Deepas-MacBook-Air ~ % cd documents
(base) deepajoshi@Deepas-MacBook-Air documents % ls
BT-H-CSE-CSF-VII-B2 Sem 7 Major Project 2.numbers
BT-H-CSE-CSF-VII-B2 Sem 7 Major Project.numbers
Blooms taxonomy.key
DP_patterns_examples.numbers
Hackathon (1).key
Hackathon 2.key
Program1.py
Question Bank Unit-7.pages
UPES
Unit2 _Revision_reply.pages
marks AC.numbers
pick.py
serialized_data.pkl
(base) deepajoshi@Deepas-MacBook-Air documents % python pick.py
Deserialized Python object: {'name': 'Jane', 'age': 28, 'city': 'San Francisco'}
(base) deepajoshi@Deepas-MacBook-Air documents %
```

### **Exploring python IDLE**

Upon opening, IDLE presents an interactive shell window, allowing users to enter python code and see the results immediately. It features a script editor for writing and saving python scripts, with the ability to create a new script through file -> new file or ctrl + N and save scripts via file -> save or ctrl + S.

- IDLE can be launched by searching for "IDLE" in the start menu or by typing idle in the command prompt or terminal.
- Upon opening, idle presents an interactive shell window, allowing users to enter python code and see the results immediately. It features a script editor for writing and saving python scripts, with the ability to create a new script through file -> new file or ctrl + N and save scripts via file -> save or ctrl + S.
- Running scripts in idle is accomplished by executing a script through run -> run module or pressing f5, with the output displayed in the interactive shell. IDLE supports basic interactive debugging, accessible through run -> debug module or alt + F5.

Features of python idle include:

- Script editor: Use the script editor to write and save python scripts. Create a new script by selecting file -> new file or pressing ctrl + N. Save the script with file -> save or ctrl + S.

- Running scripts: Execute a script by selecting run -> run module or pressing f5. The output will be displayed in the interactive shell.
- Interactive debugging: Idle provides a basic interactive debugger. Use run -> debug module or press alt + F5 to start debugging.
- Autocomplete: Idle supports autocomplete, aiding in writing code more efficiently. Press the tab to complete partially typed words.
- Access python documentation and help by selecting help -> python docs or pressing f1.
- Configurable preferences: customize the idle experience through options -> configure idle.
- To exit idle, close the windows or select file -> exit.

**Note:** Refer the video in the below link for step-by-step execution of the above steps-

[https://sites.pitt.edu/~naraehan/python3/getting\\_started\\_win\\_first\\_try.html](https://sites.pitt.edu/~naraehan/python3/getting_started_win_first_try.html)



# Python basics

- Python identifiers:
  - In Python, an identifier is a name given to entities such as variables, functions, classes, modules, or other objects.
  - Identifiers help in uniquely identifying these entities in a program. Here are some rules and conventions regarding python identifiers.

```
# Valid identifiers

my_variable = 10
myFunction = lambda x: x * 2
MyClass = class MyClass:
    def __init__(self):
        self.my_attribute = 42

# Invalid identifiers (due to starting with a number)
2variable = 5
```

Figure 1-5. Python basics

## Rules for Python identifiers:

- Valid characters: It can include letters (both uppercase and lowercase), numbers, and underscore (\_). Starts with a letter or an underscore: It must start with a letter (a-z, A-Z) or an underscore (\_).
- Case-sensitive: Python is case-sensitive, so myvar and myvar are different identifiers.
- No special characters are allowed in an identifier. Python does not allow punctuations like @, \$, and % to be used in identifiers.
- Reserved words: Identifiers cannot be a reserved word in Python.
- White space: It should not contain white space.

## Conventions for Python identifiers:

- Snake case: For variable names, use lowercase with underscores between words (e.g., my\_variable).
- Camel case: For function and method names, use lowercase for the first word and capitalize the first letter of each subsequent concatenated word (e.g., myfunction).
- Pascal case: For class names, capitalize the first letter of each word (e.g., myclass).
- Single leading underscore: By convention, a single leading underscore indicates a weak "internal use" indicator. It is not enforced by the interpreter.
- Double leading underscore: By convention, a double leading underscore indicates a "strong internal use" or a name mangling indicator. It is used to make an attribute or method more difficult to access from outside the class.

- Double leading and trailing underscores: Identifiers surrounded by double leading and trailing underscores (e.g., `__init__`) are used for special methods or attributes.

Choosing meaningful and descriptive identifiers is deemed crucial for the composition of readable and maintainable code. Adhering to Python naming conventions is advocated as it contributes to rendering code more consistent and understandable for both the author and other stakeholders.



IBM ICE (Innovation Centre for Education)

# Python variables

- In Python, variables are symbolic names that represent memory locations storing data. They are dynamically typed, allowing flexibility in assigning values of different types.
- Variable names must adhere to certain rules, including starting with a letter or underscore, and can include letters, numbers, and underscores.
- Python supports various data types for variables, such as integers, floats, strings, lists, and more.
- Variables play a pivotal role in programming, facilitating the storage and manipulation of data within a program.
- Understanding how to declare, assign, and use variables is fundamental to writing efficient and expressive Python code.

*Figure 1-6. Python variables*

### Variable declaration and assignment

```
name = "John"
```

```
age = 25
```

```
height = 1.75
```

```
is_student = True
```

### Displaying variable values

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

### Modifying variable values-

```
age = 26
```

```
height += 0.1
```

### Displaying updated values

```
print(f"Updated Age: {age}, Updated Height: {height}")
```

In this example, we declare and assign values to variables (name, age, height, is\_student). We then display these values using the print() function. Afterward, we modify the age and height variables and display the updated values. This showcases the dynamic nature of Python variables and how they can be easily reassigned and manipulated within a program.

## Many Values to Multiple Variables

Python allows you to assign values to multiple variables in one line:

```
a, b, c = "Orange", "Banana", "Cherry"
```

```
print(a) print(b) print(c)
```

## One Value to Multiple Variables

You can assign the same value to multiple variables in one line:

```
a = b = c = "Orange"
```

```
print(a) print(b) print(c)
```

Point	Code
String Variable	<code>name: str = "John"</code> <code>name = "John"</code>
Integer Variable	<code>age: int = 25</code> <code>age = 25</code>
Float Variable	<code>height: float = 1.75</code> <code>height = 1.75</code>
Boolean Variable	<code>is_student: bool = True</code> <code>is_student = True</code>
Many String Variables	<code>a: str = "Orange", b: str = "Banana", c: str = "Cherry"</code> <code>a, b, c = "Orange", "Banana", "Cherry"</code>
One String Variable	<code>a: str = "Orange", b: str = "Orange", c: str = "Orange"</code> <code>a = b = c = "Orange"</code>



IBM ICE (Innovation Centre for Education)

## Python reserved words/keywords

- In Python, reserved words are words that have a special meaning and cannot be used as identifiers (variable names, function names, etc.) Because they are reserved for specific purposes in the language.
- `Async`: It is used to create asynchronous coroutine.
- `Await`: It releases the flow of control back to the event loop.

---

Figure 1-7. Python reserved words/keywords

These reserved words are an integral part of the Python language syntax and are used to define the structure and behavior of programs. Attempting to use any of these words as identifiers will result in a syntax error. There are 35 keywords in Python version 3.7 and above. The earlier version of Python 2 has 30 keywords. One major change between the versions is the removal of the "print" keyword from version 2 and making it a function call in version 3. Since these keywords have special meaning to the interpreter, they cannot be used as a normal variable inside Python.

Rules for keywords in python:

- Python keywords could not be used as identifiers.
- All the keywords in Python should be in lowercase except True and False.

Here is a list of Python-reserved words:

Keyword	Description
<code>and</code>	A logical operator
<code>as</code>	To create an alias
<code>assert</code>	For debugging
<code>break</code>	To break out of a loop
<code>class</code>	To define a class
<code>continue</code>	To continue to the next iteration of a loop
<code>def</code>	To define a function
<code>del</code>	To delete an object
<code>elif</code>	Used in conditional statements, same as else if
<code>else</code>	Used in conditional statements
<code>except</code>	Used with exceptions, what to do when an exception occurs
<code>False</code>	Boolean value, result of comparison operations
<code>finally</code>	Used with exceptions, a block of code that will be executed no matter if there is an exception or not
<code>for</code>	To create a for loop
<code>from</code>	To import specific parts of a module
<code>global</code>	To declare a global variable
<code>if</code>	To make a conditional statement
<code>import</code>	To import a module
<code>in</code>	To check if a value is present in a list, tuple, etc.
<code>is</code>	To test if two variables are equal
<code>lambda</code>	To create an anonymous function
<code>None</code>	Represents a null value

<u>for</u>	To create a for loop
<u>from</u>	To import specific parts of a module
<u>global</u>	To declare a global variable
<u>if</u>	To make a conditional statement
<u>import</u>	To import a module
<u>in</u>	To check if a value is present in a list, tuple, etc.
<u>is</u>	To test if two variables are equal
<u>lambda</u>	To create an anonymous function
<u>None</u>	Represents a null value
<u>nonlocal</u>	To declare a non-local variable
<u>not</u>	A logical operator
<u>or</u>	A logical operator
<u>pass</u>	A null statement, a statement that will do nothing
<u>raise</u>	To raise an exception
<u>return</u>	To exit a function and return a value
<u>True</u>	Boolean value, result of comparison operations
<u>try</u>	To make a try...except statement
<u>while</u>	To create a while loop
<u>with</u>	Used to simplify exception handling
<u>yield</u>	To return a list of values from a generator



# Python data types

- A variety of data types is supported by python, encompassing different kinds of data within programs.
- In this section, a brief overview of these data types will be provided, as depicted in figure. It should be noted that more complex data types, including dictionaries, sequence types, sets, etc., Will be explored in greater detail later in this book.

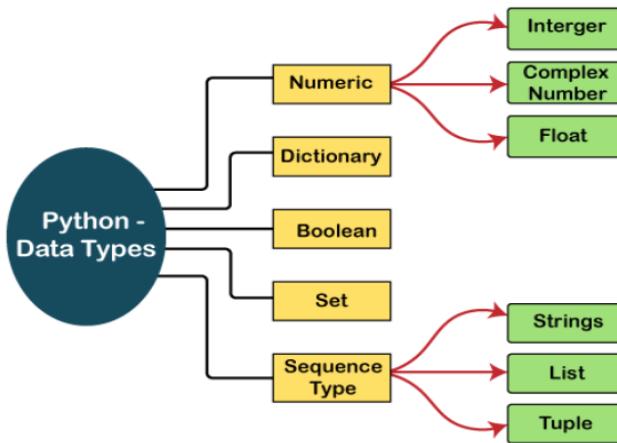


Figure 3: Python data types

Source: <https://www.Javatpoint.Com/python-data-types>

Figure 1-8. Python data types

## Dynamic typing in python

Dynamic typing is a feature in python that allows variables to change their data type during runtime. This means the type of a variable doesn't need to be explicitly declared when defined; the python interpreter determines the type based on the assigned value. The data type is inferred by the python interpreter based on the assigned value. This flexibility makes python a dynamically typed language. Key points about dynamic typing in python:

- No explicit type declaration: Unlike statically typed languages (e.g., C++ or java), the data type of a variable doesn't need to be declared when created.

```

x = 5          # x is an integer
x = "Hello"    # x is now a string
  
```

Variables can change type: The type of a variable can change during execution by assigning a new value of a different type to the variable.

```
age = 25      # age is an integer
age = "young" # age is now a string
```

Type checking at runtime: Type errors are detected during runtime rather than at compile-time.

```
x = 10
y = "20"
z = x + y # This will result in a TypeError at runtime
```

### Numeric data type

- int: Integer data type represents whole numbers (e.g., 5, -10, 100).
- float: Float data type represents floating-point or decimal numbers (e.g., 3.14, -0.5, 2.0).
- Complex: In Python, the complex data type is used to represent complex numbers. A complex number is a number that comprises a real part and an imaginary part. In Python, the imaginary part is denoted by a suffix j or J. The general form of a complex number is a + bj, where a is the real part, b is the imaginary part, and j is the imaginary unit. In the below example, complex\_num1 and complex\_num2 are two complex numbers. The program performs the addition and multiplication of these complex numbers and then prints the results. The output will show the real and imaginary parts of the complex numbers and the results of the operations.

```
# Complex numbers
complex_num1 = 2 + 3j
complex_num2 = 1 - 2j

# Addition of complex numbers
sum_complex = complex_num1 + complex_num2

# Multiplication of complex numbers
product_complex = complex_num1 * complex_num2

# Displaying the results
print(f"Complex Number 1: {complex_num1}")
print(f"Complex Number 2: {complex_num2}")
print(f"Sum of Complex Numbers: {sum_complex}")
print(f"Product of Complex Numbers: {product_complex}")
```

### Dictionary data type

In Python, a dictionary is a built-in data type that represents an unordered and mutable collection of key-value pairs. Dictionaries are also sometimes referred to as associative arrays or hash maps in other programming languages. Each key in a dictionary must be unique and it is associated with a specific value. Dictionaries are defined by enclosing key-value pairs in curly braces {} or by using the dict() constructor.

#### Creating dictionaries

A dictionary can be created by enclosing key-value pairs in curly braces {}.

```
my_dict = {'name': 'Alice', 'age': 25, 'city': 'Wonderland'}
```

Alternatively, the dict() constructor can be utilized.

```
another_dict = dict(name='Bob', age=30, city='Dreamland')
```

### Features of dictionaries

**Unordered:** Dictionaries do not maintain the order of key-value pairs as they are inserted.

```
my_dict = {'name': 'Alice', 'age': 25, 'city': 'Wonderland'}
```

**Mutable:** The values associated with keys can be modified, new key-value pairs can be added, or existing ones can be removed.

```
my_dict['age'] = 26 # Update the value associated with the 'age' key
```

```
my_dict['gender'] = 'female' # Add a new key-value pair
```

```
del my_dict['city'] # Remove the 'city' key and its associated value
```

**Keys are unique:** Uniqueness is required for each key in a dictionary. If an attempt is made to add a duplicate key, the existing value will be overwritten.

```
my_dict = {'name': 'Alice', 'age': 25, 'name': 'Bob'}
```

# After this, my\_dict will be {'name': 'Bob', 'age': 25}

### Common operations

```
name_value = my_dict['name'] # Access the value associated with the 'name' key
```

```
keys_list = my_dict.keys() # Get a list of all keys
```

```
values_list = my_dict.values() # Get a list of all values
```

```
items_list = my_dict.items() # Get a list of all key-value pairs (as tuples)
```

### Dictionary examples

**dictionary for student information:** In this example, a dictionary is used to store information about a student, including their name, age, and grades in different subjects.

```
#dictionary for student information
student_info = {
    'name': 'Alice',
    'age': 20,
    'grades': {'math': 90, 'English': 85, 'history': 92}
}
```

# Accessing values

```
print(f"Student Name: {student_info['name']}")  
print(f"Math Grade: {student_info['grades']['math']}")
```

**Word frequency counter:** Here, a dictionary is used to count the frequency of each word in a text.

```
# Word frequency counter using a dictionary
text = "python is powerful and python is easy"
word_frequency = {}
for a word in text.split():
    word_frequency[word] = word_frequency.get(word, 0) + 1
print("Word Frequency:")
for word, count in word_frequency.items():
    print(f"{word}: {count}")
```

**Configuring a system:** In this example, a dictionary is used to store and access configuration settings for a system.

```
# Configuration settings using a dictionary
config_settings = {
    'server': 'localhost',
    'port': 8080,
    'debug_mode': True,
    'max_connections': 100
}
# Accessing and modifying configuration settings
print(f"Server: {config_settings['server']}")
config_settings['debug_mode'] = False
```

### Boolean data type

Boolean type: Represents boolean values, true or false.

In the below program, we have variables representing the int data type (age), the float data type (height), the boolean data type (is\_student), and the complex data type (complex\_number). The type() function is used to print the type of each variable.

```
age = 25
height = 5.9
is_student = True
complex_number = 3 + 4j

print(f"Age: {age}, Type: {type(age)}")
print(f"Height: {height}, Type: {type(height)}")
print(f"Is Student: {is_student}, Type: {type(is_student)}")
print(f"Complex Number: {complex_number}, Type: {type(complex_number)}")
```

### Set data type

In Python, a set is a built-in data type that represents an unordered and mutable collection of unique elements. Sets are defined by enclosing elements in curly braces {} or by using the set() constructor. Sets do not allow duplicate elements, and they do not maintain the order of elements.

Here's an overview of the set data type:

#### Creating sets

A set can be created by enclosing a sequence of elements in curly braces {}.

```
my_set = {1, 2, 3, 'apple', 'banana'}
```

#### Features of sets

Lists are a fundamental and versatile data type in Python. They are commonly used to store collections of items. Here are some key features and operations associated with lists in Python:

- **Unordered:** Sets do not maintain the order of elements as they are inserted.
- ```
my_set = {1, 2, 3, 'apple', 'banana'}
```
- **Mutable:** The elements of a set can be modified by adding or removing elements. 

```
my_set.add('orange')
```

  
# Adds an element to the set
- ```
my_set.remove(2) # Removes the element 2 from the set
```

- **No duplicates:** Duplicate elements are not allowed in sets. If an attempt is made to add an element that already exists, it will not be added again.

```
• my_set = {1, 2, 3, 'apple', 'banana', 2, 'apple'}
# After this, my_set will be {1, 2, 3, 'apple', 'banana'}
```

### Common operations:

```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}
union_set = set1.union(set2) # Union of two sets
intersection_set = set1.intersection(set2) # Intersection of two sets
difference_set = set1.difference(set2) # Set difference (elements in set1 but not in set2)
```

### Membership Test:

```
element_to_check = 'apple'
if element_to_check in my_set:
    print(f"{element_to_check} is present in the set.")
else:
    print(f"{element_to_check} is not present in the set.")
```

### Sets examples

**Removing duplicates from a list:** In this example, using a set helps remove duplicate elements from the original list; sets automatically discard duplicates.

```
my_list = [1, 2, 3, 2, 4, 5, 6, 4]
unique_numbers = set(my_list)
print(list(unique_numbers))
```

**Checking common elements between two lists:** Here, a set is used to find the common elements between two lists by leveraging the intersection method.

```
list1 = [1, 2, 3, 4, 5]
list2 = [3, 4, 5, 6, 7]
common_elements = set(list1).intersection(list2)
print(list(common_elements))
```

**Fast membership testing:** Sets are particularly efficient for membership tests. Checking if an element is in a set has an average time complexity of O(1).

```
my_set = {1, 2, 3, 4, 5}
element_to_check = 3
if element_to_check in my_set:
    print(f"{element_to_check} is present in the set.")
else:
    print(f"{element_to_check} is not present in the set.")
```

**Finding unique elements in a text:** Sets can be used to find unique elements (words in this case) in a text by splitting the text into words.

```
text = "python is powerful and python is easy"
unique_words = set(text.split())
print(unique_words)
```

## Sequence data type

Sequences represent an ordered collections of items. It consists of:

- string: Represent sequences of characters, often used for text and human-readable data. It represents text data enclosed in single or double quotes, such as "Hello, world!" or 'Python is awesome!'.
- list: Represents a mutable list of items, which can be modified after creation.
- tuple: Represents an immutable list of items that cannot be modified after creation.

## String type

In python, a string is a data type that represents a sequence of characters. Strings are used to store and manipulate text or any sequence of characters, and they play a significant role in python programming. Here are key aspects of the string data type and their significance in python:

**Definition:** A string in python is created by enclosing a sequence of characters within single (' '), double (" "), or triple (""" """) quotes. Examples:

```
single_quoted_string = 'Hello, World!'
double_quoted_string = "Python Programming"
triple_quoted_string = """ This is a multi-line
string in Python""
```

**Immutability:** Strings in python are immutable, meaning their values cannot be changed after they are created. Any operation that appears to modify a string actually creates a new string.

Common string operations:

- Concatenation: Combining two or more strings using the + operator.
- Repetition: Repeating a string multiple times using the \* operator.
- Indexing: Accessing individual characters using index positions.
- Slicing: Extracting a portion of a string using the slice notation ([start: stop]).
- Length: Determining the length of a string using the len() function.

**String methods:** String methods in python are built-in functions that operate on strings and provide a wide range of operations for string manipulation. These methods allow common tasks to be performed, such as modifying the case of characters, finding substrings, replacing text, and more. Here are some frequently used string methods in python:

- upper() and lower(): Upper() converts all characters in a string to uppercase, and lower() converts all characters in a string to lowercase.

```
text = "Hello, World!"
print(text.upper()) # Outputs: HELLO, WORLD!
print(text.lower()) # Outputs: hello, world!
• capitalize(): Converts the first character of a string to uppercase and the rest to lowercase.
text = "python programming"
print(text. capitalize()) # Outputs: Python programming
• title(): Converts the first character of each word to uppercase.
text = "python programming is fun"
print(text.title()) # Outputs: Python Programming Is Fun
• strip(), lstrip(), and rstrip(): Strip() removes leading and trailing whitespaces, lstrip() removes leading whitespaces, rstrip() removes trailing whitespaces.
```

```

text = "    Python Programming    "
print(text.strip())    # Outputs: Python Programming
    • replace(): Replaces occurrences of a specified substring with another
      substring.

text = "Hello, World!"
new_text = text.replace("World", "Python")
print(new_text)    # Outputs: Hello, Python!
    • find() and index(): Find() returns the index of the first occurrence of a substring (-1 if not found), and
      index() returns the index of the first occurrence of a substring (raises an exception if not found).

text = "Python is powerful and Python is easy to learn"
print(text.find("Python"))    # Outputs: 0
print(text.index("powerful"))    # Outputs: 11
    • count(): Returns the number of occurrences of a specified substring in the string.

text = "Python is powerful and Python is easy to learn"
print(text.count("Python"))    # Outputs: 2
    • startswith() and endswith(): Startswith() checks if a string starts with a specified prefix, endswith() checks
      if a string ends with a specified suffix.

text = "Python is powerful and Python is easy to learn"
print(text.startswith("Python"))    # Outputs: True
print(text.endswith("learn"))    # Outputs: True
    • Creating Strings: Strings in Python are created by enclosing characters in single ('') or double ("") quotes.

single_quoted_string = 'Hello, World!'
double_quoted_string = "Python Programming"
    • String Concatenation: Strings can be concatenated using the + operator.

greeting = "Hello"
name = "Alice"
full_greeting = greeting + " " + name # Results in "Hello Alice"
    • String Indexing and Slicing: Strings in Python are zero-indexed, meaning the first character is at index 0,
      the second at index 1, and so on. Indexing is used to access individual characters while slicing is used to
      extract substrings.

text = "Python"
first_char = text[0] # Results in 'P'
substring = text[1:4] # Results in 'yth'
    • String Formatting: Strings can be formatted using f-strings (formatted string literals) or the format()
      method.

name = "Alice"
age = 30
formatted_string = f"My name is {name} and I am {age} years old."
    • Escape characters: Escape characters are used to include special characters within a string. Common
      escape characters include \n for a newline and \" for a double quote.

special_string = "This is a line\nwith a newline character."

```

- Triple-quoted strings (enclosed by "" or "") are used for multi-line strings, allowing the inclusion of line breaks without using escape characters.

```
multiline_string = '''This is a multi-line string in Python.'''
    •Membership Testing (in): In operator can be used to check if a substring exists within a string.
```

```
text = "Python Programming"
```

```
if "Python" in text:
```

```
    print("Substring found!")
```

- Repetition (\*): The \* operator can be used to repeat a string multiple times.

```
repeated_text = "Hello, " * 3 # Results in "Hello, Hello, Hello, "
```

- Comparison: Strings can be compared using standard comparison operators (==, !=, <, >, <=, >=).

```
string1 = "apple"
```

```
string2 = "banana"
```

```
result = string1 < string2 # Results in True (lexicographical order)
```

- Unicode and Encoding: Strings in Python 3 are Unicode by default. This means that characters from various languages and symbol sets can be represented.

```
unicode_string = "This is a Unicode string: ???"
```

- Encoding and Decoding: Methods are provided by Python to encode and decode strings using different encoding schemes. The encode() method converts a string into bytes and the decode() method converts bytes back to a string.

```
utf8_encoded = unicode_string.encode('utf-8')
```

```
decoded_string = utf8_encoded.decode('utf-8')
```

Example code-

```
# String Methods
length = len(message)
uppercase = message.upper()
lowercase = message.lower()
stripped = message.strip()
replaced = message.replace("Py", "Jy")
index_of_t = message.find("t")
count_of_o = message.count("o")

# String Formatting
age = 25
formatted_string = f"My name is {name} and I am {age} years old."
alternative_format = "My name is {} and I am {} years old.".format(name, age)

# Escape Characters
escaped_string = "This is a line.\nThis is a new line."
```

**The output of the above code snippet is:**

single\_quoted\_string: Hello, World!

double\_quoted\_string: Python Programming

```

multiline_string: This is a
multi-line string.
full_greeting: Hello, Alice
repeated_string: Python Python Python
first_char: P
substring: the
length: 6
uppercase: PYTHON
lowercase: Python
stripped: Python
replaced: Jython
index_of_t: 2
count_of_o: 2
formatted_string: My name is Alice, and I am 25 years old.
alternative_format: My name is Alice, and I am 25 years old.
escaped_string: This is a line.

This is a new line.

```

These are just a few examples of the many string methods available in Python. String methods provide a powerful and convenient way to manipulate and process textual data in Python programs. In summary, the string data type is a fundamental and versatile component of Python, providing the means to work with textual data efficiently. Its immutability ensures data integrity, and the rich set of operations and methods makes string manipulation convenient for various programming tasks.

## Lists

In Python, a list is a built-in data type that represents an ordered and mutable collection of elements. Lists are one of the most versatile and widely used data structures in Python. Here's an overview of the list data type:

### Creating lists

A list is created by enclosing a sequence of elements in square brackets []

```
my_list = [1, 2, 3, 'apple', 'banana']
```

### Features of lists

Lists are a fundamental and versatile data type in Python. They are commonly used to store collections of items. Here are some key features and operations associated with lists in Python:

**Ordered:** Lists maintain the order of elements as they are inserted.

```
my_list = [1, 'apple', 3.14, True]
```

**Mutable:** The elements of a list can be modified by adding new elements or removing existing ones.

```
my_list[1] = 'banana'
```

```
my_list.append('orange')
```

```
del my_list[0]
```

**Indexing and slicing:** Elements in a list can be accessed using index notation; list slicing is used to extract sublists.

```
first_element = my_list[0]
```

```
sublist = my_list[1:4]
```

**Dynamic:** Lists in Python can grow or shrink in size dynamically.

```
my_list.append('grape') # Add an element
```

```
my_list.pop() # Remove the last element
```

### **Common operations (append, extend, insert, delete, length, count, sort, reverse)**

```
my_list.append(4) # Adds an element to the end
```

```
my_list.extend([5, 6]) # Extends the list with another iterable
```

```
my_list.insert(2, 'orange') # Inserts 'orange' at index 2
```

```
del my_list[1] # Deletes the element at index 1
```

```
length = len(my_list) # Returns the number of elements
```

```
count_apple = my_list.count('apple') # Returns the number of occurrences of 'apple'
```

```
my_list.sort() # Sorts the list in ascending order
```

```
my_list.reverse() # Reverses the order of elements
```

### **Example code**

In a use case where a list of numbers is available, various operations, including finding the sum, average, maximum, and minimum values, are desired to be performed on this list.

```
# List of numbers
numbers = [10, 5, 8, 12, 7, 3, 15, 20]

# Find the sum of the numbers
sum_of_numbers = sum(numbers)
print(f"Sum of numbers: {sum_of_numbers}")

# Find the average of the numbers
average_of_numbers = sum_of_numbers / len(numbers)
print(f"Average of numbers: {average_of_numbers:.2f}")

# Find the maximum and minimum values
max_value = max(numbers)
min_value = min(numbers)

print(f"Maximum value: {max_value}")
print(f"Minimum value: {min_value}")

# Sort the list in ascending order
sorted_numbers = sorted(numbers)
print(f"Sorted numbers: {sorted_numbers}")

# Check if a specific value exists in the list
value_to_check = 8
if value_to_check in numbers:
    print(f"{value_to_check} is present in the list.")
else:
    print(f"{value_to_check} is not present in the list.")
```

### **Tuples**

In Python, a tuple is a built-in data type that represents an ordered and immutable collection of elements. Tuples are similar to lists, but the key difference is that once a tuple is created, its elements cannot be changed or modified. Tuples are defined using parentheses () .

## **Creating tuple**

A tuple can be created by enclosing a sequence of elements in parentheses ().

```
my_tuple = (1, 2, 'apple', 'banana')
```

## **Accessing elements**

Access to elements in tuples is achieved through indexing, similar to lists. Indexing starts from 0, and elements from the end of the tuple can also be accessed using negative indices.

```
print(numbers_tuple[0]) # Output: 1
print(numbers_tuple[2]) # Output: 3
print(numbers_tuple[-1]) # Output: 5
```

## **Slicing**

Tuples can be sliced using the colon: Operator. Slicing creates a new tuple containing a subset of the original tuple's elements.

```
print(numbers_tuple[1:3]) # Output: (2, 3)
print(numbers_tuple[:3]) # Output: (1, 2, 3)
print(numbers_tuple[-3:]) # Output: (3, 4, 5)
```

## **Features of tuple**

Lists are a fundamental and versatile data type in Python. They are commonly used to store collections of items. Here are some key features and operations associated with lists in Python:

- **Ordered:** Tuple maintains the order of elements as they are inserted.

```
my_list = (1, 'apple', 3.14, True)
```

- **Immutable:** Once a tuple is created, its elements cannot be changed. Elements in a tuple cannot be added, removed, or modified.

```
my_tuple[1] = 'orange' # This will raise an error del my_list[0]
```

- **Indexing and slicing:** Elements in a tuple can be accessed using index notation, and sublists can also be extracted using slicing.

```
first_element = my_tuple[0]
```

```
subtuple = my_tuple[1:3]
```

- **Dynamic:** Tuples can grow or shrink in size dynamically.

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

```
my_tuple = my_tuple + (4, 5) # Creates a new tuple
```

## **Common operations**

```
length = len(my_tuple) # Returns the number of elements
```

```
count_apple = my_tuple.count('apple') # Returns the number of occurrences of 'apple'
```

```
fruits = ('apple', 'orange', 'banana')
```

```
# Iterating through the tuple
```

```
for fruit in fruits:
```

```
    print(fruit)
```

```
# Accessing elements
```

```
first_fruit = fruits[0]
```

```
# Length of the tuple
```

```
tuple_length = len(fruits)
```

```
# Count occurrences of an element
count_apple = fruits.count('apple')
# Displaying the results
print(f"First fruit: {first_fruit}")
print(f"Tuple length: {tuple_length}")
print(f"Count of 'apple': {count_apple}")
```

**Output:**

```
apple
orange
banana
First fruit: apple
Tuple length: 3
Count of 'apple': 1
```



IBM ICE (Innovation Centre for Education)

# Operators and expressions

- In python, operators and expressions are fundamental concepts that play a crucial role in programming.
- Operators: operators in python are symbols that perform operations on operands. Operands can be variables, values, or expressions.
- Python supports a variety of operators, which can be categorized into different types:
  - Arithmetic operators.
  - Assignment operators.
  - Comparison operators.
  - Logical operators.
  - Identity operators.
  - Membership operators.
  - Bitwise operators.

---

Figure 1-9. Operators and expressions

## Arithmetic operators

Arithmetic operators in Python are symbols used to perform mathematical operations on numeric values. These arithmetic operators are fundamental for performing mathematical calculations in python. They can be used with variables, literals, or expressions to manipulate numeric data. Understanding how to use these operators is essential for various applications, from simple calculations to more complex mathematical operations in programming. Here are the main arithmetic operators in Python.

```
# Sample values
num1 = 5
num2 = 3

# Arithmetic operations and results
sum_result = num1 + num2 # Sum: 8.0 (Example values: 5, 3)
difference_result = num1 - num2 # Difference: 2.0
product_result = num1 * num2 # Product: 15.0
quotient_result = num1 / num2 # Quotient: 1.6666666666666667
floor_division_result = num1 // num2 # Floor Division: 1
remainder_result = num1 % num2 # Remainder: 2
exponentiation_result = num1 ** num2 # Exponentiation: 125
```

Note:

- The floor division operator (//) is used to perform division and discard the fractional part. It results in an integer value.
- The division operator (/) is used to divide the left operand by the right operand. It performs floating-point division, even if the operands are integers.
- The exponentiation operator (\*\*) is used to raise the left operand to the power of the right operand. It calculates exponentiation.

### Comparison operators

Python comparison operators are used to compare values and return a boolean result (true or false). The below example demonstrates the use of comparison operators in python. Each operator compares the values of the variables x and y and returns a boolean result based on the specified condition.

## Logical operators

```

# Equal to (==) operator
x = 5
y = 10
result_equal = x == y
print(result_equal) # Output: False

# Not equal to (!=) operator
result_not_equal = x != y
print(result_not_equal) # Output: True

# Greater than (>) operator
result_greater_than = x > y
print(result_greater_than) # Output: False

# Less than (<) operator
result_less_than = x < y
print(result_less_than) # Output: True

# Greater than or equal to (>=) operator
result_greater_equal = x >= y
print(result_greater_equal) # Output: False

# Less than or equal to (<=) operator
result_less_equal = x <= y
print(result_less_equal) # Output: True

```

Python has three logical operators: and, or, and not. These operators are used to perform logical operations on boolean values.

```

# Get user input for age and student status
age = int(input("Enter your age: "))
is_student = input("Are you a student? (yes/no): ").lower() == 'yes'

# Check eligibility for a discount
if age >= 18 and not is_student:
    print("You qualify for a regular discount.")
elif age < 18 or is_student:
    print("You qualify for a special discount.")
else:
    print("You are not eligible for any discount.")

```

In the above example,

- We use an operator to check if the person is 18 years or older and not a student to qualify for a regular discount. We use or operator to check if the person is either under 18 years old or a student to qualify for a special discount.
- The not operator is used to negate the boolean value of is\_student.

This practical example demonstrates how logical operators can be used to make decisions based on multiple conditions in a real-world scenario, such as determining eligibility for discounts based on age and student status.

## Identity operators

In python, identity operators (is and is not) are useful for comparing the memory locations of two objects. They check if two variables or objects refer to the same object in memory rather than comparing their values. Here's why identity operators are useful:

```

# Identity operator 'is'
a = [1, 2, 3]
b = a
result_is = a is b
print("Result for 'a is b':", result_is)

# Identity operator 'is not'
c = [1, 2, 3]
result_is_not = a is not c
print("Result for 'a is not c':", result_is_not)

# Example with integers
x = 10
y = 10
result_is_int = x is y
print("Result for 'x is y':", result_is_int)

# Example with strings
str1 = "hello"
str2 = "hello"
result_is_str = str1 is str2
print("Result for 'str1 is str2':", result_is_str)

```

- In the first example, a is b returns true because both variables a and b reference the same list object in memory.
- In the second example, a is not c returns true because a and c reference different list-objects.
- The last two examples show that small integers and strings are often cached and reused in python so that it may return true for them, but this behavior is not guaranteed for larger or dynamically created objects. Generally, it's recommended to use the == operator for comparing values and reserve the use of is for comparing object identity.

## Membership operators

Python has two membership operators, in and not in, which are used to test whether a value exists in a sequence (such as a string, list, or tuple). The below example demonstrates how the in and not-in operators work with different data types. They are commonly used to check for the existence of an element in a list, a character in a string, or a key in a dictionary.

```

# Membership operator 'in'
my_list = [1, 2, 3, 4, 5]
element_to_check = 3
result_in = element_to_check in my_list
print(f"{element_to_check} is in {my_list}: {result_in}") # Output: True

# Membership operator 'not in'
element_to_check = 6
result_not_in = element_to_check not in my_list
print(f"{element_to_check} is not in {my_list}: {result_not_in}") # Output: True

# Membership operator with strings
my_string = "Hello, World!"
substring_to_check = "World"
result_in_string = substring_to_check in my_string
print(f"'{substring_to_check}' is in '{my_string}': {result_in_string}') # Output: Tr

# Membership operator with dictionaries (checks keys)
my_dict = {'a': 1, 'b': 2, 'c': 3}
key_to_check = 'b'
result_in_dict = key_to_check in my_dict
print(f"[key_to_check] is a key in {my_dict}: {result_in_dict}") # Output: True

```

## Bitwise operators

Several bitwise operators are provided by Python, allowing manipulation of individual bits in integers. Bitwise operators are typically utilized in low-level programming, such as system-level programming, device drivers, embedded systems, and situations requiring direct manipulation of binary data. Here are some common scenarios where bitwise operators are useful: here are the main bitwise operators in Python:

- **Bitwise AND (&):** Sets each bit to 1 if both bits are 1.

```

a = 5 # 0b0101 in binary
b = 3 # 0b0011 in binary
result_and = a & b
print(result_and) # Output: 1 (0b0001 in binary)

```

- **Bitwise or (|):** Sets each bit to 1 if at least one of the corresponding bits is 1.

```

a = 5 # 0b0101 in binary
b = 3 # 0b0011 in binary
result_or = a | b
print(result_or) # Output: 7 (0b0111 in binary)

```

- **Bitwise xor (^):** Sets each bit to 1 if only one of the corresponding bits is 1.

```
a = 5 # 0b0101 in binary
b = 3 # 0b0011 in binary
result_xor = a ^ b
print(result_xor) # Output: 6 (0b0110 in binary)
```

- **Bitwise not (~):** Inverts the bits, changing 1s to 0s and vice versa.

```
a = 5 # 0b0101 in binary
result_not = ~a
print(result_not) # Output: -6 (the result is in two's complement form)
```

- **Left shift (<<):** Shifts the bits of the left operand to the left by the number of positions specified by the right operand.

```
a = 5 # 0b0101 in binary
shift_left = a << 1
print(shift_left) # Output: 10 (0b1010 in binary)
```

- **Right shift (>>):** Shifts the bits of the left operand to the right by the number of positions specified by the right operand.

```
a = 5 # 0b0101 in binary
shift_right = a >> 1
print(shift_right) # Output: 2 (0b0010 in binary)
```

```
# Define flags
READ = 1
WRITE = 2
EXECUTE = 4

# Set permissions using bitwise OR
permissions = READ | WRITE

# Check if READ permission is set using bitwise AND
can_read = permissions & READ != 0

# Check if EXECUTE permission is set using bitwise AND
can_execute = permissions & EXECUTE != 0

print("Can read:", can_read)      # Output: True
print("Can execute:", can_execute) # Output: False
```

## Operator precedence

Operator precedence in Python refers to the order in which different operators are evaluated when expressions are parsed. Operators with higher precedence are evaluated first. If operators have the same precedence, the associativity of the operators determines the order of evaluation (figure).

Operators	Associativity
() Highest precedence	Left - Right
**	Right - Left
+x, -x, ~x	Left - Right
*, /, //, %	Left - Right
+, -	Left - Right
<<, >>	Left - Right
&	Left - Right
^	Left - Right
	Left - Right
Is, is not, in, not in, <, <=, >, >=, ==, !=	Left - Right
Not x	Left - Right
And	Left - Right
Or	Left - Right
If else	Left - Right
Lambda	Left - Right
=, +=, -=, *=, /= Lowest Precedence	Right - Left

Source:

([https://medium.Com/@rishu\\_\\_2701>this-article-will-teach-you-about-operator-precedence-and-associativity-in-python-ee455c7fbfee](https://medium.com/@rishu__2701>this-article-will-teach-you-about-operator-precedence-and-associativity-in-python-ee455c7fbfee))

Example:

Result = 2 + 3 \* 4 > 5 \*\* 2

Here's the breakdown of how the expression is evaluated based on operator precedence:

- Exponentiation (\*\*): 5 \*\* 2 is evaluated first, resulting in 25.
- Multiplication (\*): 3 \* 4 is evaluated next, resulting in 12.
- Addition (+): 2 + 12 is evaluated last, resulting in 14.
- Comparison (>): finally, the comparison 14 > 25 is evaluated, resulting in false.

So, the output of the print(result) statement is false.

The example highlights how operator precedence determines the order in which operations are performed. Without parentheses to override the default precedence, the multiplication and exponentiation operations take precedence over addition, and then the comparison is performed. Understanding this order of evaluation is crucial for writing correct and predictable expressions in python.



IBM ICE (Innovation Centre for Education)

# Input and output operations

- Input and output operations in python refer to the ways in which a program interacts with the external world, such as receiving input from users and producing output for display or storage.
- Here's an overview of input and output operations in python:
  - Print function in python: The `print()` function in python is utilized to display information to the console. The output format can be customized using the `end` and `sep` parameters in the `print()` function.
  - Input function in python: The `input()` function in python is used to take user input from the console. It allows the user to enter data during the execution of a program.

---

Figure 1-10. Input and output operations

## Print function in python

- End parameter: The `end` parameter specifies what to print at the end of the `print` statement. By default, it is set to a newline character ('`\n`'), which means each `print` statement outputs on a new line.
- Sep parameter: The `sep` parameter specifies the separator between multiple items passed to the `print` function. By default, it is set to a space ('`'`).

```

# Example 1: Printing a simple string
print("Hello, World!")

# Example 2: Printing variables
name = "Alice"
age = 30
print("Name:", name, "Age:", age)

# Example 3: Formatting with f-strings (Python 3.6 and above)
print(f"My name is {name} and I am {age} years old.")

# Example 4: Printing multiple items with commas
print("One", "Two", "Three")

# Example 5: Printing with end parameter
print("This will be printed", end=" ")
print("on the same line.")

# Example 6: Printing with sep parameter
print("Separate", "words", sep="-")

# Example 7: Printing numbers and arithmetic
x = 5
y = 10
print("Sum of", x, "and", y, "is", x + y)

```

## Input function in python

Example:

- The user is prompted to enter a name and three scores separated by space.
- The split() method is used to split the input string into a list of values.
- The name is extracted from the first element of the list (input\_list[0]), and the scores are extracted from the remaining elements (input\_list[1:]).

The scores are converted to integers using list comprehension.

The program then prints the entered name and scores.

```
# Example: Taking multiple inputs for name and scores
user_input = input("Enter name and three scores separated by space: ")

# Split the input string into a list of values
input_list = user_input.split()

# Extracting name and scores
name = input_list[0]
scores = [int(score) for score in input_list[1:]]

# Display the result
print("Name:", name)
print("Scores:", scores)
```

```
Enter name and three scores separated by space: Alice 85 92 78
Name: Alice
Scores: [85, 92, 78]
```

The chapter introduces Python, exploring its popularity and use cases. Python syntax and structure are covered, including indentation and block structure.

Writing and executing Python programs are explained, involving Python interactive mode (REPL) and running scripts from the command line. The Python basics section delves into identifiers, variables, reserved words, and data types such as dynamic typing, numeric types, boolean types, and strings. The chapter on operators and expressions covers arithmetic, comparison, logic, identity, membership, and bitwise operators, as well as operator precedence. The section on input and output operations explores the print function and input function in Python.



IBM ICE (Innovation Centre for Education)

## Self evaluation: Exercise 1

- **Exercise 1:** Hello, World! program
- **Estimated time:** 00:10 minutes
- **Aim:** Write a python program that prints "Hello, World!" to the console.
- **Learning objective:**
  - The learner will understand the basic structure of a python program and the usage of the print statement

---

Figure 1-11. Self evaluation: Exercise 1

Self evaluation exercise 1 is as stated above.



IBM ICE (Innovation Centre for Education)

## Self evaluation: Exercise 2

- **Exercise 2:** Interactive mode basics
- **Estimated time:** 00:10 minutes
- **Aim:** Use python's interactive mode to perform basic arithmetic operations like addition, subtraction, multiplication, and division.
- **Learning objective:**
  - The learner will understand how to use python's interactive mode for basic arithmetic operations

---

Figure 1-12. Self evaluation: Exercise 2

Self evaluation exercise 2 is stated above.

## Self evaluation: Exercise 3



IBM ICE (Innovation Centre for Education)

- **Exercise 3:** User input and display
- **Estimated time:** 00:15 minutes
- **Aim:** Create a python script that takes user input for their name and displays a personalized greeting.
- **Learning objective:**
  - The learner will understand how to use python to take user input and display output

---

Figure 1-13. Self evaluation: Exercise 3

Self evaluation exercise 3 is as stated above.

## Self evaluation: Exercise 4



IBM ICE (Innovation Centre for Education)

- **Exercise 4:** Calculate rectangle and circle area
- **Estimated time:** 00:10 minutes
- **Aim:** Write a python program that calculates and prints the area of a rectangle and circle. Prompt the user for the length and width and radius respectivley.
- **Learning objective:**
  - The learner will understand how to write a python program to calculate the area of a rectangle and circle

---

Figure 1-14. Self evaluation: Exercise 4

Self evaluation exercise 4 is as stated above.

## Self evaluation: Exercise 5



IBM ICE (Innovation Centre for Education)

- **Exercise 5:** Temperature conversion
- **Estimated time:** 00:10 minutes
- **Aim:** Write a python program that converts a temperature from fahrenheit to celsius. Prompt the user for the temperature in fahrenheit and display the result in celsius upto 2 decimals.
- **Learning objective:**
  - The learner will understand how to write a python program for temperature conversion

---

Figure 1-15. Self evaluation: Exercise 5

Self evaluation exercise 5 is as stated above.

## Checkpoint (1 of 2)



IBM ICE (Innovation Centre for Education)

### Multiple-choice questions:

1. Which of the following is/are true regarding Python IDLE?
  - a) It is an integrated development environment for Python.
  - b) It is a Python Shell.
  - c) Both a and b
  - d) None of the above
2. Which of the following is/are not valid variable names in python?
  - a) length (white spaces appended before the word length)
  - b) 100\_emp
  - c) abc\$
  - d) All of the above
3. Which of the following gives an output as 8?
  - a) >>> 2 \* 3
  - b) >>> 2 \*\* 3
  - c) >>> 2 ^^ 3
  - d) None of the above

---

Figure 1-16. Checkpoint (1 of 2)

Write your answers here:

- 1.
- 2.
- 3.



## Checkpoint (2 of 2)

### Fill in the blanks:

1. Refer the code given below and fill in the blanks:

```

1. num = 16
2. num1 = num / 6
3. num2 = num // 6
4. num3 = num // 6.0
5. print("num1 =", num1)
6. print("num2 =", num2)
7. print("num3 =", num3)

```

Ans: num1= \_\_\_\_\_ , num2= \_\_\_\_\_ , num3= \_\_\_\_\_

1. Consider the following expression. Select the correct operators in place of a, b, c, d from the following given options so that the output is 55.

7  a  6  b  2%4\*(9  c  5)  d 2

Ans: \_\_\_\_\_

2. Python supports various data types for variables, such as \_\_\_\_\_.  
 3. The \_\_\_\_\_ in python is used to take user input from the console.

### True/False:

1. global = '30' creates a string variable in Python. True/False
2. Assert = True creates a Boolean variable in Python. True/False
3. Variable names are case-sensitive. True/False

Figure 1-17. Checkpoint (2 of 2)

Write your answers here:

Fill in the blanks:

- 1.
- 2.
- 3.
- 4.

True or false:

- 1.
- 2.
- 3.



## Question bank (1 of 2)

### Two-mark questions:

1. Consider two variables 'a' and 'b' in Python such that  $a = 4$  and  $b = 5$ . Swap the values of 'a' and 'b' without using a temporary variable. Print the values of 'a' and 'b' before and after swapping.
2. Given the value of radius of a circle, write a Python program to calculate the area and perimeter of the circle. Display both the values
3. Assume the following Python code.
4. Suppose the user enters the value for val1 as 10 and val2 as 5, predict the output:

### Four-mark questions:

1. In a retail application, shopkeeper wants to keep a track of following details of a customer. Sample values are provided.
2. Write a Python program for the following requirements:

Prompt the user to input two numbers num1 and num2  
Increment num1 by 4 and num2 by 6  
Find and print the sum of new values of num1 and num2  
Hint – Use type casting for converting the input into an integer.
3. Explain the concept of operators in Python. Discuss at least three types of operators and provide examples for each.
4. Explain the concept of dynamic typing in Python and highlight two features of Python that leverage dynamic typing. Support your explanation with an example for each feature.

Figure 1-18. Question bank (1 of 2)

## Question bank (2 of 2)



IBM ICE (Innovation Centre for Education)

### Eight-mark questions:

1. The finance department of a company wants to compute the monthly pay of its employees. Monthly pay should be calculated as mentioned in the formula below. Display all the employee details. Monthly Pay = Number of hours worked in a week \* Pay rate per hour \* No. of weeks in a month

The number of hours worked by the employee in a week should be considered as 40

Pay rate per hour should be considered as Rs.400

Number of weeks in a month should be considered as 4

Write a Python program to implement the above real world problem.

2. Consider the scenario of processing marks of a student in ABC Training Institute. John, the student of fifth grade takes exams in three different subjects. Create three variables to store the marks obtained by John in three subjects. Find and display the average marks scored by John. Now change the marks in one of the subjects and observe the output. Did the value of average change?

---

Figure 1-19. Question bank (2 of 2)

## Unit summary



IBM ICE (Innovation Centre for Education)

- Having completed this unit, you should be able to:
  - Understand the fundamentals of the Python programming language and its historical development
  - Master Python basics, including data types, operators, and input/output operations

---

*Figure 1-20. Unit summary*

Unit summary is as stated above.

---

# Unit 2. Control Flow and Loops

## Overview

This Unit will provide an overview of Implement control statements and loops to make decisions and perform iterative tasks, Implement Python basics and effectively work with variables, data types, and collections.

## How you will check your progress

- Checkpoint

## References

IBM Knowledge Center



IBM ICE (Innovation Centre for Education)

## Unit objectives

- After completing this unit u should be able to:
  - Implement control statements and loops to make decisions and perform iterative tasks
  - Implement Python basics and effectively work with variables, data types, and collections
- Learning outcomes:
  - Implement Python basics and effectively work with variables, data types, and collections
  - Apply control statements and loops to solve complex problems and execute iterative tasks

---

Figure 2-1. Unit objectives

Unit objectives and outcomes are as stated above.



IBM ICE (Innovation Centre for Education)

# Introduction

- In python, control flow is managed through conditional statements and loops. Conditional statements like "if," "elif," and "else" allow for the execution of specific code based on defined conditions.
- Think of these as decision points in the program. Loops, such as "for" and "while," enable the repetitive execution of code.
- The "for" loop iterates over sequences or iterable objects, while the "while" loop continues execution as long as a specified condition remains true.
- Break and continue statements provide the ability to exit loops prematurely or skip iterations, enhancing control over the program's flow.
- Together, these constructs govern the sequence and repetition of operations, offering flexibility and structure to python programs.

*Figure 2-2. Introduction*

## Control statements

Fundamental building blocks of programming languages, known as control statements, enable the regulation of program execution flow. The normal sequential flow of a program is modified through the utilization of control statements, which are considered essential for crafting efficient and effective code.

In Python, three primary types of control statements exist:

- Decision making statements (e.g., if and elif): Decision making statements provide the ability to make decisions based on conditions, with if and elif being the most common.
- Looping statements (e.g., for and while loops): Looping statements enable the repetition of a code block multiple times, with for and while loops being the most common.
- Jump statements (e.g., break and continue): Jump statements facilitate the abrupt alteration of program execution flow, with break and continue being the most common.

## Conditional statements

Conditional statements are used to control the flow of a program based on certain conditions. The most common decision-making statements in Python are the if and else statements.

### if statement

The if statement is used to execute a block of code conditionally.

### Single if statement

The if statement is used for basic conditional execution. It checks a single condition, and if that condition is true, the indented block of code beneath it is executed. If the condition is false, the block is skipped.

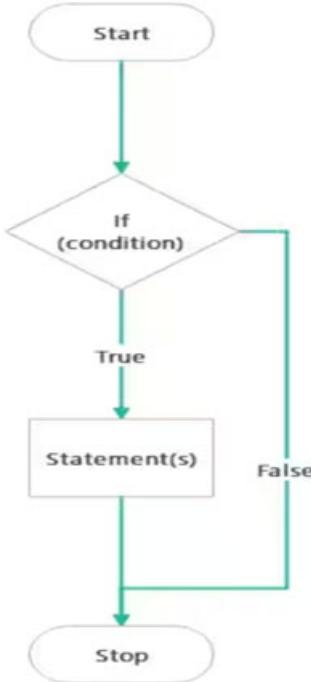


Figure 1: Single if statement

### **Example:**

In a real-life scenario, a single "if" statement in Python can be used to check a specific condition and perform an action based on the result of that condition.

For example, let's consider a simple temperature monitoring system. In this system, the temperature of a room is measured, and if the temperature exceeds a certain threshold, an alert is triggered to notify the occupants.

```

temperature = 28 # Temperature in Celsius
threshold = 25 # Threshold temperature in Celsius
if temperature > threshold:
    print("Alert: Temperature is above the threshold! Please take necessary precautions.")
  
```

In this example, the "if" statement checks if the current temperature (temperature) is greater than the predefined threshold temperature (threshold). If the condition is true (i.e., if the temperature exceeds the threshold), the system triggers an alert message informing the occupants that the temperature is above the threshold and advises them to take necessary precautions.

### **Multiple if statements**

When there are independent conditions, and each condition needs to be checked irrespective of the truth or falsity of the previous conditions, multiple if statements are employed. Each if statement is independently evaluated.

### **Example:**

In a real-life scenario, multiple "if" statements in Python can be used to evaluate multiple conditions and perform different actions based on the results of those conditions.

Let's consider a simple grading system for a class. In this system, students' grades are calculated based on their scores in different subjects. Depending on their scores, students are assigned different grades such as: A, B, C, or D.

```

math_score = 85
science_score = 75
english_score = 90
if math_score >= 90:
    math_grade = "A"
elif math_score >= 80:
    math_grade = "B"
elif math_score >= 70:
    math_grade = "C"
else:
    math_grade = "D"

if science_score >= 90:
    science_grade = "A"
elif science_score >= 80:
    science_grade = "B"
elif science_score >= 70:
    science_grade = "C"
else:
    science_grade = "D"

if english_score >= 90:
    english_grade = "A"
elif english_score >= 80:
    english_grade = "B"
elif english_score >= 70:
    english_grade = "C"
else:
    english_grade = "D"

print("Math Grade:", math_grade)
print("Science Grade:", science_grade)
print("English Grade:", english_grade)

```

In this example, each subject's score is evaluated separately using multiple "if" statements. Depending on each student's score in math, science, and English, the corresponding grade is assigned based on predefined score ranges. Finally, the grades for each subject are displayed.

### **else statement**

Used for an alternative block of code that executes when the condition in the corresponding if statement is not met. It provides a default option for the program.

### **Example:**

In a real-life scenario, an "else" statement in Python can be used to specify an action to be taken if none of the preceding conditions in an "if" statement are met. Let's consider a simple example of determining whether a person is eligible to vote based on their age.

```
age = 20
voting_age = 18
if age >= voting_age:
    print("You are eligible to vote!")
else:
    print("You are not eligible to vote yet.")
```

In this example, the "if" statement checks if the person's age (age) is greater than or equal to the voting age (voting\_age). If the condition is true (i.e., if the person is eligible to vote), the message "You are eligible to vote!" is printed. If the condition is false (i.e., if the person is not eligible to vote), the "else" statement is executed, and the message "You are not eligible to vote yet." is printed.

### **if-else statement**

The if-else statement enables the creation of a decision structure where one block of code is executed if a condition is true, and another block is executed if the condition is false. It represents a binary choice.

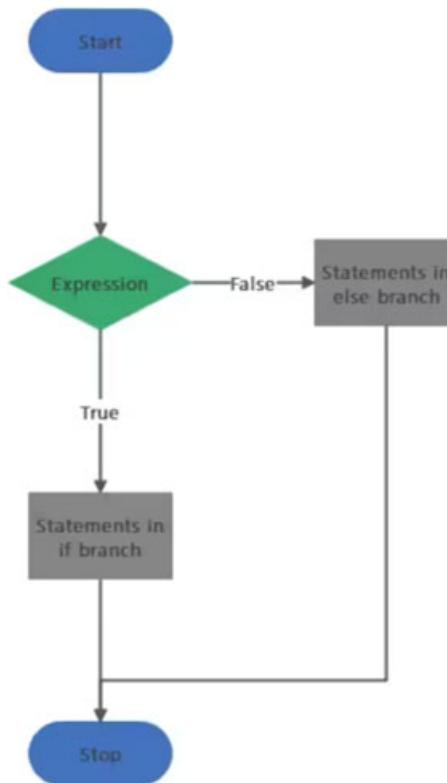


Figure 2: if-else statement

### **Example:**

In a real-life scenario, we can use the "else" statement in Python to handle cases where a certain condition is not met. In the example of a simple login system, a user is prompted to enter their username and password. If the username and password match what is stored in the system, access is granted to the user. Otherwise, if the username or password is incorrect, an error message is displayed by the system.

```
username = "user123"
```

```
password = "password123"
input_username = input("Enter your username: ")
input_password = input("Enter your password: ")
if input_username == username and input_password == password:
    print("Login successful! Welcome, " + username + "!")
else:
    print("Invalid username or password. Please try again.")
```

### **elif (else if) statement**

The if statement is used in conjunction with if and else to check multiple conditions in sequence. If the condition in the if statement is false, it moves on to the elif statement, and if that condition is true, the corresponding block is executed. An example illustrating multiple elif statements is shown below:

#### **Example:**

In the example of a simple login system, after checking whether the username and password match what is stored in the system, if they do not match, the system may further evaluate the input. If the username is correct but the password is incorrect, or vice versa, the system can display a specific error message. This is achieved using an "else if" statement, also known as "elif" in Python.

#### **1 Note:**

- If a binary decision (two possible outcomes) is encountered, the use of if-else is warranted.
- When conditions are mutually exclusive, indicating that only one should be true, and the intention is to execute the block corresponding to the first true condition, elif is often deemed more appropriate.
- In cases where conditions are independent, and the goal is to check all of them, the preference might be towards using multiple if statements.



IBM ICE (Innovation Centre for Education)

# Logical operators and conditions

- Logical operators and conditions:
  - Logical operators and conditions in python are fundamental components that enable the formulation of complex conditions for decision-making in programs.
  - These operators allow the combination of multiple conditions to create more intricate and nuanced logical expressions.
  - Conditions in python are expressions that evaluate to either true or false. They are extensively used in control flow statements such as if, elif, and while to make decisions based on the evaluation of these expressions.

---

*Figure 2-3. Logical operators and conditions*

## Using Logical AND, OR, NOT

Operator	Description	Example	Python Code
AND (and)	Combines two conditions. The result is True only if both conditions are True.	Granting library access: requires applicant to be at least 18 and have valid ID.	def grant_library_access(age, has_id): return age >= 18 and has_id
		Security clearance: requires passing background check and completing training course.	def grant_security_clearance(passed_check, completed_training): return passed_check and completed_training
OR (or)	Combines two conditions. The result is True if at least one condition is True.	Ordering food: allows delivery or pickup.	def order_food(delivery): if delivery: # Order for delivery else: # Order for pickup
		Travel restrictions: allows entry with vaccination or negative test.	def allow_entry(vaccinated, negative_test): return vaccinated or negative_test
NOT (not)	Inverts the result of a condition. True becomes False, and vice versa.	Checking eligibility: applicant is not eligible if income exceeds threshold.	def check_eligibility(income, threshold): return not (income > threshold)
		Room availability: room is not available if already occupied.	def check_availability(occupied): return not occupied

Table 1: Using Logical AND, OR, NOT

### Complex conditions

Complex conditions in Python involve the use of logical operators (and, or, not) to create compound expressions that evaluate to either True or False. These conditions allow programmers to express more sophisticated decision-making logic by combining multiple individual conditions.

### Complex condition example

In this example, complex conditions are constructed using logical operators to check eligibility criteria. The Operator ensures that both conditions must be satisfied, while the Operator allows for flexibility by considering either of the conditions. Parentheses are used to control the order of evaluation, emphasizing the importance of understanding the operator precedence. Understanding and utilizing complex conditions enable programmers to express intricate decision-making logic in a concise and readable manner, making their code more effective and flexible.

```
age = 25
is_student = False
# Using AND Operator
if age >= 18 and not is_student:
    print("You are eligible to vote.")
# Using OR Operator
if age < 18 or is_student:
    print("You are either under 18 or a student.")
```

```
# Combining multiple conditions
if (age >= 18 and not is_student) or (age >= 21 and is_student):
    print("You meet specific eligibility criteria.")
```



IBM ICE (Innovation Centre for Education)

## Switch-case (if-elif-else)

- Switch-case (if-elif-else):
  - In Python, there isn't a direct implementation of a traditional switch-case statement, as seen in some other programming languages.
  - However, the equivalent functionality can be achieved using if-elif-else statements.

Figure 2-4. Switch-case (if-elif-else)

```
variable = "value"  # The variable to be tested
if variable == "value1":
    # Code block executed if variable equals "value1"
    print("This is value1.")
elif variable == "value2":
    # Code block executed if variable equals "value2"
    print("This is value2.")
elif variable == "value3":
    # Code block executed if variable equals "value3"
    print("This is value3.")
else:
    # Code block executed if none of the above conditions are met
    print("This is the default case.")
```

### Key points:

- **Variable testing:** The variable to be tested is compared against different values in each if or elif statement.

- **elif (Else If):** elif is used for multiple conditions after the initial if statement.
- **Default case (else):** The else statement provides a default case to be executed when none of the preceding conditions are met.

**Example:**

```
day_of_week = "Monday"
if day_of_week == "Monday":
    print("It's the start of the week.")
elif day_of_week == "Wednesday":
    print("Midweek vibes.")
elif day_of_week == "Friday":
    print("Weekend is near.")
else:
    print("It's an ordinary day.)
```

In this example, the program checks the value of day\_of\_week and executes the corresponding code block. If none of the specified conditions match, the else block is executed.

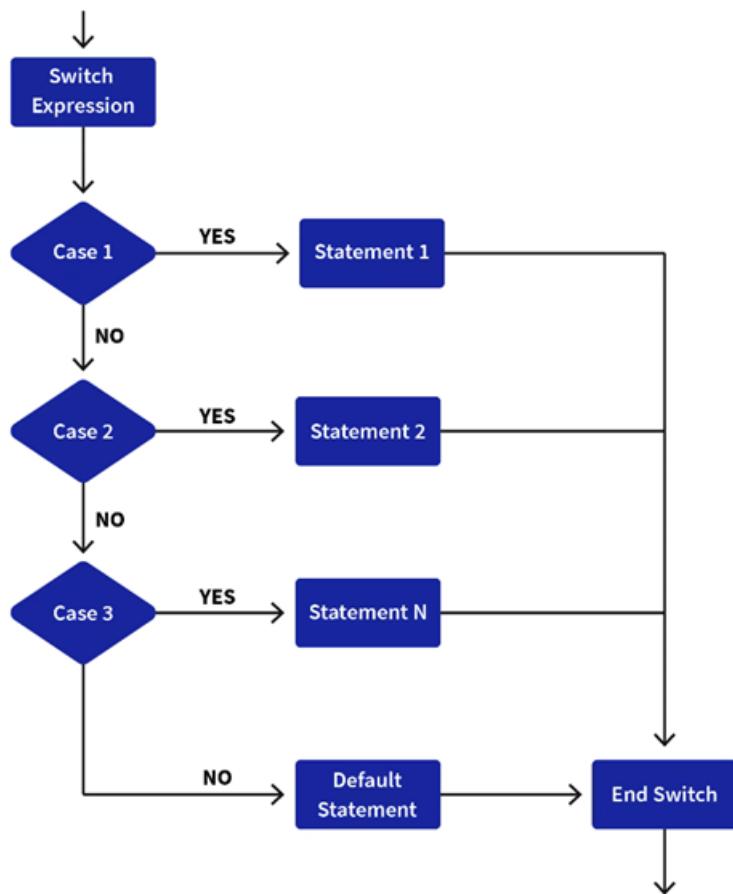


Figure 3: Switch case flowchart

Figure 3: This flowchart illustrates the functionality of the switch-case construct. It demonstrates a sequential evaluation of conditions. Initially, each condition is examined in order. If none of the conditions is met, a default statement is executed. [<https://www.scaler.com/topics/switch-case-in-python/>]

While Python doesn't have a switch-case statement, the if-elif-else structure provides a flexible and readable alternative to handle multiple cases based on the value of a variable.

### Implementing switch-like behavior

In Python, in the absence of a direct switch-case construct, similar behavior can be emulated using a dictionary. The concept involves utilizing a dictionary to associate different cases with corresponding functions or code blocks. An example is considered where switch-like behavior is implemented for processing various types of requests in a web server.

```
def process_get_request():
    print("Processing GET request.")

def process_post_request():
    print("Processing POST request.")

def process_put_request():
    print("Processing PUT request.")

def process_delete_request():
    print("Processing DELETE request.")

def handle_unknown_request():
    print("Unknown request type. Ignoring.")

# Mapping request types to functions
request_handlers = {
    "GET": process_get_request,
    "POST": process_post_request,
    "PUT": process_put_request,
    "DELETE": process_delete_request,
}

# Simulating different requests
requests = ["GET", "POST", "PUT", "PATCH", "DELETE"]

# Process each request
for request in requests:
    # Check if the request type is in the dictionary. If yes, call the corresponding function; if not, call handle_unknown_request
    request_handlers.get(request, handle_unknown_request)()
```

In this example:

- Functions `process_get_request`, `process_post_request`, `process_put_request`, and `process_delete_request` represent the code blocks for handling different types of HTTP requests.
- The `handle_unknown_request` function is called when the request type is unknown.
- The `request_handlers` dictionary maps request types to their corresponding processing functions.
- The loop simulates processing different types of HTTP requests.

Example 2

Example of both a switch-case-like structure using if-elif-else in Python, along with a basic flowchart illustrating the logic:

```
def switch_case(argument):
    """Function to mimic switch-case behavior."""
    # Dictionary to map cases to corresponding functions
    switcher = {
        1: case1_function,
        2: case2_function,
        3: case3_function,
        4: case4_function,
    }

    # Get the function corresponding to the argument or default function
    func = switcher.get(argument, default_function)
    # Execute the selected function
    func()

def case1_function():
    print("You chose case 1.")

def case2_function():
    print("You chose case 2.")

def case3_function():
    print("You chose case 3.")

def case4_function():
    print("You chose case 4.")

def default_function():
    print("Invalid choice.")

# Example usage
choice = int(input("Enter your choice (1-4): "))
switch_case(choice)
```

And here's a basic flowchart illustrating the logic:

```

Start
|
V
Get user input
|
V
Check input value
|
V
Is input value 1? --> (Yes) --> Execute case1_function() --> End
|
|
No
|
V
Is input value 2? --> (Yes) --> Execute case2_function() --> End
|
|
No
|
V
Is input value 3? --> (Yes) --> Execute case3_function() --> End
|
|
No
|
V
Is input value 4? --> (Yes) --> Execute case4_function() --> End
|
|
No
|
V
Execute default_function() --> End

```

Here are some common errors that are applicable to various conditional statements, including switch-case constructs in Python using dictionaries:

- Missing default handling: For switch-case constructs implemented using dictionaries, forgetting to include a default case can lead to errors when the input doesn't match any of the defined cases.
- Forgetting parentheses: When calling functions within the dictionary values, it's essential to include parentheses after the function name. Omitting these parentheses can lead to unexpected behavior or errors.
- Mutable objects as keys: Using mutable objects such as lists or dictionaries as keys in the switch-case dictionary can lead to unexpected behavior because mutable objects cannot be used as dictionary keys.
- Inconsistent type handling: If the input type is not consistent with the keys in the switch-case dictionary (e.g., using strings as keys when the input is an integer), it can lead to errors or the default case being triggered unintentionally.
- Redundant function definitions: Defining separate functions for each case when those functions are simple and could be implemented inline can lead to unnecessary complexity and decreased readability.
- Variable scope issues: If functions called within the switch-case construct rely on variables defined outside the switch-case function, ensure proper scoping to prevent unintended behavior.
- Improper error handling: Not properly handling exceptions that may occur within the switch-case construct, such as errors in function calls or invalid input, can lead to unexpected program behavior or crashes.

- Confusing or non-descriptive case labels: Using case labels in the switch-case construct that are not descriptive or intuitive can make the code difficult to understand and maintain.



IBM ICE (Innovation Centre for Education)

## Loops

- Loops, fundamental programming constructs, enable the repeated execution of a block of code until a specified condition is met.
- They are crucial for automating tasks involving repetitive operations. Two main types of loops, for loops and while loops are provided by python.

*Figure 2-5. Loops*

### While loop: Using while loops for iteration

A while loop in Python is used to repeatedly execute a block of code as long as a given condition is true. The general syntax is as follows:

```
count = 0
while count < 5:
    print(count)
    count += 1
```

In this example, the while loop continues to execute as long as the count variable is less than 5. Inside the loop, the current value of the count is printed, and then the count is incremented by 1 in each iteration. It's important to ensure that the condition in the while loop will eventually become false; otherwise, the loop will run indefinitely, causing a program to hang or crash. To avoid infinite loops, make sure that the condition is appropriately updated within the loop.

#### Example:

In this game:

- The program generates a random number between 1 and 100 as the secret number.
- Inside the while loop, the player is prompted to enter their guess.
- If the player enters 'q', the game ends.

- If the player enters a valid number, the program checks if it matches the secret number.
- If it matches, the player wins and the game ends.
- If it's too low or too high, the player is informed and prompted to guess again.
- The loop continues until the player guesses the correct number or decides to quit.

```
import random

print("Welcome to the Number Guessing Game!")

print("Try to guess the secret number between 1 and 100.")

print("Enter 'q' at any time to quit.")

# Generate a random number between 1 and 100
secret_number = random.randint(1, 100)

while True:
    guess = input("Enter your guess (or 'q' to quit): ")

    if guess.lower() == 'q':
        print("Quitting the game...")
        break

    try:
        guess = int(guess)
    except ValueError:
        print("Invalid input. Please enter a number or 'q' to quit.")
        continue

    if guess == secret_number:
        print("Congratulations! You guessed the correct number:", secret_number)
        break

    elif guess < secret_number:
        print("Too low! Try again.")

    else:
        print("Too high! Try again.")
```

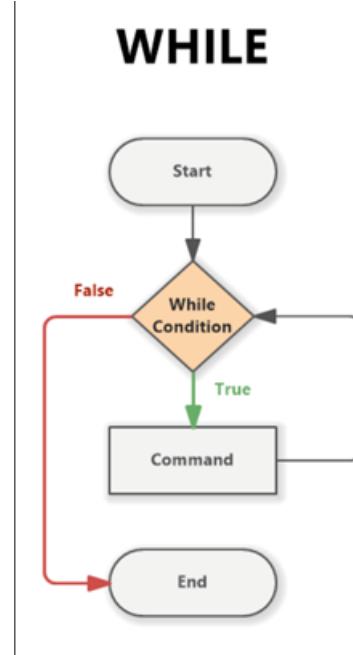


Figure 4: While flow chart

Source: [<https://www.softwareideas.net/a/1679/while-and-do-while-loop-flowchart->]

### While with else

In Python, the `else` statement can be used with a `while` loop to execute a block of code when the loop terminates naturally, i.e., when the loop condition becomes false. Here are some reasons why we use `else` with a `while` loop:

- **Handling completion conditions:** The `else` block provides a convenient way to execute code when the loop completes all iterations without encountering a `break` statement. This can be useful for handling completion conditions or providing feedback after the loop finishes its task.
- **Improved readability:** Including the `else` block directly after the `while` loop enhances code readability by clearly indicating what happens when the loop finishes normally. It makes the code more expressive and easier to understand at a glance.
- **Reduced complexity:** Using `else` with a `while` loop can help reduce code complexity by eliminating the need for additional flags or variables to track loop termination conditions. This simplifies the code structure and reduces the risk of errors or confusion.
- **Single point of exit:** With the `else` block, you have a single point of exit for the loop, which can make the code easier to manage and maintain. You don't need to scatter exit conditions throughout the loop body or rely on complex control flow mechanisms.
- **Semantic clarity:** Including an `else` block with a `while` loop communicates the intent of the code more clearly. It explicitly states that the block of code within the `else` will execute if the loop completes all iterations without interruption.

### Example code without using else statement with while:

In the below example, the correct PIN is defined as "1234". The number of attempts is initialized to zero. The maximum number of attempts allowed is set to 3. A flag, indicating whether access is granted, is initialized as `False`.

A loop is entered, continuing until either the maximum number of attempts is reached or access is granted. Within the loop, the user is prompted to enter their PIN code. If the entered PIN matches the correct PIN, "Access granted!" is printed, and the flag indicating access is granted is set to `True`.

If the entered PIN is incorrect, the number of attempts is incremented, and the remaining attempts are calculated and printed. After the loop terminates, if access has not been granted, a message stating "You've exceeded the maximum number of attempts. Access denied." is printed.

```

# Initialize the number of attempts
attempts = 0

# Set the maximum number of attempts allowed
max_attempts = 3

# Initialize a flag to track whether access is granted
access_granted = False

# Loop until the maximum number of attempts is reached
while attempts < max_attempts:
    # Prompt the user to enter the PIN
    entered_pin = input("Enter your PIN code: ")

    # Check if the entered PIN is correct
    if entered_pin == correct_pin:
        print("Access granted!")
        access_granted = True # Set the flag to True if access is granted
        break # Exit the loop if the correct PIN is entered

    # Increment the number of attempts
    attempts += 1
    remaining_attempts = max_attempts - attempts
    print("Incorrect PIN. Please try again. Remaining attempts:", remaining_attempts)

# Check if access is not granted after the loop terminates
if not access_granted:
    print("You've exceeded the maximum number of attempts. Access denied.")

```

### Code using with statement:

In the context of the below example, the else block of the while loop is significant because it allows us to handle a specific condition: when the loop terminates naturally (i.e., without using break).

**Handling Exceeded Attempts:** If the user fails to enter the correct PIN within the maximum number of attempts (max\_attempts), the while loop will naturally terminate because the condition attempts < max\_attempts becomes false. Without the else block, we would need additional logic after the loop to check whether the maximum attempts were exceeded. With the else block, we can handle this condition directly within the loop.

```
# Initialize the number of attempts
attempts = 0

# Set the maximum number of attempts allowed
max_attempts = 3

# Loop until the maximum number of attempts is reached
while attempts < max_attempts:
    # Prompt the user to enter the PIN
    entered_pin = input("Enter your PIN code: ")

    # Check if the entered PIN is correct
    if entered_pin == correct_pin:
        print("Access granted!")
        break # Exit the loop if the correct PIN is entered

    # Increment the number of attempts
    attempts += 1
    remaining_attempts = max_attempts - attempts
    print("Incorrect PIN. Please try again. Remaining attempts:", remaining_attempts)

else:
    print("You've exceeded the maximum number of attempts. Access denied.")
```



IBM ICE (Innovation Centre for Education)

## For loops

- The for loop in Python is utilized for iterating over a sequence of elements, such as a list or tuple, and executing a specified block of code for each element in the sequence.
- During each iteration, the loop variable takes on the value of the current element, allowing operations to be performed based on individual elements.
- This construct enhances code readability and conciseness when repetitive tasks need to be applied to a collection of items.
- The sequence can be of any iterable type, and the loop iterates until all elements in the sequence are exhausted.

---

Figure 2-6. For loops

The basic syntax of a for loop is as follows.

**For items in sequence:**

```
# code to execute for each item in the sequence
```

In this context, the sequence denotes the collection of items to iterate over, and the item signifies the current item in the sequence during each iteration. The block of code following the for statement will be executed for each item in the sequence.

**Iterating over a sequence**

Topic	Description	Example	Output
Iterating over a list	The for loop iterates over each element in the list 'fruits', and for each iteration, the current element is assigned to the variable 'fruit', and the 'print(fruit)' statement is executed.	'python fruits = ["apple", "banana", "cherry"] for fruit in fruits: print(fruit)'	apple banana cherry
Iterating over a tuple	The loop iterates over the tuple 'days' and prints each day to the console.	'python days = ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday") for day in days: print(day)'	Monday Tuesday Wednesday Thursday Friday
Iterating over a string	This example counts the number of vowels in the given string 'my_string'. The for loop iterates over each character in the string and checks whether it is a vowel.	'python my_string = "Python is amazing!" vowel_count = 0 for char in my_string: if char.lower() in "aeiou": vowel_count += 1 print(f"The number of vowels in the string is: {vowel_count}")'	The number of vowels in the string is: 7
Iterating over a dictionary	This loop iterates over the keys of the dictionary 'user_data' and prints each key to the console.	'python user_data = {"name": "Alice", "age": 30, "city": "New York"} for key in user_data: print(key)'	name age city

## Using range function

The **range()** function is often used in conjunction with loops in Python to generate a sequence of numbers that the loop iterates over. It provides a convenient way to control the number of iterations and the specific range of values. The range() function will return a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

The range() function will return a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

- **Iterating over a range of numbers:**

```
for i in range(5):
```

```
    print(i)
```

This loop will print numbers from 0 to 4 (not including 5), as the range(5) generates a sequence of numbers from 0 to 4.

- **Specifying start and end in range()**

```
for i in range(2, 8):
```

```
    print(i)
```

This loop will print numbers from 2 to 7 (not including 8), as the range(2, 8) generates a sequence of numbers starting from 2 up to 7, but not including 8.

- **Adding a step in range()**

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

```
    print(i)
```

This loop will print odd numbers from 1 to 9, as range(1, 10, 2) generates a sequence starting from 1, incrementing by 2 in each step.

- **Iterating in reverse**

```
for i in range(5, 0, -1):
```

```
    print(i)
```

- **Using range() with len() for Index-based Iteration**

Assuming there is a list of numbers, and the goal is to print both the index and the corresponding element for each item in the list, this can be accomplished by utilizing both range() and the index in a for loop.

```
numbers = [10, 20, 30, 40, 50]
for i in range(len(numbers)):
    print(f"Index: {i}, Value: {numbers[i]}")
```

Output:

Index: 0, Value: 10

Index: 1, Value: 20

Index: 2, Value: 30

Index: 3, Value: 40

Index: 4, Value: 50

In this example, range(len(numbers)) generates a sequence of indices corresponding to the length of the numbers list. The loop then iterates over these indices, and for each index i, it prints both the index and the value at that index in the numbers list.



IBM ICE (Innovation Centre for Education)

# Loop control statements

- Loop control statements:
  - Loop control statements are specialized statements in programming languages that permit the modification of the flow of execution within a loop.
  - Ways to control how loops iterate, enabling the skipping of iterations, early termination of loops, or handling of specific conditions within the loop structure, are provided by them.

---

Figure 2-7. Loop control statements

## Purpose of loop control statements:

Efficient and flexible loops are facilitated by loop control statements, allowing for the following:

- **Iteration skipping:** Specific iterations of a loop can be skipped based on certain conditions, avoiding unnecessary processing.
- **Early termination of loops:** A loop can be terminated before reaching its natural ending point based on specific conditions, preventing unnecessary iterations.
- **Handling special cases:** Specific conditions or cases within a loop can be addressed without disrupting the overall loop structure.

## Common types of loop control statements:

While different programming languages possess their own set of loop control statements, some common ones include:

- **Break:** This statement abruptly terminates the current loop and exits the loop structure.
- **Continue:** The remaining code in the current iteration of the loop is skipped, and the loop proceeds to the next iteration.
- **Pass:** This statement serves as a placeholder that does nothing, often used when a statement in a loop structure is needed but no code execution is desired in that particular iteration.
- **Goto:** This less common statement can be considered a jump statement, allowing the transfer of control to a specific label or location within the code.

## **Break statement**

The break statement is employed to exit the loop prematurely, regardless of whether the loop condition remains true. It is frequently used when a specific condition is met, and there is a desire to terminate the loop immediately.

### **Example using while loop-**

```
count = 0
while count < 5:
    print(count)
    if count == 2:
        break
    count += 1
```

In this example, the loop will exit when the count becomes 2 due to the break statement.

### **Example using for loop-**

```
for i in range(5):
    if i == 3:
        print("Breaking the loop at i =", i)
        break
    print(i)
```

#### **Output:**

```
0
1
2
```

Breaking the loop at i = 3

In this example, the loop iterates from 0 to 4. When **i** becomes 3, the **break** statement is encountered, and the loop is terminated.

## **Continue statement**

The continue statement is utilized to skip the remaining code inside a loop and proceed to the next iteration. It is commonly used when there is a need to skip specific iterations based on a condition.

```
count = 0
while count < 5:
    count += 1
    if count == 2:
        continue
    print(count)
```

In this example, when the count is equal to 2, the continue statement is encountered, and the loop proceeds to the next iteration without executing the print statement.

### **Example combining break and continue:**

```
count = 0
while count < 5:
    count += 1
    if count == 2:
```

```

        continue
print(count, end= " ")
if count == 4:
    break

```

### **Here's the step-by-step breakdown of the code:**

- The count starts at 0.
- In the first iteration (count == 1), the count is incremented to 1, and it does not equal 2, so the continue statement is not triggered. The value 1 is printed.
- In the second iteration (count == 2), the count is incremented to 2. The continue statement is triggered, so the loop skips the remaining code in the loop body for this iteration.
- In the third iteration (count == 3), the count is incremented to 3, and it does not equal 4, so the loop prints the value 3.
- In the fourth iteration (count == 4), the count is incremented to 4. The value 4 is printed, and then the break statement is encountered, terminating the loop.

**Output:** 1 3 4

### **else with loops**

The else clause in loops in Python provides a way to execute a block of code after the loop has finished its iterations. The key distinction is that the else block is executed only if the loop condition becomes False, and it is not executed if a break statement terminates the loop. Here are a few use cases where the else clause with loops is commonly employed.

- **Indicate Successful Iteration:** In this example, the else block is executed after the for loop has iterated over all values from 0 to 4. This can be useful for indicating that the loop is completed without encountering any issues.

```

for i in range(5):
    print(i)
else:
    print("Loop finished successfully.")

```

- **Search element in a list:** Here, the else block is executed if the for loop completes without encountering a break. This is frequently employed when searching for an element in a list.

```

search_item = 3
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    if num == search_item:
        print(f"Found {search_item} in the list.")
        break
else:
    print(f"{search_item} not found in the list.")

```

- **Validate Input:** In this example, the else block is used in combination with a while loop to validate user input. The loop keeps prompting the user until a valid positive number is entered.

```

while True:
    user_input = input("Enter a positive number: ")
    if user_input.isdigit() and int(user_input) > 0:
        print("Valid input.")

```

```
        break
else:
    print("Invalid input. Please try again.")
else:
    print("This will not be executed because of the break.")
```



IBM ICE (Innovation Centre for Education)

## Sorted vs. Sort in python

- Sorting is a fundamental operation in programming used to arrange data in a specific order.
- Python provides built-in functions and methods for sorting various data structures, including lists, tuples, and dictionaries.

*Figure 2-8. Sorted vs. Sort in python*

### Sort method

- Applicable to any iterable: sorted is a built-in function that can be applied to any iterable (lists, tuples, strings, etc.). It returns a new sorted list.
- In-place sorting: Unlike sort, sorted does not modify the original sequence. Instead, it returns a new sorted list.

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
sorted_numbers = sorted(numbers)
print(sorted_numbers) # Output: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
print(numbers)      # Original list is unchanged: [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
```

### Example: Sorting a dictionary based on keys

To sort a dictionary based on its keys or values in different cases, the sorted function can be used along with a custom sorting function or lambda expression. An example in Python is presented below-

```
my_dict = {'apple': 3, 'Banana': 2, 'Orange': 1}
```

### # Sorting by keys in a case-insensitive manner

```
sorted_keys = sorted(my_dict.keys(), key=lambda x: x.lower())
```

```
sorted_dict_keys = {key: my_dict[key] for key in sorted_keys}
```

### Sorting a dictionary based on values

In this example, key=lambda item: item[1] specifies that the sorting should be based on the values (the second element in each key-value pair).

```
my_dict = {'apple': 3, 'banana': 2, 'orange': 1}
```

### # Sorting the dictionary by values in ascending order

```
sorted_dict = dict(sorted(my_dict.items(), key=lambda item: item[1]))
```

```
print("Sorted dictionary by values (ascending):", sorted_dict)
```

To sort the dictionary in descending order, the reverse=True parameter can be added to the sorted function-

### # Sorting the dictionary by values in descending order

```
sorted_dict_desc = dict(sorted(my_dict.items(), key=lambda item: item[1], reverse=True))
```

```
print("Sorted dictionary by values (descending):", sorted_dict_desc)
```

It should be noted that these examples assume the sorting of numeric values. In cases where the values are strings, a conversion to a suitable numeric format before sorting may be necessary.

### Sort method

- Applicable to lists only: sort is a method that can be applied specifically to lists. It modifies the original list in place.
- In-place sorting: The sort method sorts the elements of a list in ascending order and does not create a new list. It directly modifies the existing list.

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

```
numbers.sort()
```

```
print(numbers) # Output: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
```

Highlight or bold it in different colour. Because couldn't find the difference between the statements.

Also difference of sort and sorted can be in table format with description.

Feature	'sort()' Method	'sorted()' Function
Usage	Available only for lists.	Available for any iterable (list, tuple, string, etc.)
Modifies Original	Sorts the list in-place, modifying the original list.	Does not modify the original iterable, instead returns a new sorted list.
Return Value	Returns 'None'.	Returns a new sorted list.
Syntax	'list.sort(key=None, reverse=False)'	'sorted(iterable, key=None, reverse=False)'
Example	'python numbers = [3, 1, 2] numbers.sort() print(numbers)'	'python numbers = [3, 1, 2] sorted_numbers = sorted(numbers) print(sorted_numbers)'
Stability	Stable. The relative order of equal elements remains unchanged.	Stable. The relative order of equal elements remains unchanged.
Efficiency	More efficient for large lists as it doesn't create a new list.	Creates a new list, which can be less efficient for very large iterables.
Error on Non-List	Cannot be used on non-list types.	Can be used on any iterable, including lists.

Table 2: Sort method

## Summary

The chapter includes discussions on control statements, encompassing conditional statements like the if statement, if-else statement, and elif (else if) statement. Logical operators and conditions are explored, including the usage of logical AND, OR, and NOT, as well as complex conditions. The Switch-Case (if-elif-else) structure is covered, along with its implementation of switch-like behavior. The section on loops includes the usage of while loops and for loops for iteration, along with loop control statements such as break and continue. Additionally, the else statement with loops is addressed. The content delves into Python data types, including lists, tuples, sets, and dictionaries, discussing their characteristics. The distinction between sorted and sort in Python is examined, detailing the sorted method and sort method.

## Self evaluation: Exercise 6



IBM ICE (Innovation Centre for Education)

- **Exercise 6:** Even or odd checker.
- **Estimated time:** 00:10 minutes.
- **Aim:** Implement a python program that checks if a given number is even or odd and prints the result.
- **Learning objective:**
  - The learner will understand how to implement a python program to determine if a number is even or odd.

---

Figure 2-9. Self evaluation: Exercise 6

Self evaluation exercise 6 is as stated above.

## Self evaluation: Exercise 7



IBM ICE (Innovation Centre for Education)

- **Exercise 7:** Largest among three numbers.
- **Estimated time:** 00:10 minutes.
- **Aim:** Implement a python program that finds and prints the largest among three given numbers.
- **Learning objective:**
  - The learner will understand how to implement a python program to determine the largest among three numbers.

---

Figure 2-10. Self evaluation: Exercise 7

Self evaluation exercise 7 is as stated above.



IBM ICE (Innovation Centre for Education)

## Self evaluation: Exercise 8

- **Exercise 8:** Factorial calculation with a while loop.
- **Estimated time:** 00:15 minutes.
- **Aim:** Create a python program that calculates and prints the factorial of a number entered by the user using a while loop.
- **Learning objective:**
  - The learner will understand how to implement a python program to calculate the factorial of a number using a while loop.

---

Figure 2-11. Self evaluation: Exercise 8

Self evaluation exercise 8 is as stated above.



IBM ICE (Innovation Centre for Education)

## Self evaluation: Exercise 9

- **Exercise 9:** Fibonacci sequence with for loop.
- **Estimated time:** 00:15 minutes.
- **Aim:** Use a for loop to generate and print the fibonacci sequence up to a specified number of terms. Prompt the user for the number of terms.
- **Learning objective:**
  - The learner will understand how to implement a python program to generate the fibonacci sequence using a for loop.

---

Figure 2-12. Self evaluation: Exercise 9

Self evaluation exercise 9 is as stated above.



IBM ICE (Innovation Centre for Education)

## Self evaluation: Exercise 10

- **Exercise 10:** Sum of prime numbers in a given range.
- **Estimated time:** 00:15 minutes.
- **Aim:** Write a python program that calculates and prints the sum of all prime numbers in a given range. Prompt the user for the range.
- **Learning objective:**
  - The learner will understand how to implement a python program to calculate the sum of prime numbers in a specified range.

---

Figure 2-13. Self evaluation: Exercise 10

Self evaluation exercise 10 is as stated above.

## Checkpoint (1 of 2)



IBM ICE (Innovation Centre for Education)

### Multiple choice questions:

1. Predict the output of the code snippet given below:

```
> Sample_string = "Python is easy"  
> For l in sample_string:  
    print(l, end=" ", )
```

- a) Py.t,h,o,n, .i,s, ,e,a,s,y
- b) Py,t,h,o,n, ,i,s, , e,a,s,y,
- c) Syntax error
- d) Python, is, easy,

2. Conditions in Python are expressions that evaluate to either \_\_\_\_\_ or \_\_\_\_\_.

- a) True or False
- b) NOT or OR
- c) None of the above
- d) All of the above

3. The equivalent functionality can be achieved using\_\_\_\_\_ statements.

- a) if-elif-else
- b) If else
- c) While if
- d) None of the above

Figure 2-14. Checkpoint (1 of 2)

Write your answers here:

- 1.
- 2.
- 3

## Checkpoint (2 of 2)



IBM ICE (Innovation Centre for Education)

### Fill in the blanks:

1. The range() function in Python is commonly used in \_\_\_\_\_ loops for controlled iteration.
2. The \_\_\_\_\_ statement in Python provides an alternative to using multiple if-elif-else statements for handling different cases.
3. A \_\_\_\_\_ is an unordered, immutable collection of unique elements in Python.
4. A \_\_\_\_\_ is an ordered, immutable collection in Python.

### True/False:

1. Once a tuple is created, elements cannot be added or removed from it in Python. True/False
2. The sort() method modifies the original list and returns a new sorted list in Python. True/False
3. The sorted() function can be used with sets and tuples to return a sorted list. True/False

---

Figure 2-15. Checkpoint (2 of 2)

Write your answers here:

Fill in the blanks:

- 1.
- 2.
- 3.
- 4.

True/False:

- 1.
- 2.
- 3.

# Question bank (1 of 2)



IBM ICE (Innovation Centre for Education)

## Two-mark questions:

1. Display all even numbers between 50 and 80 (both inclusive) using "for" loop
2. Add natural numbers up to n where n is taken as an input from user. Print the sum
3. Assume, tupleA = (46,43,78,32,45,33,65,89)

Expected output is:

46

78

45

65

Write the code snippet to print the expected output.

4. What are Loop control statements ?

## Four-mark questions :

1. Consider the scenario of retail store management. The store provides discount for all bill amounts based on the criteria below:

Note: Assume that bill amount will be always greater than zero.

Bill Amount	Discount %
$\geq 1000$	5
$\geq 500 \text{ and } < 1000$	2
$> 0 \text{ and } < 500$	1

Write a Python program to find the net bill amount after discount. Observe the output with different values of bill amount.

Figure 2-16. Question bank (1 of 2)



IBM ICE (Innovation Centre for Education)

## Question bank (2 of 2)

2. Extend the above program to validate the customer id. Customer ids in the range of 101 and 1000 (both inclusive) should only be considered valid.

**Note:** Display appropriate error messages wherever applicable.

3. Prompt the user to enter a number. Print whether the number is prime or not
4. Print Fibonacci series till nth term where n is taken as an input from user.

**Hint:** Fibonacci series is a series of numbers in which each number is the sum of the two preceding numbers. Series start from 1 and goes like : 1, 1, 2, 3, 5, 8, 13 ....

### Eight-mark questions:

1. Accept two strings 'string1' and 'string2' as an input from the user. Generate a resultant string, such that it is a concatenated string of all upper case alphabets from both the strings in the order they appear. Print the actual and the resultant strings.

**Note:** Each character should be checked if it is a upper case alphabet and then it should be concatenated to the resultant string.

**Sample Input:**

string1: I Like C

string2: Mary Likes Python

**Output:** ILCMLP

2. Given a string containing both upper and lower case alphabets. Write a Python program to count the number of occurrences of each alphabet(case insensitive) and display the same.

**Sample Input:** ABaBCbGc

**Sample Output:** 2A3B 2C1G

Figure 2-17. Question bank (2 of 2)

## Unit summary



IBM ICE (Innovation Centre for Education)

- **Having completed this unit, you should be able to:**

- Implement control statements and loops to make decisions and perform iterative tasks
- Implement Python basics and effectively work with variables, data types, and collections

---

Figure 2-18. Unit summary

Unit summary is as stated above.

---

# Unit 3. Object-Oriented Programming (OOP) in Python

## Overview

This unit will provide an overview of Explore object-oriented programming (OOP) concepts and apply them to create reusable and modular code.

## How you will check your progress

- Checkpoint

## References

IBM Knowledge Center

# Unit objectives



IBM ICE (Innovation Centre for Education)

- After completing this unit, you should be able to:
  - Explore object-oriented programming (OOP) concepts and apply them to create reusable and modular code
- Learning outcomes:
  - Develop object-oriented Python code, encapsulating data and behavior in classes and objects

---

Figure 3-1. Unit objectives

Unit objectives and outcomes are as stated above.

# Advantages of Object-Oriented Programming (OOP)



IBM ICE (Innovation Centre for Education)

- Real-world modeling.
- Abstraction and encapsulation.
- Inheritance.
- Polymorphism.
- Code efficiency and development time.

Figure 3-2. Advantages of Object-Oriented Programming (OOP)

The exploration of Object-Oriented Programming (OOP) principles is initiated, providing readers with a comprehensive understanding of how classes and objects are formed, serving as the foundational elements in this paradigm. The intricacies of constructors, destructors, attributes, and methods are unfolded, empowering developers to create robust and flexible software systems. The advanced aspects of OOP, including inheritance, polymorphism, encapsulation, and abstraction, are explored, collectively shaping the way code is designed and structured.

- **Real-world modeling:** OOP facilitates the implementation of real-world entities as objects, making it easier to represent and manipulate complex systems. For example, in a banking system, objects such as accounts, transactions, and customers can be modeled as objects, each with their own attributes and behaviors.
- **Abstraction and encapsulation:** OOP promotes abstraction by hiding the internal implementation details of objects and exposing only the necessary interfaces. This enhances code modularity and reusability, as changes to one part of the codebase do not affect other parts. For instance, a car object may have methods for starting, stopping, and accelerating, while hiding the intricate mechanisms under the hood.
- **Inheritance:** OOP allows for the creation of hierarchical relationships between classes through inheritance, where subclasses inherit properties and behaviors from their parent classes. This promotes code reuse and modularity by reducing redundancy and promoting a hierarchical structure. For example, a superclass "Vehicle" can have subclasses "Car" and "Truck", inheriting common attributes and methods such as "move" and "stop".

- **Polymorphism:** OOP supports polymorphism, allowing objects of different classes to be treated interchangeably if they share a common interface. This promotes flexibility and extensibility, as new classes can be added without modifying existing code. For example, in a drawing application, shapes such as circles and rectangles can be treated uniformly through a common "draw" method, enabling easy addition of new shapes.
- **Code efficiency and development time:** By promoting code reuse, encapsulation, and abstraction, OOP helps reduce the overall development time and increase code efficiency. Developers can leverage existing classes and libraries, focus on high-level design, and write cleaner, more maintainable code. This results in faster development cycles and fewer bugs.



IBM ICE (Innovation Centre for Education)

# OOP principles

- Object-Oriented Programming (OOP) is a programming paradigm that focuses on organizing code into objects, which are instances of classes.
- Python supports OOP principles, and here are some key OOP concepts in Python:
  - [Classes and objects](#).
  - [Encapsulation](#).
  - [Inheritance](#).
  - [Polymorphism](#).
  - [Abstraction](#).
  - [Methods](#).
  - [Attributes](#).
  - [Constructor](#).
  - [Destructor](#).
  - [Access control](#).

*Figure 3-3. OOP principles*

- **Classes and objects:** In Python, a class is a blueprint for creating objects. Objects are instances of classes and encapsulate data (attributes) and behavior (methods).
- **Encapsulation:** Encapsulation is the concept of bundling data (attributes) and methods (functions) that operate on that data into a single unit, which is the class. It hides the internal state and provides controlled access to data through methods.
- **Inheritance:** Inheritance is a mechanism that allows a class to inherit attributes and methods from another class. It promotes code reuse and the creation of a hierarchy of classes.
- **Polymorphism:** Polymorphism allows objects of different classes to be treated as objects of a common base class. It simplifies code and makes it more flexible. In Python, polymorphism is often achieved through method overriding and duck typing.
- **Abstraction:** Abstraction is the process of simplifying complex reality by modeling classes based on the essential properties and behaviors they share. It hides the irrelevant details and exposes only what is necessary.
- **Methods:** Methods are functions defined within a class and are used to perform operations on the class's attributes. They can be instance methods (operating on specific objects) or class methods (operating on the class itself).
- **Attributes:** Attributes are variables defined within a class and hold data associated with the class. They can be instance variables (unique to each object) or class variables (shared among all objects of the class).

- **Constructor:** The constructor method, `__init__`, is called when an object of the class is created. It initializes the object's attributes.
- **Destructor:** The destructor method, `__del__`, is called when an object is destroyed or goes out of scope. It can be used to perform cleanup operations.
- **Access control:** "In Python, the visibility and accessibility of attributes and methods can be controlled using naming conventions. Attributes and methods with a single leading underscore (`_attribute` or `_method`) are considered "protected," and those with double leading underscores (`__attribute` or `__method`) are "private."



IBM ICE (Innovation Centre for Education)

## Defining classes and objects

- OOP promotes code organization, modularity, and reusability.
- It is a fundamental programming paradigm in Python, and it allows developers to create more organized and maintainable code.
- In Python, classes are used to define a blueprint or a template for creating objects (instances).
- They encapsulate both data (attributes) and behaviors (methods) into a single unit.

*Figure 3-4. Defining classes and objects*

To create a class, use the `class` keyword, followed by the class name (usually in CamelCase), and a colon to start the class block. Class names conventionally start with an uppercase letter.

`__init__`:

Example-

In real life, when a car is manufactured, it goes through an initialization process where various components such as the engine, tires, and seats are installed. Similarly, in Python, the `__init__` method serves as an initializer or constructor for objects of a class. It is automatically called when a new instance of the class is created and is used to initialize the object's attributes.

Real-life example: Imagine we have a `Car` class, and each car object needs to be initialized with attributes like `make`, `model`, and `year`. We can use the `__init__` method to set these attributes during object creation.

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
    # Creating an instance of the Car class
    my_car = Car("Toyota", "Camry", 2022)
```

In this example, the `__init__` method sets the make, model, and year attributes of the `my_car` object to "Toyota", "Camry", and 2022, respectively.

### `__str__`:

In real life, when someone asks about the details of a car, they expect a meaningful description that includes information such as the make, model, and year of the car. Similarly, the `__str__` method in Python is used to define how an object should be represented as a string when it is converted to a string using the `str()` function or when printed.

Real-life example: Continuing with the Car class example, let's define a `__str__` method that returns a string containing the make, model, and year of the car.

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
    def __str__(self):
        return f"{self.year} {self.make} {self.model}"
# Creating an instance of the Car class
my_car = Car("Toyota", "Camry", 2022)
# Printing the car object
print(my_car) # Output: 2022 Toyota Camry
```

In this example, the `__str__` method defines how the `my_car` object should be represented as a string. When `print(my_car)` is called, it invokes the `__str__` method to obtain a string representation of the object, resulting in "2022 Toyota Camry" being printed.

### Creating objects

Objects in python are created by instantiating a class. The class is called as if it were a function, and a new instance of the class is returned, constituting the object. To create an object of a class, the class name is called as if it were a function, with any required parameters being passed to the constructor. Creating objects of a class in Python involves instantiating the class to create instances (objects) with specific attributes. Here's a basic example to illustrate the process:

- The car class is defined with an `init` method that initializes the make, model, and year attributes of the instances.
- Two objects (`car1` and `car2`) are created by calling the class as if it were a function. This process is known as instantiation.
- The attributes of each object are accessed using dot notation (`car1.make`, `car2.model`, etc.).
- The print statements display information about each car.
- Remember, the `init` method is a special method in Python classes that gets called when an object is created. It initializes the attributes of the object based on the values provided during instantiation.
- As many objects as needed can be created from a class, each with its own set of attributes. Objects, instances of a class, are representative of individual entities in the program .

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

```
# Create objects (instances) of the Car class
car1 = Car(make="Toyota", model="Camry", year=2022)
car2 = Car(make="Honda", model="Civic", year=2021)
# Access attributes of the objects
print(f"Car 1: {car1.make} {car1.model} ({car1.year})")
print(f"Car 2: {car2.make} {car2.model} ({car2.year})")
```



IBM ICE (Innovation Centre for Education)

## Constructor in python

- In Python, a constructor is a special method used for initializing the attributes of an object when an instance of a class is created.
- The constructor method, named `init`, is automatically called when an object is created from the class, allowing the initial state of the object to be set up.

*Figure 3-5. Constructor in python*

Example: In this example, the `init` method takes three parameters (`make`, `model`, and `year`) in addition to the `self` parameter, which refers to the instance of the class being created. Inside the method, the attributes `make`, `model`, and `year` are assigned values based on the parameters passed during object creation. The constructor method, named `init`, is automatically called when an object is created from the class, allowing the initial state of the object to be set up.

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
    # Creating an instance of the Car class
    my_car = Car(make="Toyota", model="Camry", year=2022)
    # Accessing attributes
    print(f"Make: {my_car.make}")
    print(f"Model: {my_car.model}")
    print(f"Year: {my_car.year}")
```

### Default values in the constructor:

Default values for parameters in the constructor can also be provided. In the below example, a default value of 30 is assigned to the age parameter. If an age is not explicitly provided during object creation, the default value is utilized. The Constructor, a fundamental aspect of object-oriented programming in Python, permits the initialization of object states upon their creation.

```
class person:

    def __init__(self, name, age=30):
        self.name = name
        self.age = age

    # Creating an instance of the Person class
    person1 = Person(name="Alice")

    # Accessing attributes
    print(f"Name: {person1.name}")
    print(f"Age: {person1.age}")
```

### Parameterized constructor in Python

In Python, a parameterized constructor is a special method within a class that takes parameters (arguments) during the object creation process. It is used to initialize the attributes of the class with specific values provided when an object is instantiated. Let's illustrate this with an example:

```
class car:

    def __init__(self, make, model, year):
        # Parameterized Constructor with attributes make model and year
        self.make = make
        self.model = model
        self.year = year
        self.is_engine_on = False

    def start_engine(self):
        print(f"The {self.year} {self.make} {self.model}'s engine is now running.")
        self.is_engine_on = True

    def stop_engine(self):
        print(f"The {self.year} {self.make} {self.model}'s engine is now turned off.")
        self.is_engine_on = False

    def display_info(self):
        print(f"Car Information: {self.year} {self.make} {self.model}")

    # Creating an instance of the Car class with a parameterized constructor
    my_car = Car(make="Toyota", model="Camry", year=2022)

    # Displaying car information before starting the engine
    my_car.display_info()

    # Starting the engine
    my_car.start_engine()

    # Displaying car information after starting the engine
```

```
my_car.display_info()  
# Stopping the engine  
my_car.stop_engine()  
# Displaying car information after stopping the engine  
my_car.display_info()
```

**In this example:**

- The car class has a parameterized constructor `__init__` that takes three parameters (make, model, and year). These parameters are used to initialize the attributes of the class.
- The class also has methods (`start_engine`, `stop_engine`, and `display_info`) to perform actions and display information about the car.
- An instance of the Car class (`my_car`) is created with specific values for make, model, and year during instantiation.
- The methods are then called to start and stop the engine, and the information about the car is displayed before and after these actions.
- This example demonstrates the use of a parameterized constructor to initialize the attributes of the Car class when an object is created, allowing for more flexibility and customization during object instantiation.



IBM ICE (Innovation Centre for Education)

## Destructor in python

- In Python, the destructor is defined using the `__del__` method in a class. The `__del__` method is automatically called when an object is about to be destroyed or garbage collected.
- However, it's important to note that the exact timing of when the destructor is called is not guaranteed due to the behavior of Python's garbage collection mechanism.

*Figure 3-6. Destructor in python*

Here's an example illustrating the use of a destructor:

```
class ExampleClass:
    def __init__(self, name):
        # Constructor
        self.name = name
        print(f"{self.name} is created.")
    def perform_operation(self):
        print(f"{self.name} is performing an operation.")
    def __del__(self):
        # Destructor
        print(f"{self.name} is being destroyed.")

# Creating an instance of ExampleClass
obj = ExampleClass(name="Object")
# Performing an operation
obj.perform_operation()
# Deleting the instance explicitly to trigger the Destructor
```

```
del obj
```

**In the above example:**

- The `__init__` method serves as the constructor, initializing the object when it is created.
- The `perform_operation` method is just a regular method to simulate some operations.
- The `__del__` method serves as the Destructor, and it is automatically called when the object is explicitly deleted (using `del obj` in this case) or when the object is no longer referenced and is being garbage collected.
- Keep in mind that relying on the `__del__` method for resource cleanup is not always recommended. For better resource management, consider using context managers (with statements) or other appropriate mechanisms depending on the specific use case.



# Types of methods

- In Python, methods are functions defined inside a class, and they operate on the class's attributes and data.
- There are various types of methods defined inside a class, each serving a specific purpose. Here are the main types of methods defined in Python classes:

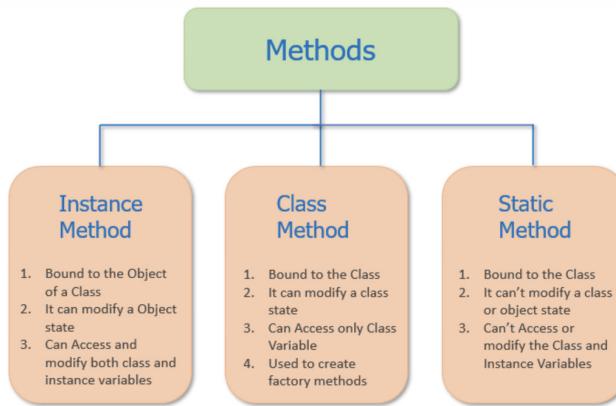


Figure 1: Method types

Source: <https://pynative.com/python-class-method-vs-static-method-vs-instance-method/>

Figure 3-7. Types of methods

## Instance methods

- Instance methods are the most common type of method in a class.
- They take the `self` parameter as their first argument, which refers to the instance (object) itself.
- These methods can access and modify the instance's attributes and perform operations specific to that instance.
- Instance methods can also access and call class methods and other instance methods.

### Example:

- The `Student` class has an `__init__` method to initialize the name, age, and grade attributes.
- The `display_info` method prints the student's name, age, and current grades.
- The `add_grade` method allows adding a new grade to the student's record.
- The `calculate_average` method calculates and prints the average grade of the student.
- An instance of the `Student` class (`student1`) is created with an initial set of grades.
- The student's information is displayed, a new grade is added, and the average grade is calculated.

```

class Student:

    def __init__(self, name, age, grade=[]):
        self.name = name
  
```

```
    self.age = age
    self.grade = grade
def display_info(self):
    print(f"Student Name: {self.name}")
    print(f"Age: {self.age}")
    print(f"Grades: {self.grade}")
def add_grade(self, new_grade):
    self.grade.append(new_grade)
    print(f"{self.name}'s grades: {self.grade}")
def calculate_average(self):
    if not self.grade:
        print(f"No grades available for {self.name}.")
        return None
    average = sum(self.grade) / len(self.grade)
    print(f"{self.name}'s average grade: {average}")
    return average
# Creating an instance of the Student class
student1 = Student(name="Alice", age=20, grade=[90, 85, 92])
# Displaying student information
student1.display_info()
# Adding a new grade
student1.add_grade(88)
# Calculating the average grade
student1.calculate_average()
```



IBM ICE (Innovation Centre for Education)

## Class methods

- Class methods in Python are methods that are bound to the class rather than the instance of the class.
- They are defined using the `@classmethod` decorator.
- Class methods have access to the class itself and can be used to perform actions related to the class rather than a specific instance.

*Figure 3-8. Class methods*

Here's an example with a practical scenario.

In this example:

- The `BankAccount` class has instance methods (`display_balance`, `deposit`, `withdraw`) for managing individual accounts.
- It also has class methods (`set_interest_rate`, `from_string`). `set_interest_rate` changes the class attribute `interest_rate` for all instances, and `from_string` is a class method that creates an instance from a string.
- An instance of `BankAccount` (`account1`) is created, and various operations are performed.
- The class method `set_interest_rate` is used to update the interest rate for all accounts.
- Another instance of `BankAccount` (`account2`) is created using the class method `from_string`.

Class `BankAccount`:

```
interest_rate = 0.03 # Class attribute
def __init__(self, account_holder, balance=0):
    self.account_holder = account_holder
    self.balance = balance
def display_balance(self):
    print(f"Balance for {self.account_holder}: ${self.balance}")
```

```

def deposit(self, amount):
    self.balance += amount
    print(f"Deposited ${amount}. New balance: ${self.balance}")
def withdraw(self, amount):
    if amount > self.balance:
        print("Insufficient funds!")
    else:
        self.balance -= amount
        print(f"Withdrew ${amount}. New balance: ${self.balance}")
@classmethod
def set_interest_rate(cls, new_rate):
    cls.interest_rate = new_rate
    print(f"New interest rate set to {new_rate}%")
@classmethod
def from_string(cls, account_info):
    account_holder, balance_str = account_info.split(',')
    balance = float(balance_str)
    return cls(account_holder, balance)
# Creating an instance of the BankAccount class
account1 = BankAccount(account_holder="Alice", balance=1000)
# Displaying balance
account1.display_balance()
# Depositing and withdrawing
account1.deposit(500)
account1.withdraw(200)
# Setting a new interest rate using a class method
BankAccount.set_interest_rate(0.04)
# Creating an account from a string using a class method
account_info = "Bob,1500"
account2 = BankAccount.from_string(account_info)
# Displaying balance for the new account
account2.display_balance()

```



## Static methods

	instance method	class method	static method
decorator	no decorator needed	@classmethod	@staticmethod
bound	Bound to the object.	Bound to the class.	Bound to the class.
example	<code>michael_jackson.do_moon dance()</code>	<code>michael_jackson.sing('T hriller')</code>	<code>michael_jackson.walk()</code>
parameter	takes <code>self</code> as parameter, the instance itself that calls the method.	takes <code>cls</code> as parameter, the class itself.	no parameter is taken as the instance or class, performs in isolation.
access	Access attributes and methods on the instance and its class, so we can modify the state of <b>both</b> objects. You have access to make a change in the state of the instance itself, but also its class and every other instance of the same class.	Can only modify the class state, being able to access attributes and methods of the class. So a change in state applies to all classes.	As no parameter of the object or class is available, it can't change a state.
Use example	Personalization of the object itself.	Factory methods, returns an object of the class.	Independency, accidental modifications.

Figure 2: Method types

Source: <https://ixcheldelsun.hashnode.dev/instance-method-vs-class-method-vs-static-method>

### Figure 3-9. Static methods

Defined with `@staticmethod` decorator. Static methods do not have access to either instance or class attributes (no `self` or `cls` parameter). Cannot modify instance or class attributes directly. Used for utility functions that do not depend on instance or class state. Independent of the instance or class. Unlike instance methods and class methods, we do not need to pass any special or default parameters.

Example:

A real-life example of a static method can be seen in a utility class that provides various helper functions or calculations, independent of any specific instance of the class. Let's consider a Math utility class with a static method `calculate_area` to calculate the area of different geometric shapes.

```
class Math:
    @staticmethod
    def calculate_area(shape, *args):
        if shape == "rectangle":
            length, width = args
            return length * width
        elif shape == "circle":
            radius, = args
            return 3.14 * (radius ** 2)
```

```

        elif shape == "triangle":
            base, height = args
            return 0.5 * base * height
        else:
            raise ValueError("Unsupported shape")

# Using the static method to calculate areas
rectangle_area = Math.calculate_area("rectangle", 5, 3)
print("Area of rectangle:", rectangle_area)
circle_area = Math.calculate_area("circle", 4)
print("Area of circle:", circle_area)
triangle_area = Math.calculate_area("triangle", 6, 8)
print("Area of triangle:", triangle_area)

```

In this example, the Math class contains a static method calculate\_area that calculates the area of different geometric shapes such as rectangles, circles, and triangles. The method is decorated with `@staticmethod` to indicate that it does not depend on any specific instance of the class.

We can use the `Math.calculate_area()` method directly without creating an instance of the Math class. It takes the shape as the first argument and additional arguments depending on the shape, calculates the area, and returns the result. This demonstrates how static methods can be used to encapsulate utility functions that are logically related to a class but do not require access to instance attributes or methods.

### Special methods

Special methods in Python, also known as magic methods or dunder methods (double underscore methods), provide a way to define how objects behave in certain situations. These methods are surrounded by double underscores, such as `__init__`, `__str__`, `__add__`, etc. Let's look at some practical examples to understand the use of a few common special methods.

`__init__`: Initialization method: In this example, `__init__` is used to initialize the attributes of the Point class when an object is created.

```

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

# Creating an instance
p = Point(x=3, y=5)

__str__ - String Representation Method: Here, __str__ is used to define a
human-readable string representation of the object.

```

```

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __str__(self):
        return f"Point(x={self.x}, y={self.y})"

# Creating an instance
p = Point(x=3, y=5)

# Printing the object

```

```
print(p)  # Output: Point(x=3, y=5)
Type of variables
__add__ - Addition Method: In this example, __add__ is used to define how two
objects of the Point class should be added.

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

# Creating two instances
p1 = Point(x=3, y=5)
p2 = Point(x=1, y=2)
# Adding two objects
result = p1 + p2
print(result.x, result.y)
# Output: 4 7
```

These are just a few examples, and many more special methods are available. Special methods enable the customization of the behavior of objects, making them more versatile and intuitive in various contexts.



# Type of variables

- In Python, variable types are distinguished into instance variables and class variables.
- These variables serve different purposes and are associated with either individual instances of a class or the class itself.
- Here is a detailed explanation of instance and class variables.

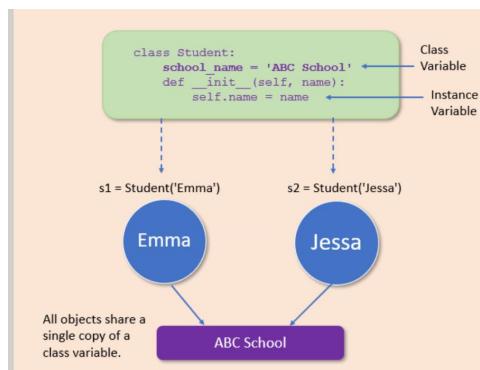


Figure 3: Class variable vs instance variable

Source: <https://pynative.com/python-class-variables/>

Figure 3-10. Type of variables

## Instance variables (Object Attributes)

An instance variable is a variable that is specific to an instance of a class. Each object (instance) of the class can have different values for its instance variables:

- Instance variables are declared within the constructor method (`__init__`) of a class using the `self` keyword.
- Instance variables are associated with individual objects (instances) of the class.
- They are defined within methods, typically inside the Constructor (`__init__`) using the `self` keyword.
- Each object of the class can have its unique values for these attributes.
- Instance variables are accessed and modified using the object's name.
- Modify value of an Instance Variable:** The value of the instance variables can be changed by adding a new value to it by using the object reference. It is important to note that the changes made in one object does not reflect in any of the other objects because the instance variable is maintained separately by each object.

Create instance variable using:

- within class---create and initialize using Constructor.
- within class---Inside instance method by using `self`.
- outside class---using object reference.

Access instance variable using

- within class---by using self.
- outside class-- using obj ref variable.

Delete instance variable using

- within class---del self.variable name.
- outside class-- del ref name.

### **Example code**

In the below example, each Car object (car1 and car2) has its own instance variables make, model, year, and color, which are initialized using the values provided during object creation. These instance variables represent unique characteristics of each car instance and are accessed and displayed using the `__str__` method.

```
class Car:
    def __init__(self, make, model, year, color):
        self.make = make
        self.model = model
        self.year = year
        self.color = color
    def __str__(self):
        return f"Make: {self.make}, Model: {self.model}, Year: {self.year}, Color: {self.color}"
# Creating instances of the Car class
car1 = Car("Toyota", "Camry", 2022, "Red")
car2 = Car("Honda", "Civic", 2021, "Blue")
# Printing each car object (which invokes the __str__ method)
print(car1)
print(car2)
```

### **Class variables (Class Attributes)**

A class variable is a variable that is shared among all instances of a class. It is defined at the class level and remains the same for all objects created from that class.

- All instances of the class share class variables.
- They are defined within the class but outside any method, usually at the class level.
- Class variables are accessed using the class name or an instance of the class.
- They are typically used for data that is common to all objects of the class and for better memory management.

Create static variable using:

- within class---inside cons using class name.
- within class---inside ins methods using class name.
- within class---inside class methods using class name or cls variable.
- within class---inside static methods using class name.
- within class---outside all methods.
- outside class-- using class name.

Access static variables using:

- within class---by using cls, class name and self.
- outside class-- using a class name or ref variable.

Modify static variables using:

- within class---by using cls, class name.
- outside class-- using class name.

Delete instance variable using:

- within class---del self.variable name.
- outside class-- del ref name.

**Example:** The code has below code had been modified to demonstrate the usage of class variables and the `__str__` method. In the Car class, a class variable `total_cars` is initialized to 0. Within the `__init__` method, `total_cars` is incremented by 1 each time a new Car object is created. Additionally, the `__str__` method has been implemented to return a string representation of the Car object's state. When each car object and the total number of cars are printed, Python automatically calls the `__str__` method for each car object to obtain its string representation. Finally, the total number of cars is printed using the class variable `total_cars`.

```
class Car:
    # Class variable
    total_cars = 0

    def __init__(self, make, model, year, color):
        self.make = make
        self.model = model
        self.year = year
        self.color = color

        # Incrementing the class variable total_cars
        Car.total_cars += 1

    def __str__(self):
        return f"Make: {self.make}, Model: {self.model}, Year: {self.year}, Color: {self.color}"

# Creating instances of the Car class
car1 = Car("Toyota", "Camry", 2022, "Red")
car2 = Car("Honda", "Civic", 2021, "Blue")

# Printing each car object and the total number of cars
print(car1)
print(car2)

print("Total number of cars:", Car.total_cars)
```

Instance Variable	Class Variable
Instance variables are not shared by objects. Every object has its own copy of the instance attribute.	Class variables are shared by all instances.
Instance variables are declared inside the constructor i.e., the <code>__init__()</code> method.	Class variables are declared inside the class definition but outside any of the instance methods and constructors.
It gets created when an instance of the class is created.	It is created when the program begins to execute.
Changes made to these variables through one object will not reflect in another object.	Changes made in the class variable will reflect in all objects.

Table 1: Instance vs class variables in python

[<https://pynative.com/python-class-variables/>]

Feature	Instance Variables	Static Variables	Local Variables
Scope	Object instances	Entire class	Function/block
Lifetime	Object duration	Program duration	Function/block execution
Initial value	None or assigned at object creation	None or assigned at class load	None or assigned within function/block
Modification	By methods within object	By any class method	Within function/block definition
Sharing	Individual copies per object	Single copy for all instances	Unique to each function/block execution
Impact	Define and track individual object state	Maintain class-level data	Provide temporary storage and control within functions/blocks

Figure 4: Variable Types

### Local variables

Local variables in Python are variables that are defined within a function and are only accessible within that function's scope. Here's a practical example demonstrating the use of local variables: In the given example - we use class variables for attributes shared among all employees, instance variables for attributes unique to each employee, and local variables within methods for temporary calculations or storage.

```
class Employee:
    # Class variable
    company_name = "ABC Corporation"
    def __init__(self, name, salary):
        # Instance variables
```

```
    self.name = name
    self.salary = salary
def calculate_bonus(self, performance_rating):
    # Local variable for bonus calculation
    bonus_percentage = 0.1
    bonus = self.salary * bonus_percentage * performance_rating
    return bonus
# Creating instances of the Employee class
employee1 = Employee(name="Alice", salary=50000)
employee2 = Employee(name="Bob", salary=60000)
# Calling a method with a local variable for bonus calculation
performance_rating1 = 1.2
bonus1 = employee1.calculate_bonus(performance_rating1)
print("\nBonus for Employee 1:", bonus1)
```



# Advanced OOP concepts

- Inheritance and Polymorphism.
- Inheritance is a fundamental concept in Object-Oriented Programming (OOP) that allows a new class to inherit attributes and methods from an existing class.
- In Python, a new class that is a modified version of an existing class can be created by using inheritance. The existing class is referred to as the "base class" or "parent class," and the new class is called the "derived class" or "child class."

Feature	Instance Variables	Static Variables	Local Variables
Scope	Object instances	Entire class	Function/block
Lifetime	Object duration	Program duration	Function/block execution
Initial value	None or assigned at object creation	None or assigned at class load	None or assigned within function/block
Modification	By methods within object	By any class method	Within function/block definition
Sharing	Individual copies per object	Single copy for all instances	Unique to each function/block execution
Impact	Define and track individual object state	Maintain class-level data	Provide temporary storage and control within functions/blocks

Figure 5: Variable types

Figure 3-11. Advanced OOP concepts

Let's consider an example involving a base class, **Shape** and two derived classes, **Circle** and **Rectangle**. This example demonstrates how inheritance allows us to reuse code and create specialized classes. In this example:

- Shape is the base class with a constructor that takes the color of the Shape, a method area (marked as abstract), and a method display\_info to display information about the Shape.
- Circle and Rectangle are derived classes that inherit from Shape. They each have their own Constructor, implement the area method specific to their Shape, and may have additional attributes.
- The display\_info method in the Shape class uses self.class\_name\_\_ to dynamically get the name of the class, making it easy to display the type of Shape.
- This example demonstrates how inheritance enables the creation of a common base class with shared functionality, followed by the construction of specialized classes that inherit and extend that functionality. It also illustrates the concept of polymorphism, where different shapes can be treated as instances of the common base class.

```
import math
class Shape:
    def __init__(self, color):
        self.color = color
```

```
def area(self):
    raise NotImplementedError("Subclasses must implement the area method.")
def display_info(self):
    print(f"Shape: {self.__class__.__name__}, Color: {self.color}")
class Circle(Shape):
    def __init__(self, color, radius):
        super().__init__(color)
        self.radius = radius
    def area(self):
        return math.pi * self.radius**2
class Rectangle(Shape):
    def __init__(self, color, length, width):
        super().__init__(color)
        self.length = length
        self.width = width
    def area(self):
        return self.length * self.width
# Creating instances of the classes
circle = Circle(color="Red", radius=5)
rectangle = Rectangle(color="Blue", length=4, width=6)
# Displaying information and calculating areas
circle.display_info()
print("Area:", circle.area())
rectangle.display_info()
print("Area:", rectangle.area())
```



# Inheritance types

- Single Inheritance:
  - A class inherits from only one base class.
  - It's a simple form of inheritance.

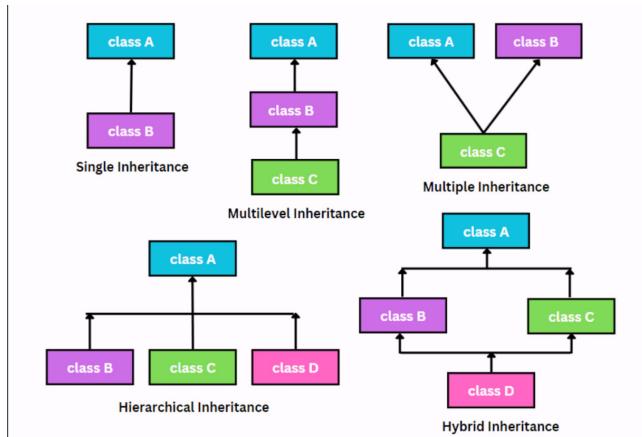


Figure 6: Inheritance types in python

Source: [https://www.scientecheeasy.com/2023/09/types-of-inheritance-in-python.html#google\\_vignette](https://www.scientecheasy.com/2023/09/types-of-inheritance-in-python.html#google_vignette)

Figure 3-12. Inheritance types

### In this example:

The vehicle is the base class with a constructor (`__init__`) that initializes the brand and model attributes and a method (`display_info`) to display information about the Vehicle. Car is the derived class that inherits from Vehicle. It has a Constructor that extends the functionality of the base class constructor by adding a color attribute, and it also has a method (`display_car_info`) to display information about the car.

```
class Vehicle:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
    def display_info(self):
        print(f"Brand: {self.brand}, Model: {self.model}")

class Car(Vehicle):
    def __init__(self, brand, model, color):
        # Calling the Constructor of the base class
        super().__init__(brand, model)
        self.color = color
```

```

def display_car_info(self):
    print(f"Color: {self.color}")

# Creating an instance of the car
my_car = Car(brand="Toyota", model="Camry", color="Blue")
# Accessing methods from the base class (Vehicle)
my_car.display_info()
# Accessing methods from the derived class (Car)
my_car.display_car_info()

```

### Key points

- The `super().__init__(brand, model)` line in the `Car` class constructor calls the Constructor of the base class (`Vehicle`). This ensures that the common attributes are initialized.
- Instances of the `Car` class can access methods and attributes from both the base class (`Vehicle`) and the derived class (`Car`).
- Single inheritance is illustrated here because a car inherits from only one base class (`Vehicle`).
- When this code is executed, it will be observed that the `Car` class inherits the `brand` and `model` attributes, as well as the `display_info` method from the `Vehicle` class, thereby illustrating the concept of single inheritance.

### Multiple inheritance

Multiple inheritance involves a class inheriting from more than one base class. In this example, we'll introduce an `Engine` class, and a `HybridCar` class will inherit from both `Vehicle` and `Engine`.

#### **class Vehicle:**

```

def __init__(self, brand, model):
    self.brand = brand
    self.model = model
def display_info(self):
    print(f"Brand: {self.brand}, Model: {self.model}")

```

#### **class Car(Vehicle):**

```

def __init__(self, brand, model, color):
    # Calling the Constructor of the base class
    super().__init__(brand, model)
    self.color = color
def display_car_info(self):
    print(f"Color: {self.color}")

```

#### **class Engine:**

```

def __init__(self, fuel_type):
    self.fuel_type = fuel_type
def display_engine_info(self):
    print(f"Fuel Type: {self.fuel_type}")

```

#### **class HybridCar(Car, Engine):**

```

def __init__(self, brand, model, color, fuel_type):
    # Calling constructors of both base classes

```

```

        Car.__init__(self, brand, model, color)
        Engine.__init__(self, fuel_type)

# Creating an instance of HybridCar
my_hybrid_car = HybridCar(brand="Toyota", model="Prius", color="Silver",
fuel_type="Electric + Gasoline")

# Accessing methods from both base classes (Car and Engine)
my_hybrid_car.display_info()
my_hybrid_car.display_car_info()
my_hybrid_car.display_engine_info()

```

### Multilevel inheritance

Multilevel inheritance involves a chain of inheritance. In this example, we'll introduce a LuxuryCar class that inherits from the Car class, creating a multilevel inheritance chain.

#### **class Vehicle:**

```

def __init__(self, brand, model):
    self.brand = brand
    self.model = model

def display_info(self):
    print(f"Brand: {self.brand}, Model: {self.model}")

```

#### **class Car(Vehicle):**

```

def __init__(self, brand, model, color):
    # Calling the Constructor of the base class
    super().__init__(brand, model)
    self.color = color

def display_car_info(self):
    print(f"Color: {self.color}")

```

#### **class LuxuryCar(Car):**

```

def __init__(self, brand, model, color, luxury_feature):
    # Calling the Constructor of the immediate base class (Car)
    super().__init__(brand, model, color)
    self.luxury_feature = luxury_feature

def display_luxury_info(self):
    print(f"Luxury Feature: {self.luxury_feature}")

```

#### # Creating an instance of LuxuryCar

```

my_luxury_car = LuxuryCar(brand="Mercedes", model="S-Class", color="Black",
luxury_feature="Massaging Seats")

# Accessing methods from both base classes (Car and LuxuryCar)
my_luxury_car.display_info()
my_luxury_car.display_car_info()
my_luxury_car.display_luxury_info()

```

### Hierarchical inheritance

Hierarchical inheritance involves multiple classes inheriting from a single base class. In this example, we'll introduce a Truck class that inherits from the common base class Vehicle.

### **class Vehicle:**

```
def __init__(self, brand, model):
    self.brand = brand
    self.model = model
def display_info(self):
    print(f"Brand: {self.brand}, Model: {self.model}")
```

### **class Car(Vehicle):**

```
def __init__(self, brand, model, color):
    # Calling the Constructor of the base class
    super().__init__(brand, model)
    self.color = color
def display_car_info(self):
    print(f"Color: {self.color}")
```

### **class Truck(Vehicle):**

```
def __init__(self, brand, model, cargo_capacity):
    # Calling the Constructor of the base class (Vehicle)
    super().__init__(brand, model)
    self.cargo_capacity = cargo_capacity
def display_truck_info(self):
    print(f"Cargo Capacity: {self.cargo_capacity}")
# Creating an instance of a Truck
my_truck = Truck(brand="Ford", model="F-150", cargo_capacity="2000 lbs")
# Accessing methods from the base class (Vehicle) and derived class (Truck)
my_truck.display_info()
my_truck.display_truck_info()
```

### **Hybrid inheritance**

Hybrid inheritance is a combination of two or more types of inheritance within a single program. It often involves a mix of single inheritance, multiple inheritance, and multilevel inheritance. In Python, where multiple inheritance is supported, scenarios demonstrating hybrid inheritance can be created. Let's extend our previous example to create a scenario that involves hybrid inheritance.

### **class Vehicle:**

```
def __init__(self, brand, model):
    self.brand = brand
    self.model = model
def display_info(self):
    print(f"Brand: {self.brand}, Model: {self.model}")
```

### **class Car(Vehicle):**

```
def __init__(self, brand, model, color):
    # Calling the Constructor of the base class
```

```

        super().__init__(brand, model)
        self.color = color
    def display_car_info(self):
        print(f"Color: {self.color}")
class Engine:
    def __init__(self, fuel_type):
        self.fuel_type = fuel_type
    def display_engine_info(self):
        print(f"Fuel Type: {self.fuel_type}")
class HybridCar(Car, Engine):
    def __init__(self, brand, model, color, fuel_type):
        # Calling constructors of both base classes
        Car.__init__(self, brand, model, color)
        Engine.__init__(self, fuel_type)
class ElectricEngine:
    def __init__(self, battery_capacity):
        self.battery_capacity = battery_capacity
    def display_electric_engine_info(self):
        print(f"Battery Capacity: {self.battery_capacity} kWh")
class HybridLuxuryCar(HybridCar, ElectricEngine):
    def __init__(self, brand, model, color, fuel_type, battery_capacity,
luxury_feature):
        # Calling constructors of multiple base classes
        HybridCar.__init__(self, brand, model, color, fuel_type)
        ElectricEngine.__init__(self, battery_capacity)
        self.luxury_feature = luxury_feature
    def display_hybrid_luxury_info(self):
        print(f"Luxury Feature: {self.luxury_feature}")
# Creating an instance of HybridLuxuryCar
my_hybrid_luxury_car = HybridLuxuryCar(
    brand="Lexus", model="RX450h", color="White",
    fuel_type="Electric" + " Gasoline", battery_capacity="300",
luxury_feature="Leather Seats"
)
# Accessing methods from multiple base classes (HybridCar and ElectricEngine)
my_hybrid_luxury_car.display_info()
my_hybrid_luxury_car.display_car_info()
my_hybrid_luxury_car.display_engine_info()
my_hybrid_luxury_car.display_hybrid_luxury_info()

```



IBM ICE (Innovation Centre for Education)

# Polymorphism

- Polymorphism is the ability of a single function, method, or operator to operate on different types of data or to respond to different messages or inputs based on the context in which it is used.
- It allows objects of various types to be treated as objects of a common type, enabling code to be more versatile and adaptable.
- In essence, polymorphism allows the same operation or function to behave differently based on the types of entities involved, making code more generic and capable of handling a variety of scenarios without the need for explicit type-checking.
- This flexibility is achieved through features like method overloading, method overriding, and the use of abstract classes or interfaces.
- Polymorphism can be achieved through method overloading, method overriding, or using abstract classes and interfaces.

*Figure 3-13. Polymorphism*

### Examples of polymorphism:

**Function overloading:** Function overloading allows a single function name to perform different operations based on the number or type of parameters. This enables developers to create functions with the same name but different parameter lists.

```
class Calculator:
    def add(self, a, b):
        return a + b
    def add(self, a, b, c):
        return a + b + c
calc = Calculator()
print(calc.add(2, 3))      # Output: Error: add() missing 1 required positional argument: 'c'
print(calc.add(2, 3, 4))   # Output: 9
```

In this example, the `add()` method in the `Calculator` class is defined twice with different parameter lists. Depending on the number of arguments provided, Python automatically selects the appropriate version of the method to execute.

**Operator overloading:** Operator overloading enables operators to have different implementations based on the operands' types. This allows developers to define custom behavior for operators like +, -, \*, etc., when used with instances of user-defined classes.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)
p1 = Point(1, 2)
p2 = Point(3, 4)
result = p1 + p2
print(result.x, result.y) # Output: 4 6
```

Here, the `__add__()` method is overridden in the `Point` class to define the addition behavior for `Point` objects. When `p1 + p2` is evaluated, Python invokes the `__add__()` method on `p1`, passing `p2` as the other parameter, resulting in the addition of their coordinates.

- **Polymorphism in Built-in Function `len()`:** The `len()` function in Python is polymorphic because it can accept various types of objects, such as sequences (e.g., lists, tuples, strings), and return their lengths. This allows developers to use the same function name (`len()`) to perform different operations depending on the type of the object passed as an argument.

For example:

```
pythonCopy code
print(len([1, 2, 3])) # Output: 3
print(len("hello")) # Output: 5
print(len((1, 2, 3, 4))) # Output: 4
```

In each case, `len()` behaves differently based on the type of the object passed to it. It returns the number of elements in a list, the number of characters in a string, and the number of items in a tuple, respectively.

**Polymorphism With Inheritance:** Inheritance enables polymorphism by allowing objects of different subclasses to be treated as objects of the parent class. This facilitates code reuse and promotes flexibility in object-oriented programming.

For example:

```
pythonCopy code
class Animal:
    def speak(self):
        pass
class Dog(Animal):
    def speak(self):
        return "Woof!"
class Cat(Animal):
    def speak(self):
        return "Meow!"
def make_sound(animal):
    return animal.speak()
```

```

dog = Dog()
cat = Cat()

print(make_sound(dog))  # Output: Woof!
print(make_sound(cat))  # Output: Meow!

```

In this example, both the Dog and Cat classes inherit from the Animal class and override the speak() method. The make\_sound() function accepts an Animal object and calls its speak() method. Depending on the actual object passed (either Dog or Cat), different sounds are produced, demonstrating polymorphism in action.

**Method overloading:** Method overloading in Python allows a class to define multiple methods with the same name but with different parameters. The method that gets invoked is determined based on the number and types of arguments provided during the function call.

```

class MathOperations:

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

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

# Usage

math_ops = MathOperations()
result1 = math_ops.add(2, 3)           # Invokes the first add method
result2 = math_ops.add(2, 3, 4)         # Invokes the second add method

```

- **Method overriding:** Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. This allows objects of the subclass to be used in the same way as objects of the superclass:
  - Inheritance in Python allows a subclass to inherit methods from its superclass.
  - Method overriding occurs when a subclass provides its implementation for a method that is already defined in the superclass.
  - This enables customization of behavior in the subclass without modifying the original method in the superclass.

## Example

```

class Animal:

    def speak(self):
        return "Generic animal sound"

class Dog(Animal):

    def speak(self):
        return "Woof!"

class Cat(Animal):

    def speak(self):
        return "Meow!"

# Usage

dog = Dog()
cat = Cat()

print(dog.speak())    # Outputs: "Woof!"
print(cat.speak())   # Outputs: "Meow!"

```

Overriding methods with super()

Overriding methods using super() in Python is a mechanism that allows a subclass to provide a specific implementation for a method that is already defined in its superclass. The super() function is used to invoke the method from the parent class, enabling the extension of behavior rather than a complete replacement.

## Key concepts

### Using super()

- The super() function is used to call a method from the parent class (superclass).
- It is commonly used in the overridden method of the subclass to invoke the same method from the superclass.

Example:

```
class Parent:
    def show_info(self):
        print("Parent class method")
class Child(Parent):
    def show_info(self):
        super().show_info()  # Calling the method from the parent class
        print("Child class method")
```

### Method extension

- By using super(), a subclass can extend the behavior of the overridden method rather than completely replacing it.
- This becomes especially useful when the retention of the functionality of the parent class method is desired, along with the addition of specific behavior in the subclass.

### Constructor overriding example

A common use case is overriding the Constructor (`__init__`) to include additional initialization steps in the subclass.

```
class Vehicle:
    def __init__(self, brand):
        self.brand = brand
class Car(Vehicle):
    def __init__(self, brand, model):
        super().__init__(brand)  # Calling the constructor from the parent class
        self.model = model
```

Example

```
class Animal:
    def speak(self):
        print("Generic animal sound")
class Dog(Animal):
    def speak(self):
        super().speak()  # Calling the method from the parent class
        print("Woof!")
# Usage
dog = Dog()
```

```
dog.speak()
```

In this example, the Dog class overrides the speak method from the Animal class using super(). It first calls the speak method from the Animal class and then adds specific behavior for a dog. This ensures that the original functionality of the speak method in the Animal class is retained in the Dog class. Using super() in method overriding promotes code reusability and maintainability by allowing subclasses to build upon the existing functionality of their parent classes.



IBM ICE (Innovation Centre for Education)

## Polymorphism with abstract classes

- Abstract classes in Python can be used to achieve a form of polymorphism by defining abstract methods that must be implemented by concrete subclasses.
- This ensures that different classes adhere to a common interface.

Figure 3-14. Polymorphism with abstract classes

```
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**2

class Square(Shape):
    def __init__(self, side_length):
        self.side_length = side_length
    def area(self):
        return self.side_length**2

# Usage
```

```

circle = Circle(radius=5)
square = Square(side_length=4)
print(circle.area())    # Outputs: 78.5
print(square.area())    # Outputs: 16

```

Polymorphism enhances code flexibility and promotes the reuse of code by allowing objects of different types to be treated uniformly when they share a common interface.

### **Dynamic binding**

Dynamic binding in Python refers to the process where the actual method or function to be called is determined at runtime, based on the type of the object being referenced. This allows for flexibility and polymorphic behavior, as different objects can respond differently to the same method call.

Here's how dynamic binding works in Python:

- **Determination at Runtime:** In Python, method calls are resolved dynamically at runtime, based on the type of the object being referenced. When a method is called on an object, Python searches for the appropriate method implementation in the object's class hierarchy.
- **Polymorphic Behavior:** Dynamic binding enables polymorphism, where different objects can respond differently to the same method call. This allows for flexibility in designing and using classes, as objects of different types can share a common interface (e.g., method name), but provide different implementations.

Example:

```

pythonCopy code
class Animal:
    def speak(self):
        return "Animal speaks"
class Dog(Animal):
    def speak(self):
        return "Woof!"
class Cat(Animal):
    def speak(self):
        return "Meow!"
def make_sound(animal):
    return animal.speak()
# Dynamic binding in action
dog = Dog()
cat = Cat()
print(make_sound(dog))    # Output: Woof!
print(make_sound(cat))    # Output: Meow!

```

In this example, the `make_sound()` function accepts an `Animal` object and calls its `speak()` method. Depending on the actual object passed (either `Dog` or `Cat`), different sounds are produced. This demonstrates dynamic binding, as the method to be called is determined at runtime based on the type of the object being referenced.

Dynamic binding in Python contributes to the language's flexibility and support for polymorphism, making it easier to write expressive and maintainable code.



# Encapsulation and abstraction

- Access modifiers:
  - Public.
  - Private.
  - Protected.
- In Python, attributes and methods can have different levels of visibility or access control. While Python does not have keywords like public, protected, or private, as seen in some other languages, it uses naming conventions to indicate the visibility of attributes and methods.

Class member access specifier	Access from own class	Accessible from derived class	Accessible from object
Private member	Yes	No	No
Protected member	Yes	Yes	No
Public member	Yes	Yes	Yes

Figure 7: Access modifiers in python

Source: <https://www.scaler.com/topics/python/encapsulation-in-python/>

Figure 3-15. Encapsulation and abstraction

## The conventions are as follows:

**Public:** Attributes and methods that are intended to be part of the public API of a class should have names in lowercase with underscores separating words. For example, `public_attribute` or `public_method()`.

**Protected:** Attributes and methods intended for internal use within the class or its subclasses are typically named with a single leading underscore. For example, `_protected_attribute` or `_protected_method()`.

**Private:** Attributes and methods intended to be private to the class (not even accessible in subclasses) are typically named with a double-leading underscore. For example, `__private_attribute` or `__private_method()`.

It's important to note that the Python interpreter does not enforce these conventions. They are simply conventions agreed upon by the Python community.

Here's an example illustrating the conventions:

- name is a public attribute accessible from outside the class.
- `_age` is a protected attribute, suggesting that it should be treated as internal to the class or its subclasses.
- `__person_id` is a private attribute. Note that Python uses name mangling, so it's not directly accessible as `__person_id` from outside the class. It is accessible using the mangled name `_Person__person_id` within the class.

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

```

        self.name = name # Public attribute
        self._age = age # Protected attribute
        self.__person_id = person_id # Private attribute
    def display_info(self):
        # Accessing attributes within the class
        print(f"Name: {self.name}")
        print(f"Age: {self._age}")
        print(f"Person ID: {self.__person_id}")

# Creating an instance of a Person
person = Person(name="Alice", age=30, person_id="12345")
# Accessing public, protected, and private attributes using a public method
person.display_info()
# Accessing attributes directly from outside the class (not recommended)
print(person.name) # Public attribute
print(person._age) # Protected attribute (accessible but not recommended)
print(person.__Person__person_id) # Private attribute (using name mangling)
While Python allows this access, it's generally considered good practice to respect these conventions to promote code readability and maintainability.

```

## Encapsulation

Encapsulation is one of the fundamental principles of Object-oriented Programming (OOP) that involves bundling the data (attributes) and the methods (functions) that operate on the data into a single unit known as a class. It also involves restricting access to some of the object's components, making certain details of the implementation private.

In Python, encapsulation is implemented through the use of private and protected attributes, as well as methods. However, it's important to note that Python does not enforce strict encapsulation, as access control is based on conventions rather than strict language features.

In the below example:

- The attributes `_account_holder` and `__balance` are encapsulated within the `BankAccount` class.
- `_account_holder` is a protected attribute, and `__balance` is a private attribute. Conventionally, a single-leading underscore indicates that an attribute is intended to be protected (not part of the public API), and a double-leading underscore indicates that an attribute is intended to be private.
- Public methods (`deposit`, `withdraw`, `get_balance`, and `get_account_holder`) provide controlled access to the encapsulated attributes.
- While Python doesn't enforce strict encapsulation, the use of conventions helps developers understand which attributes and methods are intended to be part of the public API and which are considered internal to the class.

```

class BankAccount:
    def __init__(self, account_holder, balance):
        self._account_holder = account_holder # Protected attribute
        self.__balance = balance # Private attribute
    def deposit(self, amount):
        self.__balance += amount
    def withdraw(self, amount):

```

```

        if amount <= self.__balance:
            self.__balance -= amount
        else:
            print("Insufficient funds.")
    def get_balance(self):
        return self.__balance
    def get_account_holder(self):
        return self.__account_holder
# Creating an instance of BankAccount
account = BankAccount(account_holder="John Doe", balance=1000)
# Accessing public methods to interact with encapsulated data
print(f"Account Holder: {account.get_account_holder()}")
print(f"Initial Balance: {account.get_balance()}")
account.deposit(500)
print(f"Balance after deposit: {account.get_balance()}")
account.withdraw(200)
print(f"Balance after withdrawal: {account.get_balance()}")

```

## Abstraction

In Python, abstraction is a programming concept that allows the creation of abstract classes and methods. Abstraction involves hiding the complex implementation details of an object and exposing only the relevant functionalities or features. It helps in managing complexity by providing a simplified view of an object and focusing on what it does rather than how it achieves its functionality.

**Abstract classes:** Abstract classes in Python are classes that cannot be instantiated on their own. They may contain abstract methods, which are methods without a defined implementation.

An abstract class is created in Python, and its implementation for a real-life use case is provided. In this example, an abstract class `Vehicle` is defined with an abstract method `start_engine()`. Implementations for this abstract class are then provided for two specific types of vehicles: `Car` and `Motorcycle`.

```

from abc import ABC, abstractmethod

# The abstract class representing a generic Vehicle is initialized
class Vehicle(ABC):
    def __init__(self, make, model):
        self.make = make
        self.model = model
    @abstractmethod
    def start_engine(self):
        pass
# Concrete classes representing a Car and a Motorcycle are defined
class Car(Vehicle):
    def __init__(self, make, model, num_doors):
        super().__init__(make, model)
        self.num_doors = num_doors
    def start_engine(self):

```

```

        print(f"The {self.make} {self.model}'s engine is now running.")

    def drive(self):
        print(f"The {self.make} {self.model} is now moving.")

class Motorcycle(Vehicle):
    def __init__(self, make, model):
        super().__init__(make, model)

    def start_engine(self):
        print(f"The {self.make} {self.model}'s engine is now running.")

    def ride(self):
        print(f"The {self.make} {self.model} is now on the road.")

# Example usage
car_instance = Car(make="Toyota", model="Camry", num_doors=4)
car_instance.start_engine()
car_instance.drive()

motorcycle_instance = Motorcycle(make="Harley-Davidson", model="Sportster")
motorcycle_instance.start_engine()
motorcycle_instance.ride()

```

**In this example:**

- The abstract class Vehicle is instantiated with a constructor that takes the make and model of the Vehicle. Additionally, it declares an abstract method start\_engine().
- Concrete classes Car and Motorcycle are instantiated, both inheriting from the Vehicle abstract class. These classes provide implementations for the start\_engine() method.
- Both car and motorcycle have additional methods (drive() and ride() respectively) that represent behaviors specific to each type of Vehicle.
- This form of abstraction allows the definition of a common interface (start\_engine()) for various types of vehicles while permitting each concrete class to implement its specific behavior.

- This type of abstraction proves useful in scenarios where enforcing a common set of methods across different classes ensures a consistent interface.

<b>Abstraction</b>	<b>Encapsulation</b>
1. Abstraction solves the problem in the design level.	1. Encapsulation solves the problem in the implementation level.
2. Abstraction is used for hiding the unwanted data and giving relevant data.	2. Encapsulation means hiding the code and data into a single unit to protect the data from outside world.
3. Abstraction lets you focus on what the object does instead of how it does it	3. Encapsulation means hiding the internal details or mechanics of how an object does something.
<b>4. Abstraction-</b> Outer layout, used in terms of design. For Example:- Outer Look of a Mobile Phone, like it has a display screen and keypad buttons to dial a number.	<b>4. Encapsulation-</b> Inner layout, used in terms of implementation. For Example:- Inner Implementation detail of a Mobile Phone, how keypad button and Display Screen are connect with each other using circuits.

Table 2: Encapsulation vs abstraction [https://i.stack.imgur.com/07dXi.png]

## Achieving abstraction through interfaces



IBM ICE (Innovation Centre for Education)

- In Python, there is no explicit interface keyword as found in some other programming languages like Java or C#.
- Abstraction through interfaces in programming involves creating a common set of method signatures that multiple classes can implement.
- It allows for a unified way to interact with objects of different types while abstracting away the underlying details of their implementations.
- Achieving abstraction through interfaces promotes code modularity, flexibility, and maintainability.

*Figure 3-16. Achieving abstraction through interfaces*

**Interface definition:** An interface is a contract that specifies a set of methods without providing their implementations. In Python, interfaces are typically represented by classes with method signatures but no method bodies.

```
class ShapeInterface:
    def calculate_area(self):
        pass
    def display(self):
        pass
```

**Concrete implementations:** Concrete classes implement the methods defined in the interface. Each class provides its specific implementation for the methods.

**Example:**

```
class Circle(ShapeInterface):
    def __init__(self, radius):
        self.radius = radius
    def calculate_area(self):
        return 3.14 * self.radius**2
    def display(self):
```

```
print(f"Circle with radius {self.radius}")
```

### Benefits of Abstraction through Interfaces

- Flexibility: New classes can be added without modifying existing code as long as they adhere to the interface.
- Modularity: Code is organized around well-defined interfaces, making it easier to understand and maintain.
- Code reusability: Interfaces allow for the creation of reusable components that can be employed in various contexts.

### Real-life analogy

Think of an interface as a remote control. Regardless of the brand or model of a TV, as long as it has a standard remote control interface (buttons for power, volume, channels), users can interact with any TV without needing to know the internal workings of each TV. The remote control serves as an abstraction, providing a common way to operate different TVs.

In the same way, interfaces in programming provide a standardized way to interact with different classes, promoting a higher level of abstraction and simplifying code maintenance and extension.

### Summary

In conclusion, a journey has been taken through the core principles of Object-Oriented Programming in this chapter. From the basics of defining classes and objects to the finer details of constructors, destructors, and variable types, valuable insights have been gained into the building blocks of OOP. The effectiveness of OOP in crafting modular, scalable, and easily maintainable code has been demonstrated through advanced concepts like inheritance, polymorphism, encapsulation, and abstraction. With this knowledge, individuals are now well-prepared to apply OOP principles to create sleek and effective software solutions, mastering the craft of object-oriented design. The upcoming chapters will delve even deeper into these principles, offering a thorough guide to mastering OOP in Python.

## Self evaluation: Exercise 11



IBM ICE (Innovation Centre for Education)

- **Exercise 11:** Simple Calculator Class
- **Estimated time:** 00:15 minutes
- **Aim:** Define a Python class for a simple calculator that has methods for addition and subtraction. Allow the user to perform calculations using objects of this class
- **Learning objective:**
  - The learner will understand how to create and use a simple class in Python for basic arithmetic operations
- **Learning outcome:**
  - The learner will be able to define a class, create objects, and perform addition and subtraction operations using methods

---

Figure 3-17. Self evaluation: Exercise 11

Self evaluation exercise 11 is as stated above:

## Self evaluation: Exercise 12



IBM ICE (Innovation Centre for Education)

- **Exercise 12:** Class Inheritance Hierarchy
- **Estimated time:** 00:15 minutes
- **Aim:** Create a class hierarchy with a base class and two derived classes. Demonstrate inheritance by accessing attributes and methods of each class
- **Learning objective:**
  - The learner will understand the concept of class inheritance in Python and how to create a hierarchy of classes
- **Learning outcome:**
  - The learner will be able to define a base class, create derived classes, and demonstrate inheritance by accessing attributes and methods

---

Figure 3-18. Self evaluation: Exercise 12

Self evaluation exercise 12 is as stated above:

## Self evaluation: Exercise 13



IBM ICE (Innovation Centre for Education)

- **Exercise 13:** Method Overriding
- **Estimated time:** 00:15 minutes
- **Aim:** Implement method overriding in a Python class. Create a base class with a method and override it in a derived class.
- **Learning objective:**
  - The learner will understand the concept of method overriding in Python and how to implement it in a class hierarchy
- **Learning outcome:**
  - The learner will be able to define a base class with a method, create a derived class, and override the method to provide a specialized implementation

---

Figure 3-19. Self evaluation: Exercise 13

Self evaluation: Exercise 13 is as stated above:



IBM ICE (Innovation Centre for Education)

## Self evaluation: Exercise 14

- **Exercise 14:** Encapsulation Demonstration
- **Estimated time:** 00:15 minutes
- **Aim:** Use encapsulation to restrict access to class attributes. Create a class with private attributes and demonstrate encapsulation principles.
- **Learning objective:**
  - The learner will understand the concept of encapsulation in Python and how to use it to control access to class attributes
- **Learning outcome:**
  - The learner will be able to define a class with private attributes, implement getter and setter methods, and demonstrate the benefits of encapsulation

---

Figure 3-20. Self evaluation: Exercise 14

Self evaluation exercise 14 is as stated above:



IBM ICE (Innovation Centre for Education)

## Self evaluation: Exercise 15

- **Exercise 15:** Abstract Geometric Shape Class
- **Estimated time:** 00:15 minutes
- **Aim:** Create an abstract class representing a geometric shape with abstract methods like area and perimeter. Define derived classes (e.g., Circle, Rectangle) to implement these methods.
- **Learning objective:**
  - The learner will understand the concept of abstract classes in Python and how to use them to define a common interface for related classes
- **Learning outcome:**
  - The learner will be able to create an abstract class for geometric shapes, define abstract methods for common operations, and implement those methods in derived classes

---

Figure 3-21. Self evaluation: Exercise 15

Self evaluation exercise 15 is as stated above:

## Checkpoint (1 of 2)



IBM ICE (Innovation Centre for Education)

### Multiple choice questions:

1. Inheritance in Python allows a class to:
  - a) Be instantiated multiple times
  - b) Inherit attributes and methods from another class
  - c) Be defined with private methods only
  - d) Be used as a base class and a derived class simultaneously
2. What is polymorphism in Python?
  - a) The ability of a class to inherit from multiple classes
  - b) The ability of a class to have multiple methods with the same name
  - c) class to hide its implementation details
  - d) The ability of a class to access variables of another class
3. What is the purpose of the `__init__` method in a Python class?
  - a) It is used to initialize class attributes.
  - b) It defines the inheritance hierarchy.
  - c) It is required for encapsulation.
  - d) It handles exceptions in the class.

Figure 3-22. Checkpoint (1 of 2)

Write your answers here:

- 1.
- 2.
- 3.

## Checkpoint (2 of 2)



IBM ICE (Innovation Centre for Education)

### Fill in the blanks:

1. \_\_\_\_\_ is a special method in Python classes that is automatically called when an object is created.
2. A \_\_\_\_\_ is an instance of a class.
3. In Python, a \_\_\_\_\_ is a blueprint for creating objects.
4. Encapsulation in Python helps to hide the internal state of an object and restrict access to certain attributes by using\_\_\_\_\_.

### True/False:

1. Inheritance allows a class to acquire properties and behaviors from multiple parent classes in Python. True/False
2. Encapsulation helps in achieving data hiding by restricting access to certain attributes and methods. True/False
3. Polymorphism allows a single function or method name to be used for different types of objects in Python. True/False

---

Figure 3-23. Checkpoint (2 of 2)

Write your answers here:

Fill in the blanks:

- 1.
- 2.
- 3.
- 4.

True or false:

- 1.
- 2.
- 3.

# Question bank (1 of 2)



IBM ICE (Innovation Centre for Education)

## Two mark questions:

1. Discuss two types of methods commonly used in Python classes. Provide examples for each type and explain their significance in object-oriented programming.
2. Differentiate between instance variables and class variables in Python. Provide an example of each and explain when it is appropriate to use them in a class.
3. Explain the purpose of a constructor in Python classes. Provide an example of a simple constructor and describe the role it plays during object instantiation.
4. List at least three key features of object-oriented programming (OOP) and briefly explain their significance.

## Four mark questions:

1. Create a class Employee with following properties

First Name

Last Name

Pay

Email: Should be automatically generated as: Firstname + '.' + Lastname + "@company.com"

Test the code with following information of an Employee:

First name is: Mohandas

Last name is: Gandhi

Pay is: 50000

Employee
Properties:
First Name
Last Name
Pay
Email

Figure 3-24. Question bank (1 of 2)

## Question bank (2 of 2)



IBM ICE (Innovation Centre for Education)

2. Explain Constructors in Python .
3. Explain the concept of operator overloading in OOP. Provide a Python code example to demonstrate how to overload the + operator for a custom class named Vector that represents 2D vectors.
4. What is polymorphism? Demonstrate its use in Python with a simple example.

### Eight mark questions:

1. (a) Describe the concepts of inheritance and polymorphism in OOP, and explain how they are implemented in Python.  
(b) Create a base class named Employee with attributes for name and salary. Derive two subclasses from it: Manager (which adds a bonus attribute) and Developer (which adds a programming\_language attribute). Implement a method in each subclass to calculate and return the total compensation for that employee type. Create instances of both subclasses and demonstrate their usage, including calling their respective compensation methods.
2. (a) Explain how object-oriented principles can be leveraged to enhance security in Python applications. Provide specific examples of how encapsulation, inheritance, and access control can contribute to a more secure codebase.  
(b) Consider a scenario where you are building a Python application that handles sensitive user data. Design a class structure and access control mechanisms using OOP principles to ensure secure storage, retrieval, and modification of this data. Include examples of methods and attributes that would enforce appropriate access levels and prevent unauthorized data manipulation

---

Figure 3-25. Question bank (2 of 2)

## Unit summary



IBM ICE (Innovation Centre for Education)

- Having completed this unit, you should be able to:
  - Explore object-oriented programming (OOP) concepts and apply them to create reusable and modular code

---

*Figure 3-26. Unit summary*

Unit summary is as stated above.

---

# Unit 4. Error Handling and Exception Handling

## Overview

This Unit will provide an overview of Explore object-oriented programming (OOP) concepts and apply them to create reusable and modular code.

## How you will check your progress

- Checkpoint

## References

IBM Knowledge Center

# Unit objectives



IBM ICE (Innovation Centre for Education)

- After completing this unit, you should be able to:
  - Explore object-oriented programming (OOP) concepts and apply them to create reusable and modular code
- Learning outcomes:
  - Develop object-oriented Python code, encapsulating data and behavior in classes and objects

---

Figure 4-1. Unit objectives

Unit objectives and outcomes are as stated above.



IBM ICE (Innovation Centre for Education)

## Error handling and exception handling

- In this chapter, an introduction to exception handling and file handling (I/O) in Python is provided.
- The discussion begins with an exploration of the fundamentals of handling exceptions in Python, delving into the understanding of various types of exceptions and common built-in exceptions.
- The subsequent topics include the practical aspects of handling exceptions using try and except blocks, along with the nuanced handling of specific and multiple exceptions.
- The introduction further addresses the concept of custom exceptions, where the creation of custom exception classes and the process of raising exceptions are explored.

---

*Figure 4-2. Error handling and exception handling*

Transitioning to the realm of file handling (I/O), the content unfolds with an exploration of reading and writing files. The procedures involved in opening and closing files, as well as specific techniques for reading and writing both text and binary files, are detailed. The discussion then delves into the intricacies of working with text and binary files, encompassing essential aspects such as text encoding and decoding. The section concludes with a comprehensive exploration of best practices in file handling, with an emphasis on the recommended use of the `with` statement for effective file management and the incorporation of error-handling strategies in file operations.

### Introduction to exception

In programming languages, an exception is an event or occurrence that interrupts the normal flow of a program's execution. Exceptions are typically raised when an error or unexpected condition occurs during the runtime of a program. Instead of allowing the program to proceed with a potentially invalid or unpredictable state, exceptions provide a mechanism to handle such situations gracefully. In Python, an exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. When an exceptional situation arises, an object representing the exception is created and thrown (or "raised") in the program. If the program does not handle the exception, it will terminate, and an error message will be displayed.

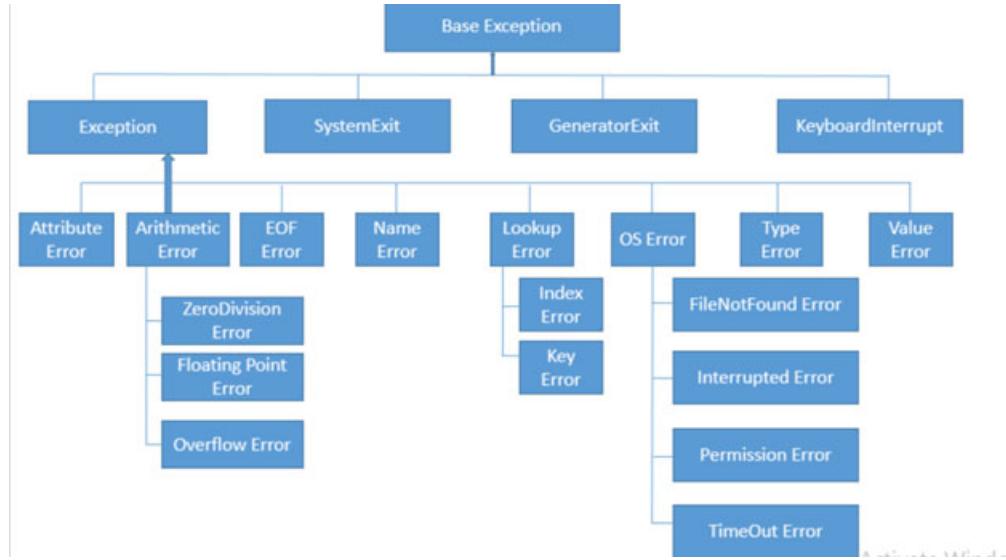


Figure 1: Exception hierarchy in python

[<https://chercher.tech/python-programming/python-exception-handling>]

#### Here are some common scenarios that can lead to exceptions:

- **Runtime errors:** These are errors that occur during the execution of the program. Examples include division by zero, attempting to access an index that is out of range, or trying to perform an operation on incompatible data types.
- **External factors:** Operations that involve external resources, such as file I/O, network communication, or database access, can result in exceptions. For example, trying to open a file that doesn't exist or attempting to connect to a server that is unreachable.
- **Logical errors:** These are errors in the program's logic that might not be immediately apparent. They can lead to unexpected behavior and trigger exceptions.

The handling of exceptions typically includes the use of try-catch (or try-except) blocks. In a try block, the code that might raise an exception is placed, and in the corresponding catch (or except) block, the code to be executed when that particular exception occurs is specified.



IBM ICE (Innovation Centre for Education)

## Understanding exceptions in python

- Exception handling in Python is a mechanism that allows errors and unexpected situations during the execution of a program to be gracefully managed.
- A try-except block is employed by Python to handle exceptions.
- A breakdown of the key components is provided below.

---

Figure 4-3. Understanding exceptions in python

### Common built-in exceptions

The table listed below represent common situations in Python programming where exceptions may occur. Understanding these exceptions helps developers anticipate potential errors and handle them appropriately in their code.

Scenario	Description
Division by zero	Occurs when attempting to divide a number by zero, which is mathematically undefined.
File not found	Raised when attempting to access a file that does not exist or cannot be found in the specified directory.
Index out of range	Occurs when trying to access an index that is outside the bounds of a sequence, such as a list or tuple.
Key error	Raised when attempting to access a key that does not exist in a dictionary.
Type error	Occurs when performing an operation on incompatible data types.
Value error	Raised when a function receives an argument of the correct type but with an invalid value.
Attribute error	Occurs when trying to access or modify an attribute that does not exist on an object.
Import error	Raised when an imported module cannot be found or loaded.
IOError	Raised when an input/output operation fails, such as when reading from or writing to a file.
Memory error	Occurs when an operation requires more memory than is available.
KeyboardInterrupt	Raised when the user interrupts the execution of a program, typically by pressing Ctrl+C.
Name error	Raised when an identifier (variable or function name) is used that has not been defined.
Overflow error	Occurs when a numerical operation exceeds the limits of the data type, resulting

Recursion error	Raised when the maximum recursion depth is exceeded, typically due to infinite recursion.
Syntax error	Occurs when the syntax of a Python script is invalid and cannot be parsed by the interpreter.
ZeroDivisionError	A specific type of division error raised when dividing a number by zero.
FileNotFoundException	A specific type of file error raised when a specified file is not found.
IndexError	A specific type of index error raised when trying to access an index that does not exist in a sequence.
KeyError	A specific type of key error raised when attempting to access a key that does not exist in a dictionary.
TypeError	A specific type of type error raised when performing an operation on incompatible data types.
ValueError	A specific type of value error raised when a function receives an argument with an invalid value.
AttributeError	A specific type of attribute error raised when trying to access or modify a non-existent attribute of an object.
ImportError	A specific type of import error raised when an imported module cannot be found or loaded.
OSError	A generic class for I/O-related errors, such as file operation errors.
PermissionError	Raised when attempting to perform an operation without the necessary permissions, such as writing to a read-only file.

NotImplementedError	Raised when an abstract method or function that should be implemented in a subclass is not overridden.
EOFError	Raised when the end-of-file (EOF) condition is reached unexpectedly while reading input from a file or stream.

Table 1: Common Exception types

### SyntaxError

Description: This exception is raised when the interpreter encounters a syntaxError in the code, such as a misspelled keyword, a missing colon, or an unbalanced parenthesis.

Scenario	Description
Missing colon after `if`, `elif`, `else`, `for`, `while`, or `def` statement	Occurs when a colon (`:`) is omitted after `if`, `elif`, `else`, `for`, `while`, or `def` statements.
Missing parentheses in function or method call	Raised when parentheses are missing around the arguments in a function or method call.
Unmatched or missing quotation marks	Occurs when there are unmatched or missing quotation marks (`'` or `") in string literals.
Indentation error	Raised when there is incorrect or inconsistent indentation in Python code blocks.
Unexpected indentation within compound statement	Occurs when there is unexpected indentation within a compound statement, such as an `if` or `while` block.
Incorrect use of operators or operands	Raised when operators are used incorrectly, such as using `=` instead of `==` for comparison, or using unsupported operands.
Incorrect use of keywords	Occurs when Python keywords are used incorrectly, such as using `print` as a variable name.
Incorrect use of punctuation marks	Raised when punctuation marks, such as commas, semicolons, or periods, are used incorrectly or in unexpected places.
Missing or unmatched parentheses	Occurs when there are missing or unmatched parentheses in expressions or function definitions.
Misspelled keywords or identifiers	Raised when keywords or identifiers are misspelled, making them unrecognizable to the Python interpreter.

Table 2: Common syntax error in python

**Example codes**

```
# Missing colon after `if` statement
if True # Missing colon
    print("This line will cause a SyntaxError")

# Missing parentheses in function call
print "Hello, World!" # Missing parentheses

# Unmatched or missing quotation marks
message = "Hello, World!' # Unmatched quotation marks

# Indentation error
def example_function():
    print("Indentation error") # Incorrect indentation

# Unexpected indentation within compound statement
if True:
    print("This line has unexpected indentation")
    print("It will cause a SyntaxError")

# Incorrect use of operators or operands
```

```

x = 10

if x = 5: # Incorrect use of `=` instead of `==`
    print("x is equal to 5")

# Incorrect use of keywords

True = False # Using `True` as a variable name

# Incorrect use of punctuation marks

print("Hello"), "World" # Incorrect use of comma

# Missing or unmatched parentheses

def example_function:

    print("Missing parentheses") # Missing parentheses

# Misspelled keywords or identifiers

whlie True: # Misspelled `while`

    print("Misspelled keyword")

```

### TypeError

A `TypeError` in Python is an exception that occurs when an operation is performed on an object of an inappropriate type. In other words, the data type of an object does not support the operation being performed. Here are some common scenarios that can lead to `TypeError`:

Scenario	Description	Example
Incorrect operand type	Occurs when an operation is performed on operands of incompatible types.	<code>'result = "Hello" + 123'</code>
Incorrect function argument type	Raised when a function is called with arguments of incorrect types.	<code>'len(123)'</code>
Unsupported operation	Occurs when an operation is performed on an object that does not support it.	<code>'result = 10 / "2"'</code>
Incorrect usage of built-in functions/methods	Raised when built-in functions or methods are used incorrectly with incompatible types.	<code>'list(123)'</code>
Incorrect usage of operators	Occurs when operators are used incorrectly with incompatible types or unsupported operands.	<code>'result = "Hello" * "World"'</code>
Incorrect usage of methods	Raised when methods are called on objects with incorrect types or incompatible arguments.	<code>'"Hello".append("World")'</code>
Incorrect usage of data types	Occurs when incorrect data types are used in Python constructs, such as list indexing, tuple unpacking, etc.	<code>'x = (1, 2, 3)' &lt;br&gt; 'y = x[0]' &lt;br&gt; '+ "Hello"'</code>

Table 3: Type error examples

Explanation:

- **Incorrect operand type:**

This occurs when an operation is performed on operands of incompatible types.

Example: `result = "Hello" + 123`.

Here, the `+` operator is used to concatenate a string ("Hello") with an integer (123), which results in a `TypeError` because concatenation is not supported between a string and an integer.

- **Incorrect function argument type:**

Raised when a function is called with arguments of incorrect types.

Example: `len(123)`.

The `len()` function expects a sequence or collection as an argument, but it is called with an integer (123), resulting in a `TypeError`.

- **Unsupported operation:**

Occurs when an operation is performed on an object that does not support it.

Example: `result = 10 / "2"`

Here, division is performed between an integer (10) and a string ("2"), which is not supported, leading to a `TypeError`.

- **Incorrect usage of built-in functions/methods:**

Raised when built-in functions or methods are used incorrectly with incompatible types.

Example: `list(123)`

The `list()` function expects an iterable object as an argument, but it is called with an integer (123), resulting in a `TypeError`.

- **Incorrect usage of operators:**

Occurs when operators are used incorrectly with incompatible types or unsupported operands.

Example: `result = "Hello" * "World"`

The `*` operator is used to multiply two strings ("Hello" and "World"), which is not supported, resulting in a `TypeError`.

- **Incorrect usage of methods:**

Raised when methods are called on objects with incorrect types or incompatible arguments.

Example: `"Hello".append("World")`

The `append()` method is called on a string ("Hello"), which does not support appending, leading to a `TypeError`.

- **Incorrect usage of data types:**

Occurs when incorrect data types are used in Python constructs, such as list indexing, tuple unpacking, etc.

Example 1: `x = (1, 2, 3)`

`y = x[0] + "Hello"`

Here, a tuple (`x`) is indexed with an integer (0), but the result is concatenated with a string ("Hello"), resulting in a `TypeError`.



## NameError

- A NameError in Python is an exception that is encountered when a variable or a name is not defined in the current scope.
- This usually occurs when a variable or a function is attempted to be used without having been assigned or defined.

Scenario	Description	Example
Misspelled Variable Name	Occurs when a variable is referenced with a name that has not been defined or is misspelled.	<code>'print(variable_name)'</code>
Undefined Function or Method	Raised when a function or method is called but has not been defined or imported.	<code>'undefined_function()'</code>
Undefined Class or Object	Occurs when a class or object is referenced but has not been defined or imported.	<code>'obj = UndefinedClass()'</code>
Incorrect Module Name	Raised when attempting to import a module with an incorrect name or one that does not exist.	<code>'import non_existent_module'</code>
Scope Issue	Occurs when trying to access a variable or function outside its scope, such as within a function or loop.	<code>'def my_function():&lt;br&gt;    print(my_variable)'</code>

Table 4: NameError examples

Figure 4-4. NameError

- IndexError

An IndexError in Python is an exception that is triggered when an attempt is made to access an index of a sequence (such as a list or a tuple) that falls outside the valid range. The index must be within the bounds of the sequence.

The below examples illustrate common scenarios where an IndexError may occur in Python code. Understanding these errors can help developers debug and correct their code effectively.

Scenario	Description	Example
Index out of range	Occurs when trying to access an index that is outside the bounds of a sequence, such as a list, tuple, or string.	my_list = [1, 2, 3] print(my_list[3])
Negative index	Raised when attempting to access elements using negative indexing on a sequence that does not support it.	my_string = "Python" print(my_string[-10])
Accessing non-existent item in collection	Occurs when attempting to access an item at an index that does not exist in a collection, such as a list or dictionary.	my_dict = {'a': 1, 'b': 2} print(my_dict['c'])
Incorrect use of indexing operator	Raised when the indexing operator is used incorrectly, such as applying it to a non-indexable object.	my_integer = 123 print(my_integer[0])
Empty sequence	Occurs when trying to access an element from an empty sequence.	empty_list = [] print(empty_list[0])

Table 5: IndexError Examples

- **KeyError**

A KeyError in Python is an exception that is encountered when an attempt is made to access a key in a dictionary that is not present. This usually occurs when the value associated with a key that does not exist in the dictionary is attempted to be retrieved.

The below examples illustrate common scenarios where a KeyError may occur in Python code. Understanding these errors can help developers debug and correct their code effectively.

Scenario	Description	Example
Key not found in dictionary	Occurs when trying to access a key that does not exist in a dictionary.	my_dict = {'a': 1, 'b': 2} print(my_dict['c'])
Accessing non-existent key in dictionary	Raised when attempting to access a value associated with a key that is not present in the dictionary.	my_dict = {'a': 1, 'b': 2} print(my_dict['c'])
Incorrect use of dictionary methods	Occurs when dictionary methods like `get()` or `pop()` are used with a key that does not exist in the dictionary.	my_dict = {'a': 1, 'b': 2} print(my_dict.get('c'))
Accessing non-existent item in collection	Occurs when attempting to access an item at a key that does not exist in a collection, such as a dictionary.	my_dict = {'a': 1, 'b': 2} print(my_dict['c'])

Table 6: KeyError Examples

- **ValueError**

A ValueError in Python is an exception that occurs when a function receives an argument of the correct type but an inappropriate value. It indicates that the input to a function or operation is outside the expected range or does not meet certain criteria.

The below examples illustrate common scenarios where a ValueError may occur in Python code. Understanding these errors can help developers debug and correct their code effectively.

Scenario	Description	Example
Incorrect argument type	Occurs when a function or method is called with an argument of the correct type but with an inappropriate value.	<code>int("abc")</code> <code>print(float("abc"))</code>
Missing or incorrect function arguments	Raised when a function or method is called with too few or too many arguments, or with arguments of the wrong type.	<code>sum([1, 2, "3"])</code>
Incorrect conversion between data types	Occurs when attempting to convert a value from one data type to another, but the value is not compatible with the target type.	<code>int("3.14")</code> <code>print(int("abc"))</code>
Incorrect format for string conversion	Raised when attempting to convert a string to another data type, but the string does not have the expected format.	<code>int("ten")</code> <code>print(float("3,14"))</code>

Table 7: Examples of ValueError

In this example, if the user enters a value that is not an integer or is not between 1 and 10 (inclusive), a ValueError will be raised with a corresponding error message. When faced with a ValueError in the code, attention should be given to the specific error message and traceback to identify the line of code causing the issue. Subsequently, the input values or conditions should be checked to ensure they meet the expected criteria.



IBM ICE (Innovation Centre for Education)

## Handling exceptions with try and except

- Exception handling in Python is a way to deal with unexpected errors or exceptional situations that may occur during the execution of a program.
- The try and except blocks are used for this purpose.

*Figure 4-5. Handling exceptions with try and except*

### Using try-except blocks

The try block encompasses the code where an exception might occur. It represents the portion of the code intended to be monitored for errors. If an exception occurs in the try block, the normal flow of execution is interrupted, and control is transferred to the nearest except block.

```
try:
    # Code that may raise an exception
    result = 10 / 0  # This will raise a ZeroDivisionError
except:
    # Code to handle the exception
    print("An error occurred!")
```

### Handling specific exception

Specific exceptions can be caught and handled differently. The ZeroDivisionError exception will be caught exclusively by the except block with ZeroDivisionError, while the other except block will catch other arithmetic errors.

```
try:
    result = 10 / 0
except ZeroDivisionError:
```

```

        print("Cannot divide by zero!")
except ArithmeticError:
    print("Arithmetic error occurred!")
except:
    print("An error occurred!")

```

**Example:**

```

try:
    # Code that may result in exceptions
    result = 10 / 0
    value = int("abc")
    index = [1, 2, 3][5]
except ZeroDivisionError as division_error:
    print(f"Error: Division by zero - {division_error}")
except ValueError as value_error:
    print(f"Error: Invalid conversion - {value_error}")
except IndexError as index_error:
    print(f"Error: Index out of range - {index_error}")
except Exception as generic_error:
    print(f"Error: An unexpected error occurred - {generic_error}")
finally:
    print("This block is always executed, regardless of exceptions.")

```

**In this illustration:**

- The try block encapsulates code that could potentially raise various types of exceptions.
- Each except block specifies a distinct exception type and outlines the corresponding action for that specific exception.
- If multiple exceptions are possible, Python will execute the first matching except block.
- The presence of a finally block ensures execution irrespective of whether an exception occurs or not.

This methodology enables the graceful handling of diverse exception scenarios, offering specific responses for different error conditions.



IBM ICE (Innovation Centre for Education)

## Handling multiple exception

- This methodology enables the graceful handling of diverse exception scenarios, offering a consolidated response for multiple error conditions.

*Figure 4-6. Handling multiple exception*

**In the below code:**

- The try block encapsulates code that could potentially raise various types of exceptions.
- Multiple exception types are grouped in a single except block using parentheses, allowing a common action to be defined for them.
- If multiple exceptions are possible, python will execute the first matching except block.
- The presence of a finally block ensures execution irrespective of whether an exception occurs or not.

```
try:
    # Code that may result in exceptions
    result = 10 / 0
    value = int("abc")
    index = [1, 2, 3][5]
except (ZeroDivisionError, ValueError, IndexError) as error:
    print(f"Error: {error}")
except Exception as generic_error:
    print(f"Error: An unexpected error occurred - {generic_error}")
finally:
```

```
print("This block is always executed, regardless of exceptions.")
```



## Finally block

- In exception handling, the use of a finally block is considered important for several reasons.
- It allows resources to be released or cleanup operations to be performed regardless of whether an exception is raised or not.
- When exceptions occur, the finally block ensures that essential tasks, such as closing files or network connections, are executed, contributing to the robustness and reliability of the code.
- Additionally, the finally block is beneficial in scenarios where post-exception actions need to be taken, providing a designated space for code that should always be executed, and enhancing the overall integrity of the program.

*Figure 4-7. Finally block*

```
def read_file(file_path):
    try:
        # Open the file for reading
        with open(file_path, 'r') as file:
            content = file.read()
            print(f"File content: {content}")
        # Simulate a potential exception
        result = 10 / 0
    except ZeroDivisionError as division_error:
        print(f"Error: Division by zero - {division_error}")
    except FileNotFoundError as file_not_found_error:
        print(f"Error: File not found - {file_not_found_error}")
    finally:
        # Ensure the file is closed, regardless of exceptions
        print("Finally block: Closing the file.")
        if file and not file.closed:
```

```
    file.close()  
  
# Example Usage  
read_file("example.txt")
```

**In this scenario:**

- The try block attempts to read the content of a file and performs a division that will result in a ZeroDivisionError.
- The except blocks handle specific exceptions that might occur during file reading or the division operation.
- The finally block is guaranteed to execute, ensuring that the file is closed even if an exception occurs.

This example illustrates the practical use of a finally block to perform cleanup tasks, such as closing a file, regardless of whether an exception is raised or not.



IBM ICE (Innovation Centre for Education)

## Custom exceptions

- In Python, custom exceptions are instances of user-defined exception classes that allow the creation of exceptions tailored to specific scenarios in code.
- When existing built-in exceptions do not accurately capture the nature of errors in a particular application, developers have the option to define new exception classes.
- A custom exception is typically created by defining a new class that inherits from the built-in Exception class or one of its subclasses.
- For example, a custom exception class named CustomError can be raised using the raise keyword within a try block.
- This approach contributes to making the code more expressive and handling specific error conditions in a clear and organized manner, particularly in larger projects where precise error reporting is essential.

*Figure 4-8. Custom exceptions*

### Creating custom exception classes

Here's a simple example of a custom exception class.

#### Class CustomError(Exception):

```
def __init__(self, message="A custom error occurred."):
    self.message = message
    super().__init__(self.message)
```

In this example, CustomError is a custom exception that is inherited from the built-in Exception class. It incorporates an init method to initialize the exception with an optional custom error message. This custom exception can be raised using the raise keyword:

```
try:
    # Some code that may raise a custom exception
    raise CustomError("This is a specific error message.")

except for CustomError as ce:
    print(f"Caught an exception: {ce}")
```

## Example

In this code, a custom exception class, `InsufficientFundsError` is defined to handle situations where a user attempts to withdraw more money from an account than is available. The class includes an `__init__` method to initialize the exception with details about the account balance and withdrawal amount, creating a specific error message. The `withdraw_from_account` function is then designed to handle this custom exception by checking if the withdrawal amount exceeds the account balance. If so, it raises the `InsufficientFundsError` otherwise, it performs the withdrawal and prints the updated balance. The example usage demonstrates the application of this custom exception when attempting to withdraw an amount greater than the available balance. The caught custom error is then printed for reference.

### Class `InsufficientFundsError(Exception)`:

```
def __init__(self, account_balance, withdrawal_amount):
    self.account_balance = account_balance
    self.withdrawal_amount = withdrawal_amount
    super().__init__(f"Insufficient funds. Current balance: {account_balance}, Withdrawal attempted: {withdrawal_amount}")

# Example Usage

def withdraw_from_account(account_balance, withdrawal_amount):
    try:
        if withdrawal_amount > account_balance:
            raise InsufficientFundsError(account_balance, withdrawal_amount)
        else:
            updated_balance = account_balance - withdrawal_amount
            print(f"Withdrawal successful. Updated balance: {updated_balance}")
    except InsufficientFundsError as funds_error:
        print(f"Error: {funds_error}")

# Applying the custom exception

try:
    withdraw_from_account(1000, 1200)
except InsufficientFundsError as custom_error:
    print(f"Custom error caught: {custom_error}")
```

## Raising exceptions



IBM ICE (Innovation Centre for Education)

- The raise keyword in Python is significant as it allows developers to explicitly raise exceptions, indicating that a specific error condition has occurred within their code.
- Here are some key points regarding the significance of the raise keyword.

---

Figure 4-9. Raising exceptions

- **Error signaling:** Raise is used to signal that a predefined or custom exception should be raised at a specific point in the program. This is crucial for indicating exceptional situations that require special handling.
- **Custom exceptions:** Developers can create their exception classes by defining a new exception type and raising instances of it using the raise keyword. This facilitates the development of more meaningful and context-specific error messages.
- **Error handling:** The raise keyword is often employed within try blocks to raise exceptions when certain conditions, such as validation failures or unexpected states, are encountered. This, in turn, triggers the corresponding except blocks for appropriate error handling.
- **Program control flow:** By raising exceptions, developers can alter the normal control flow of a program, directing it to an exception handler. This is particularly useful for handling exceptional cases gracefully without causing the program to crash.
- **Debugging and logging:** Raising exceptions with informative error messages aids in debugging, allowing developers to quickly identify the cause of issues. It also facilitates logging, as raised exceptions can be logged along with relevant details.

Example:

In a banking system, raising exceptions is crucial for handling various scenarios that can occur during transactions and account management. Here's an example scenario:

Scenario: Insufficient Funds for Withdrawal.

Description: Consider a situation where a customer attempts to withdraw an amount from their bank account. Before processing the withdrawal, the system needs to verify if the account balance is sufficient to cover the withdrawal amount. If the balance is insufficient, an exception should be raised to notify the customer that the transaction cannot be completed due to insufficient funds.

Example:

```
class InsufficientFundsError(Exception):
    """Exception raised when there are insufficient funds in the account."""
    def __init__(self, amount, balance):
        super().__init__(f"Insufficient funds: Cannot withdraw {amount}. Available balance is {balance}.")

class BankAccount:
    def __init__(self, account_number, balance):
        self.account_number = account_number
        self.balance = balance
    def withdraw(self, amount):
        if amount > self.balance:
            raise InsufficientFundsError(amount, self.balance)
        else:
            self.balance -= amount
            print(f"Withdrawal of {amount} successful. Remaining balance: {self.balance}")
# Example usage
try:
    account = BankAccount("123456789", 1000)
    withdrawal_amount = 1500 # Attempt to withdraw more than available balance
    account.withdraw(withdrawal_amount)
except InsufficientFundsError as e:
    print(e)
```

### Explanation:

- In this example, we define a custom exception class `InsufficientFundsError`, which inherits from the base `Exception` class. This exception is raised when there are insufficient funds in the account for a withdrawal operation.
- The `BankAccount` class represents a bank account with attributes `account_number` and `balance`.
- The `withdraw()` method of the `BankAccount` class checks if the withdrawal amount exceeds the account balance. If so, it raises an `InsufficientFundsError`; otherwise, it deducts the amount from the balance.
- In the example usage, we attempt to withdraw an amount that exceeds the available balance. As a result, the `InsufficientFundsError` exception is raised, providing details about the insufficient funds.

This example demonstrates how exceptions can be used in a banking system to handle scenarios where transactions cannot be completed due to insufficient funds. It helps ensure that the system operates securely and provides informative feedback to users about transaction failures.



IBM ICE (Innovation Centre for Education)

## File handling (I/O)

File handling in Python is deemed significant to be learned as part of coding for several reasons:

- **Data persistence:** Data can be persistently stored on disk, enabling its preservation between program runs and facilitating data sharing and access across different applications.
- **Configuration and settings:** Essential for reading and writing configuration settings, file handling allows the modification of program behavior without altering the source code.
- **Input and output operations:** fundamental for input and output operations, file handling facilitates the input of information into a program and the generation of output or storage of results.
- **Data processing:** Crucial for processing large datasets, file handling enables efficient reading, processing, and analysis of data, particularly in data science, analytics, and data processing applications.
- **Logging and debugging:** Instrumental for logging and debugging purposes, file handling allows programs to write log files, recording events, errors, or debugging information for developers to analyze.

---

Figure 4-10. File handling (I/O)

File handling in Python involves working with files, which can be reading from existing files, writing to new or existing files, and performing various operations related to file manipulation.



IBM ICE (Innovation Centre for Education)

## Reading and writing files

- Reading and writing files in Python is accomplished through the use of file-handling operations.
- This process is essential for inputting and outputting data, persistently storing information, and performing various file-related tasks.

---

Figure 4-11. Reading and writing files

### Opening and closing files

File operations in Python involve establishing a connection with a file, manipulating its contents, and then releasing the allocated resources.

**Using a simple approach:** The open function is invoked with the file path and desired access mode. A file object is returned, representing the established connection.

#### # Open file in read mode

```
file_handle = open("myfile.txt", "r")
```

#### # Read file contents

```
data = file_handle.read()
```

#### # Process data (e.g., print, manipulate)

```
print(data)
```

#### # Close the file explicitly

```
file_handle.close()
```

- **Context manager approach:** Alternatively, the with statement can be used to manage the file object automatically. The file is opened within the block and closed upon exiting, regardless of exceptions.

The following examples illustrate reading and writing files in Python-

#### # A file is opened for writing

```

with open('example.txt', 'w') as file:
    # The file is written to
    file.write('Hello, World!')

# The same file is opened for reading
with open('example.txt', 'r') as file:
    # The content of the file is read
    content = file.read()
    print(content) # Output: Hello, World!

```

**In the example above:**

- The `open('example.txt', 'w')` opens the file named 'example.txt' in write mode.
- `with` statement is used to ensure that the file is properly closed after writing. This is good practice as it automatically takes care of closing the file even if an exception occurs.
- The `write()` method is used to write the string 'Hello, World!' to the file.
- Then, the same file is opened for reading using `open('example.txt', 'r')`.
- The `read()` method is used to read the content of the file, and it's printed to the console.
- **File modes:** In Python, different file modes are employed to dictate how data is interacted with within a file. These modes operate, dictating what actions can be performed and how existing data is treated. Here's a breakdown of some common modes:

Highlight the commands for which you are explaining.

File modes can be in table format which is easy to remember.

done

Mode	Description
'r'	Open for reading. Default mode.
'w'	Open for writing. Truncates existing file or creates a new file.
'x'	Open for exclusive creation, failing if the file already exists.
'a'	Open for writing. Appends to the end of the file if it exists, otherwise creates a new file.
'b'	Binary mode. Appends 'b' to other modes (e.g., 'rb' or 'wb').
't'	Text mode. Default mode, but can be appended to other modes (e.g., 'rt' or 'wt').
'+'	Open for updating (reading and writing). Appends '+' to other modes (e.g., 'r+' or 'w+').

Table 8: File modes in python

**Read Mode ("r")**

- File contents are accessed but cannot be modified.
- Existing data is read without alteration.

Example:

```

# Existing file "poem.txt" is accessed without modification
with open("poem.txt", "r") as f:
    poem = f.read()
    print(poem) # Prints the entire poem text

```

**Write Mode ("w")**

- New data is written to the file, overwriting any existing content.
- The file is created if it doesn't already exist.

**Example:**

```
# A new file, "data.csv" is created, and content is inserted
with open("data.csv", "w") as f:
    f.write("name,age\nJohn,30\nJane,25")
    # Existing data in "data.csv" (if any) would be overwritten
```

**Append Mode ("a")**

- New data is added to the end of the file.
- Existing content remains untouched.

**Example:**

```
# New lines are added to the end of "log.txt."
with open("log.txt", "a") as f:
    f.write("Entry at 12:30 PM: System online.\n")
    f.write("Entry at 1:00 PM: Process completed successfully.\n")
```

**Binary Mode ("b")**

- Treats the file as a sequence of bytes, preserving their exact representation.
- Useful for handling non-textual data like images or videos.

**Example-**

```
# Image data is read from "photo.jpg" without interpreting it as text
with open("photo.jpg", "rb") as f:
    image_data = f.read()
    # This data can be processed or written to another file in binary format
```

**Text Mode ("t")**

- Interprets the file as a sequence of textual characters.
- Newline characters are automatically handled based on the operating system.

**Additional Modes:**

- "r+" allows both reading and writing at the same time.
- "w+" opens the file for reading and writing, truncating existing content.
- "a+" allows reading and appending to the file.

These modes operate silently behind the scenes, influencing how data is accessed and manipulated within the file. Choosing the appropriate Mode is crucial for achieving the desired outcome while ensuring data integrity.



## Reading and writing text files

- Reading a file: Reading from a file in Python can be approached in various ways, depending on the requirements and the nature of the data in the file.
- Here are several approaches to reading from a file in Python:
  - Reading the entire file at once: use the `read()` method to read the entire content of the file as a single string.

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

`content = file.read()`

`print(content)`

Figure 4-12. Reading and writing text files

- Reading line by line: use the `readline()` method to read the file line by line in a loop.

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

- Iterating over the file object: directly iterate over the file object, which treats each line as an item in the iteration.

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

- Reading a specific number of characters: use the `read(n)` method to read a specific number of characters from the file.

```
with open('example.txt', 'r') as file:
    content_part = file.read(10) # Read the first 10 characters
    print(content_part)
```

- Reading into a list: use the `readlines()` method to read all lines into a list, where each element represents a line in the file.

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

```

lines = file.readlines()
print(lines)

```

- **Writing to a file**

Writing to a file in Python involves various approaches, depending on the nature of the data and the desired output. Here are several approaches to writing to a file in Python:

- Writing a single string: use the write() method to write a single string to the file.

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

```
file.write("This is a line of text.")
```

- Writing multiple lines: use the write\_lines () method to write multiple lines to the file.

lines = ["Line 1\n", "Line 2\n", "Line 3\n"]

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

```
file.writelines(lines)
```

- Appending to an existing file: Open the file in append mode ('a') to add content to the end of an existing file.

with open('existing\_file.txt', 'a') as file:

```
file.write("Appended line.")
```

- Formatting data before writing: Format data using string formatting or other techniques before writing.

name = "John"

age = 30

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

```
file.write(f"Name: {name}, Age: {age}")
```

- Using print function: Redirect the print() function's output to a file.

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

```
print("This is a line of text.", file=file)
```

The output of the print() function is redirected to a file by opening the file named 'output.txt' with the 'w' mode, and a line of text, "This is a line of text," is printed to the file using the print() function with the file specified as the output.

- Writing CSV or JSON data: Use specialized libraries like csv or JSON to write structured data.

import csv

data = [["Name", "Age"], ["John", 30], ["Jane", 25]]

with open('output.csv', 'w', newline='') as file:

```
csv_writer = csv.writer(file)
```

```
csv_writer.writerows(data)
```

### In the below example:

- Student marks are read from the input file: The content of the file named 'student\_marks.txt' is read, and the lines are stored in the variable lines using the readlines() method.
- A method is defined to calculate the average: The function calculate\_average is defined to calculate the average of a list of marks.
- Student marks are processed, and averages are calculated: Each line from the input file is processed, extracting student names and their respective marks. The average marks for each student are calculated using the defined method, and the results are formatted and stored in the results list.

- Average marks are written to the output file: The calculated average marks, along with the student names, are written to a new file named 'average\_marks.txt' using the write mode.
- The result is displayed: A message is printed to the console, indicating that the average marks have been calculated, and the results have been written to the 'average\_marks.txt' file.

**Input file (student\_marks.txt):**

John, 85, 90, 78, 92

Jane, 75, 88, 91, 82

Bob, 92, 86, 89, 78

Alice, 80, 94, 88, 85

**# Student marks are read from the input file**

```
input_file_path = 'student_marks.txt'
output_file_path = 'average_marks.txt'
with open(input_file_path, 'r') as input_file:
    lines = input_file.readlines()
```

**#Method is defined to calculate the average**

```
def calculate_average(marks):
    return sum(marks) / len(marks)
```

**# Student marks are processed, and averages are calculated**

```
results = []
for line in lines:
    data = line.strip().split(',')
    student_name = data[0]
    marks = [int(mark) for mark in data[1:]]
    average_mark = calculate_average(marks)
    results.append(f'{student_name}'s average mark: {average_mark:.2f}\n")
```

**# Average marks are written to the output file**

```
with open(output_file_path, 'w') as output_file:
    output_file.writelines(results)
```

**# The result is displayed**

```
print(f"Average marks have been calculated. Results have been written to {output_file_path}.")
```



IBM ICE (Innovation Centre for Education)

## Working with text and binary files

- In Python, text and binary files represent distinct domains for data manipulation, each governed by dedicated operations and considerations.
- Handling them effectively demands an understanding of their inherent qualities and the subtle nuances of interacting with them.
- Python handles text and binary data differently. In text files, information is stored in ASCII or Unicode characters, and each line of text is ended with a special character i.e End of Line (EOL) character. In python, the EOL character is the newline character ('\n') by default.
- **Text files:** Text files are designed for storing human-readable text.

*Figure 4-13. Working with text and binary files*

**Text files:** Text files are designed for storing human-readable text.

They are composed of characters encoded in a specific character set, such as ASCII or UTF-8. Text files are well-suited for storing plain text documents, configuration files, and other textual information.

- **Structure:** Text files are composed of sequences of characters that are interpreted as human-readable text. This implies that the contents of a text file are organized as a series of characters, and these characters form readable text when interpreted.
- **Operations:** Operations on text files primarily involve reading, writing, and manipulating character strings. Text files are typically subject to operations that include reading data from them, writing new data to them, and manipulating the existing content through various string operations.
- **Encoding:** Text files are encoded using character sets like UTF-8 or ASCII to represent characters as bytes. Character encoding is the method used to represent characters as binary data (bytes). UTF-8 and ASCII are examples of character encodings commonly used for text files.
- **Transparency:** Content in text files is directly interpretable and printable as readable text. The term "transparency" in this context means that the content of a text file is human-readable without any special processing. It can be interpreted and printed directly as readable text.

**Binary files:** Binary files are used for storing non-text data, such as images, audio, video, or any data that doesn't conform to a character encoding. Binary files preserve the exact structure of the data without interpreting it as text.

- **Structure:** Binary files are structured as sequences of bytes, often representing non-textual data such as images, audio, or executable programs. Unlike text files, binary files are not necessarily organized in a way that represents human-readable text. Instead, they consist of raw sequences of bytes that may encode various types of non-textual information.
- **Operations:** Operations on binary files involve byte-level manipulation and often require the use of dedicated libraries or frameworks. Manipulating binary files typically requires working at the level of individual bytes. Specialized libraries or frameworks may be necessary to perform operations like reading, writing, or modifying the binary data.
- **Encoding:** Binary files are not subject to specific character sets; each byte directly represents data. Unlike text files that use character encoding schemes like UTF-8 or ASCII, binary files do not adhere to specific character sets. Each byte in a binary file directly represents data, and its interpretation depends on the context of the file.
- **Opacity:** The content of binary files is not directly interpretable; it requires specific decoding or interpretation depending on the data type. Binary file content is not inherently readable by humans. Decoding or interpretation is necessary, and the method depends on the type of data the binary file represents. For example, decoding may be required for images, audio, or other binary data formats.

### Reading and writing text files

- Read/Write: Performed using file objects and methods like read, write, and readline.
- String handling: Utilize built-in string functions and regular expressions for processing text.
- Formatting: Leverage libraries like DateTime and JSON for data formatting and serialization.

### Example

**Scenario: Log file analysis:** In a data analysis scenario, a text file containing log data from a web server is provided. The objective is to analyze the log entries and extract relevant information, such as the number of successful and unsuccessful requests.

#### # A text file containing server logs is opened for reading

```
with open('server_logs.txt', 'r') as file:
    # The content of the text file is read
    log_content = file.read()
    # The log content is split into individual log entries
    log_entries = log_content.split('\n')
    # Initializing counters for successful and unsuccessful requests
    successful_requests = 0
    unsuccessful_requests = 0
    # Analyzing each log entry
    for entry in log_entries:
        # Checking if the entry indicates a successful request
        if "200 OK" in entry:
            # Incrementing the counter for successful requests
            successful_requests += 1
        else:
            # Incrementing the counter for unsuccessful requests
            unsuccessful_requests += 1
    # Outputting the analysis results
    print("Number of successful requests:", successful_requests)
```

```
print("Number of unsuccessful requests:", unsuccessful_requests)
```

**In this example:**

- A text file, assumed to be named 'server\_logs.txt', is opened for reading.
- The content of the text file is read into the variable log\_content.
- The log content is split into individual log entries based on newline characters.
- Counters for successful and unsuccessful requests are initialized.
- Each log entry is analyzed, and if it contains the string "200 OK," it is considered a successful request; otherwise, it is treated as an unsuccessful request.
- The counters are incremented accordingly.
- The results of the analysis are then printed.

### Reading and writing binary files

- Open/Close: Utilize the open function with binary mode ("b") to preserve byte accuracy.
- Byte manipulation: Employ libraries like struct and array for reading and writing specific data types.
- Decoders/Encoders: Use specialized libraries like PIL for images or wave for audio to interpret or generate binary data.

**Example:**

#### Scenario: Image processing

In a scenario involving image processing, a binary file containing raw image data from a digital camera is provided. The task is to read the binary file, process the image data, and generate a new processed image.

```
# A binary file containing raw image data is opened for reading
```

```
with open('raw_image_data.bin', 'rb') as file:
    # The content of the binary file is read
    image_data = file.read()

# Image processing operations are performed on the raw binary data
processed_image_data = perform_image_processing(image_data)

# A new binary file is opened for writing
with open('processed_image.bin', 'wb') as file:
    # The processed image data is written to the new binary file
    file.write(processed_image_data)
```

**In this example:**

- A binary file, assumed to be named 'raw\_image\_data.bin', is opened for reading.
- The content of the binary file is read into the variable image\_data.
- Image processing operations, which may involve manipulating pixel values, adjusting colors, or applying filters, are performed on the raw binary data. The specific details of the processing are abstracted in the example for simplicity.
- A new binary file, named 'processed\_image.bin', is opened for writing.
- The processed image data is written to the new binary file.

#### Nuances and considerations:

- **Text vs. Binary modes:** Choose the appropriate mode based on data type to avoid corruption or misinterpretation.
- **Encoding awareness:** Specify encoding when reading/writing text files to ensure proper character representation.

- **Error handling:** Robustly handle potential errors like invalid data or unexpected file formats.



IBM ICE (Innovation Centre for Education)

## Text encoding and decoding

- Within the Python realm, text encoding and decoding silently translate characters into sequences of bytes and vice versa.
- These quiet but critical processes seamlessly facilitate the exchange of textual data across diverse systems and languages.
- Let's consider an example: Imagine a poem composed in French, nestled within a file named "poeme.txt."
  - Encoding.
  - Sharing.
  - Decoding.
  - Mismatch.

*Figure 4-14. Text encoding and decoding*

- **Encoding:** Upon opening with the default mode, Python interprets the poem's characters based on the system's inherent encoding (often UTF-8). This encoding silently assigns unique byte sequences to each French character, such as "ç" or "é." These sequences are written to the file, representing the poem's invisible code.
- **Sharing:** Now, suppose the poem is to be shared with a friend who uses a different operating system. Their system may possess a distinct default encoding.
- **Decoding:** When the friend opens the "poeme.txt" file, the system attempts to interpret the byte sequences based on its own default encoding. If this encoding aligns with the original one, the French characters magically reappear in their proper forms.
- **Mismatch:** However, if the encodings differ, the byte sequences can be misinterpreted. This may result in garbled characters or unexpected symbols appearing in the poem, distorting its beauty.

### To prevent this, the encoding can be explicitly specified during both writing and reading:

- Encoding with UTF-8: By opening the file for writing with "`open('poeme.txt', 'w', encoding='utf-8')`", one instructs Python to ensure all characters are encoded using the widely compatible UTF-8 standard.
- Decoding with UTF-8: Similarly, the friend can instruct their system to interpret the bytes correctly by opening the file with "`open('poeme.txt', 'r', encoding='utf-8')`". This ensures the poem's original French form is restored in all its glory.

Remember: Specifying the correct encoding fosters cross-platform compatibility and prevents the appearance of garbled characters. Python offers a vast array of encoding options to cater to diverse languages and character sets. Choose the most fitting one based on the data and communication needs.



IBM ICE (Innovation Centre for Education)

## File handling best practices

- When handling files in Python, several best practices should be observed to ensure efficient and reliable file operations.
- Describing these practices emphasizes the general guidelines without specifying a particular actor.
- Here are some file-handling best practices.

---

Figure 4-15. File handling best practices

- The use of the `with` statement for file handling is encouraged: Files should be opened and managed using the `with` statement to ensure proper resource cleanup.
- Explicit specification of file modes is recommended: File modes (read, write, binary, etc.) should be explicitly specified when opening a file.
- Graceful handling of exceptions is emphasized: Exceptions during file operations should be handled gracefully to prevent program crashes.
- File existence should be checked before opening: Prior to attempting file operations, it is advisable to check whether a file exists to avoid potential issues.
- Platform-independent path handling methods are preferred: Path operations should be performed in a way that is compatible across different operating systems.
- Files should be closed explicitly if not using the `with` statement: Files should be explicitly closed to release system resources when the `with` statement is not utilized.
- Avoiding the hardcoding of file paths is recommended: File paths should be parameterized or dynamically generated instead of being hardcoded for flexibility.
- Proper file encoding should be ensured: When working with text files, the correct encoding should be specified to handle different character sets.
- Meaningful and descriptive file names contribute to code readability: Descriptive file names should be used to enhance code readability and maintainability.

- Consideration of file security measures is important: File permissions and security considerations should be taken into account when working with sensitive data.

### Use the statement for file handling

Using the with statement for file handling: Files are opened and managed using the with statement. By incorporating the statement, the file-handling process is managed in a manner that promotes clean and efficient resource management, enhancing the reliability of file operations in Python.

#### Example:

```
# A file is opened and managed using the with statement
with open('example.txt', 'r') as file:
    # The content of the file is read
    content = file.read()

# The file is automatically closed upon exiting the block
```

#### In this example:

- A file is opened and managed using with the statement: This emphasizes the action of opening and managing the file, ensuring that resources are handled appropriately.
- The content of the file is read: Within the context of the with statement, file operations, such as reading content, are performed.
- The file is automatically closed upon exiting the with block: The file is closed automatically when the block of code within the with statement is exited, contributing to better resource management. The with statement is employed to ensure proper resource cleanup. Resources associated with file handling are automatically released when the block is exited. By incorporating the statement, the file-handling process is managed in a manner that promotes clean and efficient resource management, enhancing the reliability of file operations in Python.



IBM ICE (Innovation Centre for Education)

# Error handling in file operations

- In Python, file operations often involve potential pitfalls: Missing files, incorrect permissions, or unexpected data formats.
- Fortunately, various mechanisms safeguard against these errors, ensuring smooth execution and data integrity.
- Let's explore some common approaches.

---

Figure 4-16. Error handling in file operations

## Exceptions:

- Try/except blocks proactively capture potential errors like FileNotFoundError, PermissionError, or ValueError.
- Within the except block, appropriate actions are taken, such as logging the error, displaying a message, or gracefully exiting the program.

## Context managers:

- The with statement handles opening and closing the file automatically, even if exceptions occur.
- Resources are automatically released even if errors occur, preventing leaks and corruption.

## Assertions:

- Assertions in Python involve the use of statements to check whether a given condition is true, and if it is not true, an exception is raised. These statements are typically used for debugging and to ensure that certain conditions hold during the execution of the program. Assertions serve as a means to express assumptions about the correctness of the code, and they contribute to the overall reliability and quality of the software by highlighting potential issues or inconsistencies.
- If an assertion fails, an AssertionError is raised, allowing for immediate error handling.

## Validation and cleaning:

- Data validation routines scan the file content for expected formats or values.
- Invalid data can be ignored, corrected, or logged, depending on the situation.

**Logging:**

- Errors and unexpected events are logged to a file or another destination.
- This information can be invaluable for debugging and monitoring file operations.

**Benefits of error handling:**

- Focuses on what happens to the data and program under error conditions.
- Promotes cleaner and more robust code by separating error handling from core logic.
- Enhances program stability and data integrity by isolating and addressing errors effectively.

**Example 1:**

```
try:
    # Open file for reading
    with open("data.txt", "r") as f:
        # Process data in the file
        lines = f.readlines()
except FileNotFoundError:
    print("Error: File 'data.txt' not found!")
    # Handle non-existent files gracefully
except ValueError:
    print("Error: Unexpected data format in file!")
    # Handle invalid data format
```

Here, potential errors like missing files or invalid data are captured and handled within separate except blocks, ensuring the program doesn't crash unexpectedly.

**Example 2:**

```
with open("log.txt", "a") as f:
    # Write data to the log file
    f.write("Process started at 10:00 AM\n")
# File automatically closed by context manager, even if errors occur
```

The statement manages opening and closing the file, ensuring resources are released regardless of issues like the program exiting before writing.

**Example 3:**

```
def parse_data(file_path):
    with open(file_path, "r") as f:
        line = f.readline()
        # Assert expected format to prevent errors later
        assert line.startswith("ID:"), "Invalid data format!"
        # Continue processing data based on the expected format
```

An assertion checks the data format before proceeding. If the format is unexpected, an error is raised immediately, allowing for early intervention.

```
# Example: Checking if a list is not empty
```

```
my_list = []
```

**Example 4:**

```
# Using an assertion to check if the list is not empty
```

```
assert len(my_list) > 0, "The list should not be empty"  
# If the assertion fails, the following message will be displayed  
print("Assertion passed. The list is not empty.")
```

In this example, the assertion checks whether the length of the list `my_list` is greater than 0. If the list is not empty, the program continues without any issues. However, if the list is empty (as in this case), the assertion will raise an `AssertionError` with the specified error message ("The list should not be empty").

## Self evaluation: Exercise 16



IBM ICE (Innovation Centre for Education)

- **Exercise 16:** Custom Exception Handling
- **Estimated time:** 00:15 minutes
- **Aim:** Write a Python program that raises and handles a custom exception. Define a custom exception class and demonstrate its usage.
- **Learning objective:**
  - The learner will understand how to create and handle custom exceptions in Python, enhancing the ability to manage errors in code
- **Learning outcome:**
  - The learner will be able to define a custom exception class, raise the exception in specific situations, and handle it gracefully within the program

---

Figure 4-17. Self evaluation: Exercise 16

Self evaluation exercise 16 is as stated above:



IBM ICE (Innovation Centre for Education)

## Self evaluation: Exercise 17

- **Exercise 17:** File Exception Handling
- **Estimated time:** 00:15 minutes
- **Aim:** Implement a function that reads data from a file and handles file-related exceptions such as FileNotFoundError and PermissionError.
- **Learning objective:**
  - The learner will understand how to handle file-related exceptions in Python, ensuring robust file I/O operations in programs
- **Learning outcome:**
  - The learner will be able to implement file exception handling mechanisms, making their programs more resilient to errors during file operations

---

Figure 4-18. Self evaluation: Exercise 17

Self evaluation exercise 17 is as stated above:

## Self evaluation: Exercise 18



IBM ICE (Innovation Centre for Education)

- **Exercise 18:** Division by Zero Handling
- **Estimated time:** 00:10 minutes
- **Aim:** Implement a Python program that performs division and handles the division by zero exception.
- **Learning objective:**
  - The learner will understand how to handle division by zero exceptions in Python programs, ensuring robust arithmetic operations
- **Learning outcome:**
  - The learner will be able to implement exception handling to gracefully manage division by zero scenarios in their Python programs

---

Figure 4-19. Self evaluation: Exercise 18

Self evaluation exercise 18 is as stated above:



IBM ICE (Innovation Centre for Education)

## Self evaluation: Exercise 19

- **Exercise 19:** File Handling with Multiple Exceptions
- **Estimated time:** 00:15 minutes
- **Aim:** Modify a file reading program to handle both FileNotFoundError and PermissionError exceptions.
- **Learning objective:**
  - The learner will understand how to handle multiple file-related exceptions in Python programs, enhancing the robustness of file operations
- **Learning outcome:**
  - The learner will be able to implement exception handling for scenarios involving file reading, handling both FileNotFoundError and PermissionError exceptions

---

Figure 4-20. Self evaluation: Exercise 19

Self evaluation exercise 19 is as stated above:



IBM ICE (Innovation Centre for Education)

## Self evaluation: Exercise 20

- **Exercise 20:** Finally Block Usage
- **Estimated time:** 00:15 minutes
- **Aim:** Develop a program that demonstrates the use of the finally block in exception handling. Open a file and ensure it is properly closed even if exceptions occur
- **Learning objective:**
  - The learner will understand the importance of the finally block in exception handling and its role in ensuring resource cleanup, particularly file closure
- **Learning outcome:**
  - The learner will be able to implement exception handling with the finally block to guarantee proper resource cleanup, such as closing files, even in the presence of exceptions

---

Figure 4-21. Self evaluation: Exercise 20

Self evaluation exercise 20 is as stated above:

## Checkpoint (1 of 2)



IBM ICE (Innovation Centre for Education)

### Multiple choice questions:

1. The \_\_\_\_\_ mode in file handling is used to read the contents of a file.
  - a) Read
  - b) Write
  - c) Open
  - d) All of the above
  
2. In Python, the with statement is used to ensure that a file is properly\_\_\_\_\_after operations.
  - a) Closed
  - b) Opened
  - c) Executed
  - d) None of the above
  
3. The \_\_\_\_\_ method in Python overwrites the entire file content if the file already exists.
  - a) Write()
  - b) Read()
  - c) Both a and b
  - d) None of the above

Figure 4-22. Checkpoint (1 of 2)

Write your answers here:

- 1.
- 2.
- 3.



IBM ICE (Innovation Centre for Education)

## Checkpoint (2 of 2)

**Fill in the blanks:**

1. To read the contents of a file line by line, the method \_\_\_\_\_ is used.
2. The \_\_\_\_\_ keyword in Python is used to define a new exception
3. An \_\_\_\_\_ is an unusual event or runtime error that occurs during the execution of a program.
4. The \_\_\_\_\_ block in Python allows handling exceptions and executing code even if an exception occurs.

**True/False:**

1. The seek() method in Python is used to move the file cursor to a specified position.  
**True/False**
2. True or False: The else block in a try-except statement is executed only if an exception occurs. **True/False**
3. A single except block in Python can handle multiple different types of exceptions.  
**True/False**

---

Figure 4-23. Checkpoint (2 of 2)

Write your answers here:

Fill in the blanks:

- 1.
- 2.
- 3.
- 4.

True or false:

- 1.
- 2.
- 3.



## Question bank (1 of 2)

### 2 Mark questions:

- Assume the following Python code

```
mylist = [1,2,3,"4",5]
sum = 0
for i in mylist:
    sum = sum + i
print(sum)
print(mylist[5])
```

Rewrite the code to handle the exceptions raised. Print appropriate error messages wherever applicable.

- Explain the purpose of the with statement in Python file handling.
- How does the write() method behave when opening a file in 'a' (append) mode in Python?
- Define NameError with example.?

### 4 Mark questions:

- Write a Python program to:

read a file.

add backslash (\) before every double quote in the file contents.

write it to another file in the same folder.

print the contents of both the files.

For example: If the first file is 'TestFile1.txt' with text as:

Jack said, "Hello Pune". The output of the file 'TestFile2.txt' should be:

Jack said,"Hello Pune"

Figure 4-24. Question bank (1 of 2)



IBM ICE (Innovation Centre for Education)

## Question bank (2 of 2)

2. Consider a file 'courses.txt' in D Drive with the following details: Write a program to read the file and store the courses in Python variables as a:  
**Dictionary ( Sample - {0: 'Java', 1: 'Python', 2:'Javascript' 3: 'PHP'} )**
3. You have already created a Python program to implement the following in above program:  
**Now, modify your code to implement Exception handling. Print appropriate error messages wherever applicable.**
4. Add natural numbers up to n where n is taken as an input from user. Do appropriate exception handling in the code and observe the output by providing invalid input values.

### 8 Mark questions:

1. Consider a file 'rhyme.txt' in D Drive with following text:

```
Jingle bells jingle bells  
Jingle all the way  
Oh what fun it is to ride  
In a one horse open sleigh  
Jingle bells jingle bells  
Jingle all the way
```

Write a Python program to count the words in the file using a dictionary (use space as a delimiter). Find unique words and the count of their occurrences (ignoring case). Write the output in another file "words.txt" at the same location.

2. Explain Error handling in file operations with related examples.

---

Figure 4-25. Question bank (2 of 2)

# Unit summary



IBM ICE (Innovation Centre for Education)

- Having completed this unit, you should be able to:
  - Handle exceptions and errors effectively to ensure robustness in Python programs
  - Work with files and perform input/output (I/O) operations to read and write data

---

Figure 4-26. Unit summary

## Conclusion

In conclusion, this chapter provides a thorough examination of exception handling and file handling in Python, covering a spectrum from the foundational understanding of exceptions to the intricate details of working with different file types. The complexities of exception management and file handling practices are presented objectively, allowing readers to gain a comprehensive understanding of these crucial aspects of Python programming. The insights provided serve as a valuable resource for developers aiming to enhance their proficiency in writing robust and error-tolerant Python code.

---

# Unit 5. Introduction to Data & Business Analytics

## Overview

This Unit will provide an overview of Learn regular expressions and use them for text processing and pattern matching, Gain proficiency in working with modules to modularize Python code and enhance reusability.

## How you will check your progress

- Checkpoint

## References

IBM Knowledge Center



IBM ICE (Innovation Centre for Education)

## Unit objectives

- After completing this unit, you should be able to:
  - Learn regular expressions and use them for text processing and pattern matching
  - Gain proficiency in working with modules to modularize Python code and enhance reusability
- Learning outcomes:
  - Utilize regular expressions for efficient text processing and pattern matching
  - Modularize Python code using modules and libraries to enhance code maintainability and reusability

---

Figure 5-1. Unit objectives

Unit objectives and outcomes are as stated above.



# Advanced s

## Advanced s:

- Regular expressions serve as a versatile tool for intricate text manipulation with a focus on the application of regular expressions for the efficient processing of textual data and the execution of pattern matching.
- At the same time, the modularization approach is intended to significantly enhance the maintainability and reusability of Python code.
- This strategic modularization of Python code is done through the incorporation of modules and libraries.
- This helps in acquiring proficiency in the creation of structured, scalable, and maintainable Python programs.

*Figure 5-2. Advanced s*

## Regular expressions

Regular expressions, often denoted as RegEx, are utilized as a sophisticated tool for pattern matching and manipulating textual data. The definition, diverse use cases, and practical applications of Regular Expressions in pattern matching are thoroughly explored.

### Introduction to regular expression

Regular expressions are a versatile tool employed for pattern matching and text processing in Python. Their functionality lies in defining search patterns using sequences of metacharacters, facilitating the efficient retrieval and manipulation of strings. Regular Expressions are widely utilized in diverse applications, providing a mechanism to locate, extract, or replace specific patterns within textual data. Regular Expressions are widely utilized in diverse applications, providing a mechanism to locate, extract, or replace specific patterns within textual data. Regular expressions (called REs, or regexes, or regex patterns) are basically a tiny, highly specialized programming language embedded inside Python and it is available through the `re` module.

### Need for regular expressions

The importance of Regular Expressions in string manipulation tasks is underscored by their ability to offer a concise and flexible approach to handling intricate patterns in a variety of contexts. Regular expressions are useful for tasks like:

- Validating data (e.g., email addresses, phone numbers).
- Searching and extracting data from text.

- Replacing or modifying text patterns.
- Parsing structured data.

### **Use cases for regular expressions**

Regular languages, often represented by regular expressions, have various use cases in computer science and programming. They provide a concise and powerful way to define patterns, making them valuable in scenarios where matching or manipulating text based on specific rules is required. They are widely used in various applications, ranging from simple text processing tasks to complex compiler design and language recognition. Here are some common use cases of regular languages:

- **Text search and manipulation:**
  - **Use case:** Searching and manipulating text data based on specific patterns.
  - **Example:** Searching for email addresses, phone numbers, or specific keywords in a document. Replacing or formatting text based on patterns.
- **Data validation:**
  - **Use case:** Validating user inputs or data to ensure it meets specific criteria.
  - **Example:** Validating email addresses, phone numbers, credit card numbers, or ZIP codes.
- **Lexical analysis in compilers:**
  - **Use case:** Tokenizing source code during the lexical analysis phase of compiling.
  - **Example:** Identifying keywords, operators, and identifiers in a programming language.
- **URL routing in web applications:**
  - **Use case:** Defining and matching URL patterns for routing requests in web applications.
  - **Example:** Routing requests to specific controllers or views based on URL patterns.
- **Log parsing and analysis:**
  - **Use case:** Parsing and analyzing log files to extract relevant information.
  - **Example:** Identifying and categorizing log entries, extracting error messages, or tracking user activity.
- **String matching in database queries:**
  - **Use case:** Searching for or filtering data in a database based on specific patterns.
  - **Example:** Retrieving records that match a particular pattern or criteria.
- **Network protocol analysis:**
  - **Use case:** Analyzing network protocols by matching patterns in packet data.
  - **Example:** Identifying specific protocol headers or payloads in network traffic.
- **Natural Language Processing (NLP):**
  - **Use case:** Pattern matching and extraction of information from natural language text.
  - **Example:** Named entity recognition, sentiment analysis, or extracting specific phrases from text.
- **Validation of file formats:**
  - **Use case:** Validating and parsing file formats based on predefined patterns.
  - **Example:** Parsing and validating CSV, XML, or JSON files using regular expressions.
- **Input validation in forms:**
  - **Use case:** Validating user input in web forms.
  - **Example:** Ensuring that passwords meet complexity requirements, validating date formats, or checking for valid input lengths.

- **Validation of bank account details:** It is notable that banks assign an IFSC code to each of their branches, typically starting with the bank's name. Credit card numbers generally comprise 16 digits, with the initial digits indicating whether the card is MasterCard, Visa, or Rupay. Regular expressions are commonly employed for such validation tasks.

# Pattern matching and text processing (1 of 2)



IBM ICE (Innovation Centre for Education)

- Pattern matching in Python: Involves identifying specific sequences or structures within textual data.
- Components of pattern matching and text processing: Requires a comprehensive understanding of fundamental components, including:
  - Meta characters.
  - Character classes.
  - Groups.
  - Matching patterns.
- Meta characters: Essential for crafting precise patterns that facilitate sophisticated text manipulation. Includes:
  - Dot (.).
  - Asterisk (\*).
  - Question mark (?).
  - Plus (+).
  - Square brackets ([...]).
- Character classes: Provide a mechanism for defining sets of characters, adding nuance to matching criteria.
- Groups: Enable pattern grouping and facilitate the extraction of specific text segments. Matching patterns with regular expressions.

Figure 5-3. Pattern matching and text processing (1 of 2)

Involves the application of regular expressions. Empowers developers with the capability to execute intricate and potent text processing operations.

Versatile toolkit: Serves as the cornerstone for adeptly navigating and manipulating textual data within the Python programming paradigm.

## Meta characters

Meta characters are special characters with a predefined meaning in regular expressions. They are used to define patterns for matching specific characters or sequences of characters.

Metacharacter	Description
[]	A collection of characters
\	A specific sequence is signaled (can also be used to escape special characters)
.	Any character (except newline character)
^	Begins with
\$	Finishes with
*	Zero or more occurrences
+	One or more occurrences
?	Zero or one occurrences
{}	Exactly the specified number of occurrences
	Either or
()	Capture and group

Table 1: List of common metacharacters  
[\[https://www.tutorialsfreak.com/python-tutorial/python-regex-metacharacters\]](https://www.tutorialsfreak.com/python-tutorial/python-regex-metacharacters)

### Common meta characters:

- . - Match any character

```
import re
pattern = r'a.b'
text = "a1b a2b a3b"
matches = re.findall(pattern, text)
print("Matches:", matches)
# Matches: ['a1b', 'a2b', 'a3b']
```

- \* - Match zero or more occurrences

```
import re
pattern = r'go*gle'
text = "ggle gooole google"
matches = re.findall(pattern, text)
print("Matches:", matches)
#Matches: ['ggle', 'gooole', 'google']
```

- + - Match one or more occurrences

```
import re
pattern = r'go+gle'
text = "ggle gooole google"
matches = re.findall(pattern, text)
print("Matches:", matches)
#Matches: ['gooole', 'google']
```

- ? - Match zero or one occurrence

```
import re
```

```

pattern = r'colou?r'
text = "color colour"
matches = re.findall(pattern, text)
print("Matches:", matches)
#Matches: ['color', 'colour']

•^ - Match the start of a string

import re
pattern = r'^start'
text = "start with something"
match = re.match(pattern, text)
if match:
    print("Match found:", match.group())
else:
    print("No match")

#Match found: start

•$ - Match the end of a string

import re
pattern = r'end$'
text = "something at the end"
match = re.search(pattern, text)
if match:
    print("Match found:", match.group())
else:
    print("No match")

#Match found: end

•[] - Match any character within the brackets

import re
pattern = r'[aeiou]'
text = "Hello"
matches = re.findall(pattern, text)
print("Matches:", matches)
# Matches: ['e', 'o']

•| - Match either the expression before or after the pipe

import re
pattern = r'apple|banana'
text = "I like apples and bananas."
matches = re.findall(pattern, text)
print("Matches:", matches)
# Matches: ['apple', 'banana']

```

## Character classes

Character classes in regular expressions are employed to define sets of characters that are considered acceptable matches for individual characters within a pattern.

### Common character classes:

- \d - Match any digit

```
import re
pattern = r'\d+'
text = "The price is $10 for product A and $20 for product B."
matches = re.findall(pattern, text)
print("Matches:", matches)
# Matches: ['10', '20']
```

- \D - Match any non-digit

```
import re
pattern = r'\D+'
text = "The price is $10 for product A and $20 for product B."
matches = re.findall(pattern, text)
print("Matches:", matches)
# Matches: ['The price is $', ' for product A and $', ' for product B.']
```

- \w - Match any word character (alphanumeric and underscore)

```
import re
pattern = r'\w+'
text = "This is_an example."
matches = re.findall(pattern, text)
print("Matches:", matches)
# Matches: ['This', 'is_an', 'example']
```

- \W - Match any non-word character

```
import re
pattern = r'\W+'
text = "This is_an example."
matches = re.findall(pattern, text)
print("Matches:", matches)
# Matches: [' ', ' ', '.']
```

- \s - Match any whitespace character

```
import re
pattern = r'\s+'
text = "Whitespace\tTabbed space\nNewline"
matches = re.findall(pattern, text)
print("Matches:", matches)
# Matches: ['\t', ' ', '\n']
```

- \S - Match any non-whitespace character

```
import re
```

```

pattern = r'\S+'
text = "Whitespace\tTabbed space\nNewline"
matches = re.findall(pattern, text)
print("Matches:", matches)
# Matches: ['Whitespace', 'Tabbed', 'space', 'Newline']
•\A: Represents the start of the string, It matches only at the beginning of
the string.

import re
# Example text
text = "Python is a powerful programming language."
# Pattern to match "Python" only at the beginning of the string
pattern = r'\APython'
# Find a match
match = re.search(pattern, text)
# Check if a match is found
if match:
    print("Match found at the beginning of the string:", match.group())
else:
    print("No match found.")
•\Z: Represents the end of the string, It matches only at the end of the
string.

import re
# Example text
text = "Python is a powerful programming language."
# Pattern to match "language." only at the end of the string
pattern = r'language\Z'
# Find a match
match = re.search(pattern, text)
# Check if a match is found
if match:
    print("Match found at the end of the string:", match.group())
else:
    print("No match found.")
•\b: Represents a word boundary, It matches at a position where a word
character (like a letter or digit) is followed or preceded by a non-word
character (like a space or punctuation).

import re
# Example text
text = "hello word helloworld"
# Pattern to match "word" as a whole word
pattern = r'\bword\b'

```

```
# Find all matches
matches = re.findall(pattern, text)
# Print the matches
print("Matches:", matches) # Output: ['word']

•\B: Represents a non-word boundary. It matches at a position where a word character is followed or preceded by another word character, or where a non-word character is followed or preceded by another non-word character.

import re

# Example text
text = "helloworld hello word hello_word"
# Pattern to match "word" within other characters
pattern = r'\Bword\B'

# Find all matches
matches = re.findall(pattern, text)
# Print the matches
print("Matches:", matches) # Output: ['word']
```

# Pattern matching and text processing (2 of 2)



IBM ICE (Innovation Centre for Education)

- Groups:

- In Python's re-module, groups are utilized to create subpatterns within a larger regular expression pattern. These subpatterns can be captured, repeated, or used to apply quantifiers.
- The creation of groups involves enclosing a portion of the pattern in parentheses () .
- For instance, (ab)+ represents a group capturing the sequence "ab" that can occur one or more times. The information captured by groups can be retrieved using the group() method after a successful match.
- Groups contribute to the organization and clarity of regular expression patterns, aiding in the creation of more sophisticated and flexible matching criteria.
- For example, the regular expression (cat) creates a single group having the letters 'c', 'a', and 't'.
- For example, in a real-world case, you want to capture emails and phone numbers, So you should write two groups, the first one will search email, and the second one will search phone numbers.

*Figure 5-4. Pattern matching and text processing (2 of 2)*

### Example of creating groups

In the given example, a regular expression is used to search for a pattern in a text string using the re-module in Python.

```
import re
pattern= r'(\d+)\s(apple|banana)'

text = "I have 3 apples."
match = re.search(pattern, text)

if match:
    print("Match found:", match.group())
    print("Fruit:", match.group(2))
    print("Quantity:", match.group(1))

else:
    print("No match")

# Match found: 3 apple
# Fruit: apple
# Quantity: 3
```

### Explanation:

- **(\d+)**: This is the first group in the pattern. It captures one or more digits (\d+), representing the quantity of fruits. The parentheses (...) create a capturing group to extract the matched quantity later. The \d means match any digit from 0 to 9 in a target string. Then the + metacharacter indicates number can contain a minimum of 1 or maximum any number of digits.
- **\s**: Matches a whitespace character (space).
- **(apple|banana)**: This is the second group in the pattern. It captures either "apple" or "banana". The | is the alternation operator, allowing either of the specified options to match.
- **re. search(pattern, text)**: Searches for the pattern in the given text using the re. search() function.
- **match.group()**: Returns the entire matched string.
- **match.group(1)**: Returns the content of the first capturing group (quantity).
- **match.group(2)**: Returns the content of the second capturing group (fruit).

### Matching patterns with re-module

In Python's re-module, the operations of searching and matching involve the identification of specific patterns within a given string. These operations are executed through functions such as re. search() and re.match(), contributing to effective pattern-related tasks in text processing.

### Search operation

The re.search() function is employed to search for a specified pattern within a given string. If a match is found, a match object is returned, representing the first occurrence of the pattern in the string.

#### Example 1: Email extraction

Scenario: In the given text, the email address "support@example.com" is searched for.

```
import re

text = "Contact support@example.com for assistance or info@example.org for inquiries."

# Email search operation
email_pattern = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'
matches = re.search(email_pattern, text)

if matches:
    print("Email found:", matches.group())

```

Output: The email address "support@example.com" is identified and printed as the output.

**Let's break down the components of the email\_pattern:**

Pattern	Description
\b	Word boundary anchor. Ensures the pattern matches as a whole word, not part of a larger word.
[A-Za-z0-9._%+-]+	Matches the local part of the email address (before the @ symbol).
[A-Za-z0-9]	Matches any uppercase letter, lowercase letter, or digit.
._%+-	Allows the characters '.', '_', '%', '+', and '-' in the local part.
+	Ensures there is at least one or more of the preceding characters in the local part.
@	Matches the @ symbol, which separates the local part from the domain part.
[A-Za-z0-9.-]+	Matches the domain part of the email address (after the @ symbol).
[A-Za-z0-9]	Matches any uppercase letter, lowercase letter, or digit.
.-	Allows the characters '.' and '-' in the domain part.
+	Ensures there is at least one or more of the preceding characters in the domain part.
.	Matches the dot (.) character, which separates the domain from the top-level domain (TLD).
[A-Z a-z]{2,}	
[A-Z a-z]	
{2,}	Specifies there must be at least two or more characters in the TLD.
\b	Word boundary anchor, similar to the one at the beginning.

Table 2: Components of the email\_pattern

This regular expression pattern is crafted to match a wide variety of valid email addresses while adhering to common email address conventions. It's important to note that email address validation can be complex, and this pattern provides a basic level of validation. There are more intricate patterns to handle all possible cases, but this one is suitable for many practical situations.

#### Here are some examples that can be validated by this program:

- The valid email with an alphanumeric local part and common TLD: john.doe@example.com
- Email with a plus sign in the local part: jane+doe@example.co.uk
- Email with an underscore in the local part: alice\_smith@example.net
- Email with a percent sign in the local part: bob%johnson@example.org
- Email with a hyphen in the domain part: peter.parker@web-site.com
- Email with a dot in the TLD: mary\_jones@example.co.in
- Email with a combination of uppercase and lowercase letters: CaseSensitive@example.net
- An email with a longer TLD: user@example.coffee

#### Example 2: URL identification

**Scenario:** Within the provided text, the URL "https://www.example.com" is sought.

```
import re

text = "Visit our website at https://www.example.com for more information."
# URL search operation
url_pattern = r'https?://(?:www\.)?[a-zA-Z0-9-]+\.[a-zA-Z]{2,}\''
matches = re.search(url_pattern, text)
```

```
if matches:
    print("URL found:", matches.group())
```

This example demonstrates using `re.search()` to find a URL within the given text, providing more flexibility in handling variations and placement of URLs within the text.

**Output:** The URL "https://www.example.com" is detected and presented as the output.

**Let's break down the components of the url\_pattern:**

- **https?:** Matches either "http" or "https". The? makes the "s" character optional.
- **::/:** Matches the colon and double forward slashes that are part of the URL scheme.
- **(?:www\.)?:** This is a non-capturing group (?: ...) that makes the "www." part of the URL optional.
- **www\.:** Matches the literal "www."
- **[a-zA-Z0-9-]+:** Matches the domain name, allowing for letters (both uppercase and lowercase), digits, and hyphens.
- **+:** Ensures that there is at least one or more characters in the domain name.
- **\.:** Matches the dot (.) that separates the domain name from the top-level domain (TLD).
- **[a-zA-Z]{2,}:** Matches the TLD, requiring at least two or more letters (both uppercase and lowercase).

### Match operation

The `re.match()` function is similar to `re.search()` but specifically checks if the pattern matches at the beginning of the string. It returns a match object if the pattern is found at the start; otherwise, it returns None.

### Example-1: Validating password format

**Scenario:** The match operation is applied to check if a given password follows specific criteria, such as containing at least one uppercase letter, one lowercase letter, and one digit.

```
import re

password = "SecurePwd123"
# Password format match operation
password_pattern = r'^(?=.*[A-Z])(?=.*[a-z])(?=.*\d)[A-Za-z\d]{8,}$'
match = re.match(password_pattern, password)
if match:
    print("Valid password format.")
```

**Output:** The password "SecurePwd123" is evaluated and determined to have a valid format, meeting the specified criteria.

### Pattern explanation:

- **^:** Asserts the start of the string.
- **(?=.\*[A-Z]):** Positive lookahead assertion for at least one uppercase letter.
- **(?=.\*[a-z]):** Positive lookahead assertion for at least one lowercase letter.
- **(?=.\*\d):** Positive lookahead assertion for at least one digit.
- **[A-Za-z\d]{8,}:** Matches any combination of at least 8 characters consisting of letters (uppercase or lowercase) and digits.

### Note

A positive lookahead assertion in a regular expression is a non-consuming assertion, meaning it doesn't consume any characters in the string during the matching process. It checks whether a particular subpattern can match at a specific position in the string without actually moving the "cursor" forward in the string.

**Let's break down the positive lookahead assertion for at least one digit ((?=.\*\d)):**

- **(?= ... ):** This denotes the positive lookahead assertion.
- **.\*:** This part matches any sequence of characters (except for newline characters).
- **\d:** This matches a digit.

The positive lookahead assertion ensures that somewhere in the string (starting from the current position), there is at least one digit (\d). However, it doesn't consume any characters in the matching process; it merely checks for the presence of the digit.

### **Reasons for using re.match(), but not re.search()**

Password validation typically involves checking the entire password against specific criteria. In this case, it's essential that the entire password adheres to the specified pattern from the beginning. Using `re.match()` ensures that the pattern is matched from the start of the password.

#### **Pattern at the beginning:**

- Password validation typically involves checking the entire password against specific criteria.
- In this case, it's essential that the entire password adheres to the specified pattern from the beginning.
- Using `re.match()` ensures that the pattern is matched from the start of the password.

#### **No need to search the entire string:**

- Since the password validation pattern requires specific conditions to be met from the beginning, there's no need to search the entire string for a match.
- Using `re.search()` might be less efficient in this scenario because it would continue searching even if the initial part of the string doesn't match the pattern.

### **Example-2: Validating product code**

The product code should adhere to the format "ABC-1234," where "ABC" represents a three-letter code, and "1234" represents a four-digit number. Here's a program to check the validity of a product code:

```
import re

product_code = 'XYZ-5678'

# Custom product code pattern
product_code_pattern = r'^[A-Z]{3}-\d{4}$'

# Using re.match() to validate the product code format
match = re.match(product_code_pattern, product_code)

if match:
    print("Valid product code format.")
else:
    print("Invalid product code format.")
```

#### **In this example:**

- The `product_code` variable is set to 'XYZ-5678' as an example. Replace it with an actual product code to test.
- The `product_code_pattern` is a regular expression pattern designed to match a three-letter code followed by a hyphen and a four-digit number.
- The `re.match()` function checks if the entire `product_code` conforms to the specified pattern.
- If the pattern is matched from the beginning to the end of the string, the output will be "Valid product code format"; otherwise, it will be "Invalid product code format."



# Regex functions and methods

- Regex functions and methods:
  - In Python, the `re`-module provides functions and methods for working with regular expressions.

*Figure 5-5. Regex functions and methods*

**Here are some commonly used functions and methods from the `re` module:**

- **findall()** The `re.findall(pattern, string, flags=0)` function is used to find all occurrences of a pattern in a given string and return them as a list. Here's a breakdown of its parameters:
  - **pattern:** The regular expression pattern to search for in the string.
  - **string:** The input string where the search is performed
  - **flags:** Optional flags to modify the behavior of the pattern matching.

**Here's an example demonstrating the use of `re.findall`:**

```
import re

text = "The price of apples is $1.25, and the price of bananas is $0.75."
# Find all occurrences of prices in the text
prices = re.findall(r'\$\d+\.\d+', text)
print("Prices found:", prices)
```

In this example, the regular expression `\$\d+\.\d+` is used to match currency values in the text (e.g., \$1.25). The `re.findall` function returns a list containing all occurrences of the specified pattern in the given text.

Output: Prices found: ['\$1.25', '\$0.75']

- **Sub:** In Python's `re` module, the `re.sub(pattern, repl, string, count=0, flags=0)` function is used for string substitution based on a regular expression pattern. Here's an explanation of its parameters:

- **pattern:** The regular expression pattern to search for in the string.
- **repl:** The replacement string or a function for substitution.
- **string:** The input string where the substitutions are performed.
- **count:** An optional parameter specifying the maximum number of substitutions to make. If omitted or set to 0, all occurrences are replaced.
- **flags:** Optional flags to modify the behavior of the pattern matching.

**Here's an example demonstrating the use of re. sub:**

```
import re

text = "The cat and the hat are sitting on the mat."
# Replace occurrences of "cat" with "dog"
modified_text = re.sub(r'cat', 'dog', text)
print("Modified text:", modified_text)
```

**Output:** Modified text: The dog and the hat are sitting on the mat.

In this example, the re. sub-function is used to replace occurrences of the word "cat" with "dog" in the given text.

- **Finditer:** The re. finditer(pattern, string, flags=0) function in Python's re-module is used to find all occurrences of a pattern in a given string and return them as an iterator of match objects. Each match object provides information about the matched substring, including its start and end positions. Here's an explanation of its parameters:

- **pattern:** The regular expression pattern to search for in the string.
- **string:** The input string where the search is performed.
- **flags:** Optional flags to modify the behavior of the pattern matching.

**Here's an example demonstrating the use of re. finditer:**

```
import re

text = "The cat and the hat are sitting on the mat."
# Find all occurrences of "at" in the text
matches_iter = re.finditer(r'at', text)
# Iterate over match objects
for match in matches_iter:
    start_pos = match.start()
    end_pos = match.end()
    matched_text = match.group()
    print(f"Found '{matched_text}' at positions {start_pos}-{end_pos}.")
```

**Output:**

Found 'at' at positions 4-6.  
 Found 'at' at positions 12-14.  
 Found 'at' at positions 24-26.  
 Found 'at' at positions 37-39.

In this example, the re. finditer function is used to find all occurrences of the pattern "at" in the given text. The resulting iterator (matches\_iter) is then used to iterate over match objects, and information about each match is printed, including the matched text and its start and end positions in the original string.

- **Compile:** In Python's re module, the `re.compile(pattern, flags=0)` function is used to compile a regular expression pattern into a regex object. The resulting regex object can then be used for various pattern-matching operations. Here's an explanation of its parameters:
  - **pattern:** The regular expression pattern to be compiled.
  - **flags:** Optional flags to modify the behavior of the pattern matching.

**Here's an example demonstrating the use of `re.compile`:**

```
import re

# Define a regular expression pattern
pattern = r'\b\w+\b'

# Compile the pattern into a regex object
compiled_pattern = re.compile(pattern)

# Test the compiled pattern with a string
text = "This is a sample sentence."

matches = compiled_pattern.findall(text)

print("Matches:", matches)
Output: Matches: ['This', 'is', 'a', 'sample', 'sentence']
```

In this example, the regular expression pattern `\b\w+\b` (matching whole words) is compiled into a regex object using `re.compile`. The resulting `compiled_pattern` is then used to find all matches in the given text.

Compiling the pattern is optional, but it can be beneficial in scenarios where the same pattern is used multiple times. The compiled pattern can be reused, potentially improving performance. Additionally, the compiled pattern object has methods like `findall`, `search`, and others for pattern matching.

### Extracting data from text

Extracting data from text in Python often involves using regular expressions (re-module) or other string manipulation techniques. Here are some examples demonstrating how to extract data from text using regular expressions:

#### Example 1: Extracting dates

```
import re

text = "Meeting scheduled for 2023-12-16. Event on 2023-12-20."

# Define a regex pattern for extracting dates
date_pattern = r'\d{4}-\d{2}-\d{2}'

# Use re. findall to extract dates from the text
dates = re.findall(date_pattern, text)

print("Extracted Dates:", dates)
Extracted Dates: ['2023-12-16', '2023-12-20']
```

#### Patter explanation:

- **\d{4}:**  Matches the year part of the date (four digits).
- **-:**  Matches the hyphen separator between the year and the month.
- **\d{2}:**  Matches the month part of the date (two digits).
- **-:**  Matches the hyphen separator between the month and the day.
- **\d{2}:**  Matches the day part of the date (two digits).

So, the entire pattern `\d{4}-\d{2}-\d{2}` ensures that the string matches the "YYYY-MM-DD" date format, where "YYYY" is the four-digit year, "MM" is the two-digit month, and "DD" is the two-digit day. For example, "2023-12-16" would be a valid match for this pattern.

### Example 2: Extracting email addresses

```
import re
text = "Contact us at support@example.com or info@company.com."
# Define a regex pattern for extracting email addresses
email_pattern = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'
# Use re.findall to extract email addresses from the text
emails = re.findall(email_pattern, text)
print("Extracted Email Addresses:", emails)
Output: Extracted Email Addresses: ['support@example.com', 'info@company.com']
```

#### Pattern explanation:

Pattern	Description
\b	This is a word boundary anchor, ensuring that the match occurs at the beginning or end of a word. It helps avoid partial matches within longer words.
[A-Za-z0-9._%+-]+	This part matches the local part of the email address (before the @ symbol). It allows for one or more occurrences of letters (both uppercase and lowercase), digits, and certain special characters such as period (.), underscore (_), percent (%), plus (+), and hyphen (-).
@	This literal character matches the at symbol, which separates the local part from the domain part in an email address.
[A-Za-z0-9.-]+	This part matches the domain part of the email address (after the @ symbol). It allows for one or more occurrences of letters (both uppercase and lowercase), digits, and certain special characters such as period (.), hyphen (-), and dot (.)
.	This literal character matches the dot (.) that separates the main domain from the top-level domain (TLD).
[A-Z a-z]{2,}	
\b	Another word boundary anchor, ensuring that the match ends at the end of the email address.

Table 3: Pattern explanation

### Example 3: Extracting phone numbers

```
import re
text = "Contact us at +1 (555) 123-4567 or 555-987-6543."
# Define a regex pattern for extracting phone numbers
phone_pattern = r'(\+\d{1,2}\s?)?((\d{3})\s?\|\d{3}[-.\s?]\d{3}[-.\s?]\d{4})'
# Use re.findall to extract phone numbers from the text
phones = re.findall(phone_pattern, text)
# Format the extracted phone numbers
formatted_phones = ['.'.join(phone) for phone in phones]
```

```
print("Extracted Phone Numbers:", formatted_phones)
```

Output: Extracted Phone Numbers: ['+1 (555) 123-4567', '555-987-6543']

#### Pattern explanation:

- **(+\d{1,2}\s?)?**: This part matches an optional international dialing code (country code) at the beginning of the phone number. It includes a plus sign (+), followed by one or two digits, and an optional space. The entire group is made optional by the? Quantifier.
- **(\(\d{3}\)\s?\|\d{3}[-.\s]?)**: This part matches the area code of the phone number. It allows for two formats: enclosed in parentheses (e.g., (123)) with an optional space, or just three digits followed by an optional space, hyphen, period, or any whitespace character.
- **\d{3}[-.\s]?**: This part matches the first three digits of the phone number, followed by an optional space, hyphen, period, or any whitespace character.
- **\d{4}**: This part matches the last four digits of the phone number.



# Modules and libraries

- Modules and libraries:

- A module in Python is a file containing Python definitions and statements. The file name is the module name with the suffix .py. Modules can define functions, classes, and variables.
- They allow code to be organized, reused, and separated into logical components. Libraries are collections of modules that encapsulate specific functionalities, and they are frequently pre-built, offering a suite of tools, functions, and classes for various purposes.
- A rich ecosystem of libraries exists in Python, encompassing both built-in and third-party options, catering to diverse needs.
- Examples include NumPy for numerical computing, pandas for data manipulation, and requests for handling HTTP requests.
- The importation of a library enables the access of its modules and utilization of the functionalities it offers.
- For instance, the importing of NumPy, as demonstrated by the statement `import numpy`, permits the utilization of the functions and classes inherent in the NumPy library.

*Figure 5-6. Modules and libraries*

## Creating and using the module

In Python, the creation and utilization of modules involve defining functions, classes, or variables within a separate file. The creation and utilization of modules serve a crucial role in enhancing code organization, reusability, and maintainability. The following aspects underscore the need for creating and using modules:

- **Code organization:** Creating modules allows developers to organize code logically by grouping related functions, classes, or variables into separate files. This modular structure makes it easier to locate and understand specific components of a program.
- **Reusability:** Modules promote code reuse, as functions or classes defined in one Module can be imported and utilized in multiple scripts. This eliminates the need to duplicate code, reducing redundancy and ensuring consistency across projects.
- **Encapsulation:** Modules provide a means of encapsulating code, allowing developers to hide the implementation details and expose only the necessary interfaces. This abstraction enhances code maintainability and reduces the risk of unintended interactions between different parts of the program.
- **Collaborative development:** In collaborative development environments, modules facilitate team collaboration by allowing developers to work on different modules independently. This modular approach minimizes conflicts and streamlines the integration of various components.
- **Readability and scalability:** As projects grow in complexity, creating modules helps maintain code readability. Breaking down a large codebase into modular components makes it easier for developers to comprehend, debug, and extend the functionality without overwhelming complexity.

- **Facilitating testing:** Modules enable more effective unit testing, as individual components can be tested in isolation. This modular testing approach simplifies the identification and resolution of issues, contributing to a more robust and reliable codebase.

## Writing your own modules

In Python, the creation of custom modules involves defining functions, classes, or variables within a separate file. These modules allow for code organization and reusability.

### The following steps showcase how modules are written:

#### Create a python file:

A file named calculator.py is created to house mathematical functions:

```
# calculator.py
def addition(x, y):
    return x + y
def subtraction(x, y):
    return x - y
```

#### Renaming a module:

Renaming a module using an alias is a common practice in Python, especially when dealing with long module names or when you want to make the code more readable. Here's an example with code explanation:

Suppose we have a module named `very_long_module_name.py`, and we want to import it with a shorter name `short_module`. We can achieve this by using the `as` keyword to create an alias:

```
# Importing the module with an alias
import very_long_module_name as short_module
# Now we can use the functions/classes from the module using the alias
result = short_module.some_function()
```

This makes the code more concise and readable, especially if the original module name is lengthy or difficult to remember. Using aliases for module names can also prevent naming conflicts and make the code more maintainable.

**Define functions or classes:** Inside the module file, functions, classes, or variables are defined as per the requirements. The example above defines a simple greet function.

#### Using the dir() Function-

The `dir()` function is a built-in Python function that returns a sorted list of names defined in a module, namespace, or object. It can be used to list all the function names, variable names, and other attributes available in a module.

#### Example:

Suppose we have a module named `example_module.py` with the following content:

```
# example_module.py
def function1():
    pass
def function2():
    pass
variable1 = 10
variable2 = "Hello"
```

We can use the `dir()` function to list all the names defined in the `example_module`:

```

import example_module
# List all names in the example_module
print(dir(example_module))
Output: ['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', 'function1', 'function2', 'variable1',
'variable2']

```

**Usage in another script:** The Module can be imported and used in other Python scripts, facilitating code reuse. This is covered in the next section.

### Importing modules

By using the import statement, the main script gains access to the functionalities encapsulated within the Module, promoting code modularity and reusability. In another script, the functions from the calculator module are imported and applied:

```

# main_script.py
import calculator
result_sum = calculator.addition(5, 3)
result_difference = calculator.subtraction(8, 3)
print("Result of addition:", result_sum)
print("Result of subtraction:", result_difference)

```

### Example:

In Python, there isn't a strict concept of private modifiers as seen in some other programming languages (like Java with private access modifier). However, developers often use a convention to indicate that certain attributes or functions are intended for internal use within a module and should not be accessed directly from outside the Module. By convention, names starting with a single leading underscore (\_) are considered internal and are not intended for public use.

Let's create an example illustrating this convention:

```

# my_module.py
# Public function
def greet(name):
    return f"Hello, {name}!"
# Private function (convention with a leading underscore)
def _secret_greet(name):
    return f"Shh! Secret hello, {name}!"

```

### main\_script.py:

```

import my_module
# Using the public function
result_public = my_module.greet("Alice")
print(result_public)
# Attempting to use the private function (convention)
# Note: It's not truly private, but it's a convention to indicate it's for
internal use
result_private = my_module._secret_greet("Bob")
print(result_private)

```

In this example, `greet` is a public function intended for external use, while `_secret_greet` is marked with a leading underscore, indicating that it's intended for internal use within the `my_module`. While Python doesn't enforce true privacy, adhering to naming conventions helps developers understand which parts of a module are considered external and internal.



# Standard library modules

- Standard library modules:
  - In Python programming, an array of standard library modules exists, among which are the math, datetime, os, and sys modules.
  - These modules, pre-constructed and readily available, encompass a spectrum of functionalities designed to augment programming tasks.
  - The math module facilitates mathematical operations, datetime facilitates handling date and time, os enables interaction with the operating system, and sys provides access to interpreter-specific functionalities.
  - This collection of standard library modules forms an integral part of Python's versatility, offering essential tools for diverse programming scenarios.

*Figure 5-7. Standard library modules*

## Exploring built-in modules (math, DateTime)

### Math module

In Python, the math module is utilized for performing mathematical operations. The Module encompasses a variety of functions that facilitate tasks such as square root calculation, trigonometric operations, and exponentiation.

For example, the square root function within the math module can be applied as follows:

```
# Using the math module
import math
result_square_root = math.sqrt(25)
print("The square root is:", result_square_root)
```

In this instance, the math module is seamlessly incorporated into the script using the import statement, and the sqrt function is employed to calculate the square root of 25. The result is then printed, demonstrating the ease of integrating mathematical functionalities provided by the math module.

The math module in Python provides a set of mathematical functions and constants that enable developers to perform various mathematical operations. Here are some commonly used functions and constants from the math module:

### Mathematical functions:

Function	Description	Example
math.sqrt(x)	Returns the square root of x.	import math result = math.sqrt(25) print(result) # Output: 5.0
math.exp(x)	Returns the exponential of x.	import math result = math.exp(2) print(result) # Output: 7.3890560989306495
math.log(x[, base])	Returns the natural logarithm of x. Optionally, the base can be specified.	import math result = math.log(10) print(result) # Output: 2.302585092994046
math.sin(x)	Returns the sine of x (in radians).	import math result_sin = math.sin(math.pi/2) print(result_sin) # Output: 1.0
math.cos(x)	Returns the cosine of x (in radians).	import math result_cos = math.cos(math.pi) print(result_cos) # Output: -1.0
math.tan(x)	Returns the tangent of x (in radians).	import math result_tan = math.tan(math.pi/4) print(result_tan) # Output: 0.9999999999999999
math.pi	Mathematical constant representing the ratio of a circle's circumference to its diameter.	import math print(math.pi) # Output: 3.141592653589793
math.e	Mathematical constant representing the base of the natural logarithm.	import math print(math.e) # Output: 2.718281828459045
math.inf	Represents positive infinity.	import math print(math.inf)
math.nan	Represents NaN (Not a Number).	import math print(math.nan)

Table 4: Mathematical functions

```

import math
result = math.sqrt(25)
print(result) # Output: 5.0

import math
result = math.exp(2)
print(result) # Output: 7.3890560989306495

import math
result = math.log(10)
print(result) # Output: 2.302585092994046

import math
result_sin = math.sin(math.pi/2)
result_cos = math.cos(math.pi)
result_tan = math.tan(math.pi/4)
print(result_sin, result_cos, result_tan) # Output: 1.0 -1.0 0.9999999999999999

import math
print(math.pi) # Output: 3.141592653589793

import math
print(math.e) # Output: 2.718281828459045

import math
print(math.inf)

import math
print(math.nan)

```



These are just a few examples, and the math module provides a wide range of functions and constants for various mathematical computations. The documentation for the math module is a valuable resource for exploring all available functionalities: <https://docs.python.org/3/library/math.html>.

### **Datetime module:**

In Python, the datetime Module is employed for managing date and time information. This Module offers functionalities to create, manipulate, and format dates and times.

For instance, the datetime Module can be utilized to retrieve the current date and time:

```

# Using the datetime Module
import datetime

current_time = datetime.datetime.now()

print("The current date and time is:", current_time)

```

In this example, the Date Time Module is seamlessly integrated into the script through the import statement. The Date Time.now() function is then applied to obtain the current date and time, which is subsequently printed. This demonstrates the straightforward incorporation of date and time functionalities provided by the datetime Module in Python.

### **Datetime functions:**

The datetime Module in Python provides classes for working with dates and times. These functions and methods are commonly used when working with dates and times using the datetime Module in Python.

They facilitate various operations such as obtaining the current date and time, parsing date strings, formatting dates, and performing arithmetic with time intervals. Here are eight frequently used functions and methods from the datetime Module:

### **Working with OS and sys modules**

#### **Os module**

In Python, the os module is employed for interacting with the operating system, allowing developers to perform tasks related to file and directory manipulation, as well as system-level operations. Here, we present a description along with examples of using the os module.

#### **Example 1: Getting the current working directory**

```
# The os module is imported
import os

# The current working directory is obtained
current_directory = os.getcwd()

print("The current directory is:", current_directory)
```

In this example, the os module is effortlessly incorporated to retrieve the current working directory using the getcwd function.

#### **Example 2: Listing files in a directory**

```
# The os module is used to list files in a directory
import os

# Specify the directory path
directory_path = "/path/to/directory"

# List files in the directory
files = os.listdir(directory_path)

print("Files in the directory:", files)
```

Here, the os module is applied to list files in a specified directory using the listdir function.

#### **Example 3: Creating a new directory**

```
# The os module is utilized to create a new directory
import os

# Specify the new directory path
new_directory_path = "/path/to/new_directory"

# Create the new directory
os.mkdir(new_directory_path)

print("New directory created at:", new_directory_path)
```

In this instance, the os module is seamlessly integrated to create a new directory using the mkdir function.

#### **Example 4: Checking if a path exists**

```
# The os module is used to check if a path exists
import os

# Specify the path to check
path_to_check = "/path/to/somefile.txt"

# Check if the path exists
if os.path.exists(path_to_check):
```

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

In this example, the `os.path.exists()` function is employed to determine if a specified path exists.

### Example 5: Renaming a file

```
# The os module is utilized to rename a file  
import os  
# Specify the current and new file names  
current_file_name = "old_file.txt"  
new_file_name = "new_file.txt"  
# Rename the file  
os.rename(current_file_name, new_file_name)  
print(f"The file {current_file_name} has been renamed to {new_file_name}.")
```

Here, the `os.rename()` function is used to rename a file.

### Example 6: Deleting a file

```
# The os module is applied to delete a file  
import os  
# Specify the file to delete  
file_to_delete = "file_to_delete.txt"  
# Delete the file  
os.remove(file_to_delete)  
print(f"The file {file_to_delete} has been deleted.")
```

In this example, the `os.remove()` function is used to delete a specified file.

These examples demonstrate the diverse capabilities of the `os` module in Python, showcasing its utility for file and directory management as well as checking the existence of paths.

Method	Description
<code>os.walk()</code>	Generate all file names under the directory tree.
<code>os.chdir('dirname')</code>	Change directory to 'dirname'.
<code>os.rmdir('dirname')</code>	Delete directory.
<code>os.removedirs('dirname')</code>	Delete multi-level directory.
<code>os.chmod()</code>	Change directory permissions.
<code>os.path.basename('path/filename')</code>	Remove directory <code>path</code> and return the file name.
<code>os.path.dirname('path/filename')</code>	Remove file name and return the directory <code>path</code> .
<code>os.path.join(path1[, path2[,...]])</code>	Combine parts of the <code>path</code> into one <code>path</code> name.
<code>os.path.split('path')</code>	Return a tuple of (directory name, base name).
<code>os.path.splitext()</code>	Return a tuple of (filename, extension).
<code>os.path.getatime()</code>	Return the last access <code>time</code> .
<code>os.path.getctime()</code>	Return the creation <code>time</code> .
<code>os.path.getmtime()</code>	Return the last modification <code>time</code> .
<code>os.path.getsize()</code>	Return the size of the file.
<code>os.path.exists()</code>	Check if the <code>path</code> exists.
<code>os.path.isabs()</code>	Check if the <code>path</code> is absolute.
<code>os.path.isdir()</code>	Check if it is a directory.
<code>os.path.isfile()</code>	Check if it is a file.

## sys module

The `sys` module in Python is employed for interacting with the Python interpreter and provides access to various system-specific parameters and functions. Here, we'll explore the `sys` module with examples.

Description
List of command-line arguments passed to a Python script.
Exit the Python interpreter with optional exit status.
Name of the platform (e.g., 'linux', 'win32', 'darwin').
Version of the Python interpreter.
List of directories to search for module files.
Dictionary mapping module names to modules which have been loaded.
Standard output stream.
Standard input stream.
Standard error stream.
Return information about the current exception being handled.
Return the size of an object in bytes.
Return the reference count of an object.
Return the name of the filesystem encoding.
Set the maximum depth of the Python interpreter stack.
Return the current maximum recursion depth limit.
Return the default string encoding used by the Unicode implementation.
Encoding used for standard input.
Encoding used for standard output.
Encoding used for standard error.

### Example 1: Retrieving command line arguments

```
# The sys module is imported
import sys

# Command line arguments are obtained
command_line_arguments = sys.argv

print("Command line arguments:", command_line_arguments)
```

In this example, the sys module is effortlessly incorporated to retrieve command line arguments using the sys.argv attribute.

### Example 2: Displaying Python version information

```
# The sys module is used to display Python version information
import sys

# Python version information is retrieved
python_version_info = sys.version_info

print("Python version information:", python_version_info)
```

Here, the sys module is applied to retrieve Python version information using the sys.version\_info attribute.

### Example 3: Exiting a script with an error message

```
# The sys module is utilized to exit a script with an error message
import sys

# An error message is defined
error_message = "An error occurred. Exiting the script."
```

```
# The script is exited with an error message
sys.exit(error_message)
```

In this instance, the sys module is seamlessly integrated to exit a script with a specified error message using the sys. exit() function.

#### **Example 4: Modifying maximum recursion depth**

```
# The sys module is applied to modify the maximum recursion depth
import sys

# The maximum recursion depth is increased
sys.setrecursionlimit(3000)

print("Maximum recursion depth set to 3000.")
```

Here, the sys module is used to modify the maximum recursion depth using the sys. setrecursionlimit() function.

These examples demonstrate the sys module in Python, emphasizing its role in interacting with the Python interpreter and providing system-specific information and functionalities.

Aspect	Comparison
+-----+	
Purpose	- sys: Provides access to interpreter-related variables     and functions.     - os: Provides operating system-dependent functionality.
+-----+	
Common Methods	- sys: Interpreter state, process control, system-specific     parameters.     - os: File and directory operations, process management,     environment variables.
+-----+	
Notable Attributes	- sys: Interpreter version, platform, module management.     - os: File paths, separators, line terminators.
+-----+	
Additional Features	- sys: Interpreter-related features like version info,     memory management.     - os: File system operations, process management,     environment variables.
+-----+	



## Third-party libraries and packages

- **Python ecosystem:** The Python programming ecosystem encompasses a diverse range of third-party libraries that enhance functionality and simplify various tasks.
- **Key libraries:** Two widely adopted libraries within the Python ecosystem are 'requests' and 'pandas', each serving distinct purposes.
- **Requests library:** 'requests' is a prominent example, offering a comprehensive suite for managing HTTP requests, making it invaluable for web scraping, API interactions, and web development tasks.
- **Integration of external packages:** The process of seamlessly integrating external packages into Python projects is facilitated by resources such as the Python Package Index (PyPI) and the pip package manager.
- **Python Package Index (PyPI):** PyPI serves as a repository for Python packages, providing a vast collection of open-source libraries and modules for developers to utilize.
- **pip package manager:** pip is a command-line tool used for installing, managing, and uninstalling Python packages. It simplifies the process of acquiring and incorporating external packages into the Python environment.
- **Using pip for package installation:** Utilizing pip for package installation is a fundamental aspect of the development process. Developers can easily install packages by running commands such as `pip install package_name`, thereby streamlining the integration of external dependencies.

*Figure 5-8. Third-party libraries and packages*

**Dynamic ecosystem:** The dynamic ecosystem of third-party contributions not only expands Python's capabilities but also underscores its collaborative and community-driven nature. Developers can leverage a wide array of libraries to address diverse use cases and enhance productivity.

### Using pip for package installation

In Python, the installation of packages is facilitated through the use of the pip package manager. Packages, whether they be standard library modules or third-party libraries, are effortlessly installed using the `pip install` command, providing a seamless mechanism for developers to augment the functionality of their Python projects. The versioning and management of packages are inherently handled by pip, ensuring a streamlined and efficient process for package installation. As packages are installed, their dependencies are resolved automatically, creating a cohesive environment for Python projects.

To illustrate the usage of a pip for package installation in Python, let's consider a practical example of installing popular library requests.

### Requests library

The 'requests' library is employed for making HTTP requests in Python. It simplifies the process of interacting with web services and APIs, providing an intuitive interface for handling HTTP methods such as GET, POST, and more. For example, the following code snippet demonstrates the installation and usage of the 'requests' library:

Features of the 'requests' Library:

- **HTTP requests:** The 'requests' library provides a simple and elegant API for making HTTP requests, enabling developers to interact with web services and APIs effortlessly.
- **GET and POST requests:** It supports various HTTP methods, including GET, POST, PUT, DELETE, etc., allowing developers to perform operations such as fetching data and submitting forms.
- **Response handling:** 'requests' simplifies handling HTTP responses by providing convenient methods to access response headers, content, status codes, and other relevant information.
- **Session management:** The library supports session management, enabling developers to persist parameters across multiple requests, such as cookies and authentication credentials.
- **Custom headers and parameters:** Developers can easily add custom headers, query parameters, and request payloads, making it flexible for different use cases.
- **File uploads:** 'requests' supports file uploads with multipart encoding, simplifying the process of uploading files to web servers.
- **Timeouts and retries:** It offers features for configuring timeouts and retries, ensuring robustness and reliability in network communications.
- **SSL verification:** The library supports SSL certificate verification, providing secure communication over HTTPS.

Best Practices for Using the 'requests' Library:

- **Error handling:** Implement robust error handling to gracefully handle exceptions that may arise during HTTP requests, such as connection errors, timeouts, and invalid responses.
- **Use context managers:** Utilize context managers (i.e., with statements) when making requests to ensure proper resource management and automatic session cleanup.
- **Avoid blocking calls:** Consider using asynchronous libraries like 'aiohttp' for handling concurrent requests in high-performance applications to avoid blocking the event loop.
- **Security considerations:** Always validate and sanitize user input before including it in requests to prevent security vulnerabilities such as SQL injection and cross-site scripting (XSS) attacks.
- **Optimize performance:** Optimize performance by leveraging features such as connection pooling and persistent sessions for improved efficiency when making multiple requests to the same server.
- **Read documentation:** Familiarize yourself with the official documentation of the 'requests' library to explore its full range of features and best practices for usage.
- **Contribute and collaborate:** Contribute to the 'requests' library and collaborate with the community to improve its functionality, report bugs, and share best practices with fellow developers.
- **Supported versions:** The 'requests' library is actively maintained and supports Python 2.7 and Python 3.x versions. It is recommended to use the latest stable release to benefit from the latest features, improvements, and security patches. Regularly updating the library ensures compatibility with the latest Python versions and maintains a secure and efficient codebase.

```
# Using pip for Package Installation
# Install the 'requests' library using pip
# The '-q' flag is used for quiet mode, reducing output verbosity
# The '-U' flag is used to upgrade the package if it's already installed
# The 'requests' library is a common choice for making HTTP requests in Python
# Note: Ensure that pip is installed and available in the system PATH
!pip install -q -U requests
# Once installed, import the 'requests' module and use it in Python code
import requests
# Example: Make a simple HTTP GET request to a website
```

```

response = requests.get("https://www.example.com")
# Print the status code and content of the response
print(f"Status Code: {response.status_code}")
print("Response Content:")
print(response.text)

```

### Pandas library

The 'pandas' library is renowned for its prowess in data manipulation and analysis. It introduces powerful data structures, such as DataFrames, that facilitate efficient handling of structured data. Consider an example below, showcasing the installation of 'pandas' and a basic operation on a data frame:

```

# Installation of the 'pandas' library using pip
!pip install -q -U pandas

# Importing the library and performing a basic operation on a DataFrame
import pandas as pd

data = {'Column1': [1, 2, 3], 'Column2': ['A', 'B', 'C']}
df = pd.DataFrame(data)

print("Original DataFrame:")
print(df)

```

In this example, the code utilizes the pip install command to install the requests library. The -q flag is used for quiet mode, reducing output verbosity during installation. The -U flag ensures that the library is upgraded if it's already installed. Following the installation, the requests library is imported, and a simple HTTP GET request is made to the "https://www.example.com" website. The status code and content of the response are then printed.

### Conclusion

This exploration of Regular Expressions and Python modules concludes with a newfound understanding of essential programming tools. Regular Expressions have showcased their utility in text processing and pattern matching, revealing a robust mechanism for handling strings with precision. Similarly, the exploration of module creation, standard library modules, and third-party libraries has emphasized the significance of modular, reusable code in Python development. As this journey concludes, the knowledge gained provides a solid foundation for constructing elegant and efficient Python code, ensuring versatility and extensibility in diverse programming endeavors.



IBM ICE (Innovation Centre for Education)

## Self evaluation: Exercise 21

- **Exercise 21:** Email Validation with Regular Expressions.
- **Estimated time:** 00:15 minutes.
- **Aim:** Write a Python program that validates email addresses using regular expressions. Prompt the user for an email address and validate it.
- **Learning objective:**
  - The learner will understand the use of regular expressions for validating complex patterns, specifically focusing on email address validation.
- **Learning outcome:**
  - The learner will be able to implement email validation in Python using regular expressions, enhancing their understanding of pattern matching.

---

Figure 5-9. Self evaluation: Exercise 21

Self evaluation exercise 21 is as stated above:



IBM ICE (Innovation Centre for Education)

## Self evaluation: Exercise 22

- **Exercise 22:** Phone Number Extraction.
- **Estimated time:** 00:15 minutes.
- **Aim:** Create a Python script that extracts phone numbers from a given text using regular expressions.
- **Learning objective:**
  - The learner will gain proficiency in using regular expressions to extract specific patterns, focusing on phone number extraction in this exercise.
- **Learning outcome:**
  - The learner will be able to implement a Python script that utilizes regular expressions to extract phone numbers from a given text, enhancing their skills in pattern matching.

---

Figure 5-10. Self evaluation: Exercise 22

Self evaluation exercise 22 is as stated above:



IBM ICE (Innovation Centre for Education)

## Self evaluation: Exercise 23

- **Exercise 23:** Exploring Built-in Modules.
- **Estimated time:** 00:15 minutes.
- **Aim:** Explore Python's built-in modules like math and datetime. Use them to perform mathematical operations and work with date and time.
- **Learning objective:**
  - The learner will gain hands-on experience in utilizing Python's built-in modules, specifically math and datetime, to perform mathematical operations and work with date and time functionalities.
- **Learning outcome:**
  - The learner will be proficient in leveraging Python's math module for mathematical computations and datetime module for managing date and time in their scripts.

---

Figure 5-11. Self evaluation: Exercise 23

Self evaluation exercise 23 is as stated above:



IBM ICE (Innovation Centre for Education)

## Self evaluation: Exercise 24

- **Exercise 24:** Third-Party Library Usage.
- **Estimated time:** 00:15 minutes.
- **Aim:** Install and use a third-party library (e.g., requests) to fetch data from a web API. Retrieve data and display it in your Python script.
- **Learning objective:**
  - The primary objective of this exercise is to familiarize learners with the process of installing and utilizing third-party libraries in Python. Specifically, the focus is on incorporating the requests library to interact with web APIs and retrieve data.
- **Learning outcome:**
  - Upon completing this exercise, learners will acquire practical skills in integrating external libraries into Python scripts. This experience enhances their ability to retrieve and manipulate data from web APIs, a valuable skill in various programming scenarios.

---

Figure 5-12. Self evaluation: Exercise 24

Self evaluation exercise 24 is as stated above:



## Checkpoint (1 of 2)

### Multiple choice questions:

1. Which of the following regular expressions matches a phone number in the format (123) 456-7890?
  - a) \d{3}-\d{3}-\d{4}
  - b) (\d{3}) \d{3}-\d{4}
  - c) \d{3} \d{3}-\d{4}
  - d) [0-9]{3}-[0-9]{3}-[0-9]{4}
2. What does the '^' symbol represent in regular expressions?
  - a) Matches any single character
  - b) Matches the beginning of a string
  - c) Matches the end of a string
  - d) Matches any word character
3. Consider two modules "module1.py" and "module2.py" with the following contents:  
When module2 is executed, choose the correct output from the options given below:
  - a) 0
  - b) No output
  - c) Error
  - d) 120

---

Figure 5-13. Checkpoint (1 of 2)

Write your answers here:

- 1.
- 2.
- 3.



IBM ICE (Innovation Centre for Education)

## Checkpoint (2 of 2)

### Fill in the blanks:

1. The question mark (?) in a regular expression denotes \_\_\_\_\_ matching for the preceding character or group.
2. The asterisk (\*) in a regular expression indicates \_\_\_\_\_ matching for the preceding character or group.
3. The plus sign (+) in a regular expression signifies \_\_\_\_\_ matching for the preceding character or group.
4. To check if a pattern matches the \_\_\_\_\_ of a string, the re.match() function is commonly used.

### True/False:

1. Quantifiers in regular expressions, such as \* and +, specify the number of occurrences of a pattern. **True/False**
2. The sys module provides access to some variables used or maintained by the Python interpreter. **True/False**
3. Third-party libraries cannot be installed using tools like pip in Python. **True/False**

---

Figure 5-14. Checkpoint (2 of 2)

Write your answers here:

Fill in the blanks:

- 1.
- 2.
- 3.
- 4.

True or false:

- 1.
- 2.
- 3.

# Question bank (1 of 2)



IBM ICE (Innovation Centre for Education)

## Two mark questions:

- If area of one wall of a cubical wooden box is 16 units, write a Python program to display the volume of the box. Note: Area of a cube with side 'a' is ' $a^{**}2$ '. Volume of the cube can be computed as ' $a^{**}3$ '.  
Hint: Make use of 'sqrt' and 'pow' functions from math module.
- Explain the purpose of the character class ([...]) in regular expressions. Provide an example demonstrating the use of a character class.
- Describe the role of the caret (^) and dollar sign (\$) characters in regular expressions. Provide an example illustrating their usage.
- What are the functions and methods used from the re module ?

## Four mark questions:

- Project 'Demo' has two packages 'dictionarydemo' and 'listdemo'. Both the packages have a module with the same name 'modify.py'. Both the modules have a function with same name 'add\_element()'. Package structure is as below:  
listdemo -> modify.py -> add\_element()  
dictionarydemo -> modify.py -> add\_element()

Assume, we want to use 'add\_element()' function from both the modules in another module 'test\_module.py' in package 'test'.  
Write the import statements that will work here?

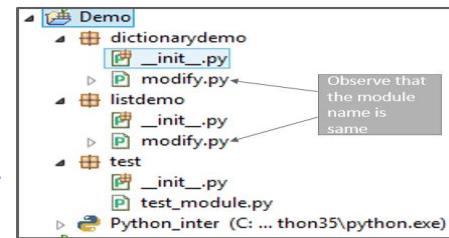


Figure 5-15. Question bank (1 of 2)



IBM ICE (Innovation Centre for Education)

## Question bank (2 of 2)

2. Explain Pattern matching and text processing.
3. Explain Modules and libraries.
4. Write a Python program to randomly print any of the below numbers:100,200,300,400,500,600,700,800,900,1000. Execute the program 10 times and verify if the number generated in every output is one out of the numbers given in the list above. Also, write a Python program to print a random odd numbers between 10 and 50.

### Eight mark questions:

1. Create a module "number\_checker.py" which has following 2 functions:

`is_prime(num)` : this function returns true if the input number is prime

`is_even(num)`: this function returns true if the input number is even

Create another Python module "test\_module.py". Invoke the functions "`is_prime(num)`" and "`is_even(num)`" in "test\_module.py". Observe the results.

Hint: Import "number\_checker.py" module in "test\_module.py" before using it's functions.

2. Consider a scenario where a web application collects user email addresses during the registration process. Design a Python script using the re module to perform the following tasks:

Implement a function named `validate_email` that takes an email address as input and returns a boolean value indicating whether the email is valid or not. Use regular expressions to validate the email against standard formatting rules.

Create a sample list of email addresses, including both valid and invalid entries.

Figure 5-16. Question bank (2 of 2)

## Unit summary



IBM ICE (Innovation Centre for Education)

- Having completed this unit, you should be able to:
  - Learn regular expressions and use them for text processing and pattern matching
  - Gain proficiency in working with modules to modularize Python code and enhance reusability

---

Figure 5-17. Unit summary

Unit summary is as stated above.

---

# Unit 6. Graphical User Interfaces (GUI) and Web programming

## Overview

This Unit will provide an overview of Develop graphical user interfaces (GUI) using various widgets and events. Understand the basics of Common Gateway Interface (CGI) and create interactive web applications.

## How you will check your progress

- Checkpoint

## References

IBM Knowledge Center

# Unit objectives



IBM ICE (Innovation Centre for Education)

- After completing this unit, you should be able to:
  - Develop graphical user interfaces (GUI) using various widgets and events
  - Understand the basics of Common Gateway Interface (CGI) and create interactive web applications
- Learning outcomes:
  - Utilize regular expressions for efficient text processing and pattern matching
  - Modularize Python code using modules and libraries to enhance code maintainability and reusability

---

Figure 6-1. Unit objectives

Unit objectives and outcomes are as stated above.

# Introduction



IBM ICE (Innovation Centre for Education)

- GUI development in python explores fundamental aspects of graphical user interface (GUI) programming, comparing GUI and command-line interfaces.
- Practical examples, like the calculator GUI and scholarly marks, illustrate python's GUI frameworks.
- Web programming with CGI introduces python's role in web development, emphasizing CGI's role in dynamic content generation.

---

*Figure 6-2. Introduction*

The chapter covers handling HTTP requests and generating responses, highlighting the communication between a web server and python scripts. The strategic link between GUI development and web programming with CGI equips learners with a comprehensive understanding of python's capabilities. This connection bridges GUI development skills with web programming, preparing learners to create both intuitive desktop interfaces and dynamic web applications.

# GUI development



IBM ICE (Innovation Centre for Education)

- Definition of Graphical User Interface (GUI).
- Utilization of libraries and frameworks in GUI Development.
- Nature of GUI programming.
- Contrast with Command-Line Interfaces (CLI).
- Role of frameworks in GUI programming.
- Enhancement of user experience.
- Application of GUI programming.
- Significance of GUI programming.

---

Figure 6-3. GUI development

- **Definition of Graphical User Interface (GUI):**
  - GUI is a user interface type allowing interaction with electronic devices or software through graphical elements like icons, buttons, and windows.
- **Utilization of libraries and frameworks in GUI Development:**
  - Various libraries and frameworks are employed in GUI development using Python.
  - Selection of GUI frameworks (e.g., Tkinter, PyQt, Kivy) is based on project requirements and preferences.
- **Nature of GUI programming:**
  - GUI programming involves creating graphical user interfaces (GUIs) for applications.
  - It entails designing graphical elements such as buttons, menus, and windows to facilitate intuitive user interaction.
- **Contrast with Command-Line Interfaces (CLI):**
  - GUIs utilize visual elements for user interaction, unlike CLI which relies on text-based commands.
- **Role of frameworks in GUI programming:**
  - Frameworks or libraries provide tools and components for creating graphical interfaces.
  - They aid in designing and implementing visual aspects, enabling seamless integration of user input and feedback.

- **Enhancement of user experience:**

- GUIs enhance user experience by offering a user-friendly and visually appealing interaction model.
- Users can perform actions and navigate through applications effortlessly using buttons, checkboxes, etc.

- **Application of GUI programming:**

- GUI programming finds application in various software domains, including desktop applications and web-based interfaces.
- It is considered an essential skill for developers across different domains.

- **Significance of GUI programming:**

- GUI programming plays a crucial role in shaping user interaction with software, emphasizing a visually intuitive approach to application design.



IBM ICE (Innovation Centre for Education)

## GUI vs. Command line interfaces

- Graphical User Interfaces (GUIs) and Command Line Interfaces (CLIs) are two distinct approaches to user interaction with software.
- In GUIs, visual elements such as icons, buttons, and windows are utilized to facilitate user input and navigation.
- On the other hand, CLIs rely on text-based commands entered into a console or terminal.

GUI	CLI
A GUI has 4 main components Windows, Icons, Menus and a Pointer (WIMP)	A command line interface is used to control a computer system solely through the use of text commands, entered by the user. The computer only responds in the form of text based messages and information. They are used by technicians, programmers and power-users.
<b>Advantages:</b>	<b>Advantages:</b>
<ul style="list-style-type: none"> <li>➢ Intuitive – using a WIMP interface is a natural way of working – you shouldn't need help or training</li> <li>➢ Multitasking – Windows enable users to open many programs, to move between them easily and transfer data between applications</li> <li>➢ Icons – Make it obvious what a program, file or folder is with a graphical representation</li> <li>➢ Pointers – Pointing is a natural way of selecting things</li> </ul>	<ul style="list-style-type: none"> <li>➢ Extremely fast – CLI's take up little system resources and CPU time, can run on old or limited hardware</li> <li>➢ Can use "scripts" – scripts are files which contain many commands and can be used to perform repetitive tasks very quickly or to automate tasks such as daily backups</li> </ul>
<b>Disadvantages:</b>	<b>Disadvantages:</b>
<ul style="list-style-type: none"> <li>➢ Resource intensive – requires a lot of processing power/time and memory</li> <li>➢ Can become confusing – users may find interfaces cluttered with icons or open programs difficult to navigate</li> </ul>	<ul style="list-style-type: none"> <li>➢ Can be very confusing to new or untrained users</li> <li>➢ Sometimes do not provide detailed error messages or feedback to help users</li> </ul>

Figure 1: GUI vs CLI

Figure 6-4. GUI vs. Command line interfaces

In GUIs, user actions are performed by clicking, dragging, or interacting with graphical elements, offering an intuitive and user-friendly experience. In contrast, CLIs require users to type specific commands, often with parameters, to execute tasks. A comparative analysis between GUI and CLI is presented in figure 1.



IBM ICE (Innovation Centre for Education)

## GUI frameworks in python

- In python, GUI frameworks play a pivotal role in facilitating the development of graphical user interfaces. Tkinter, PyQt, and Kivy, among others, are widely utilized.
- GUI frameworks in python enable developers to design intuitive interfaces, fostering efficient and user-friendly interactions within applications.
- Below is the list of some popular python GUI frameworks:
  - Tkinter.
  - PyQt.
  - Kivy.
  - wxPython.
  - Dear PyGui.
  - Eel.

*Figure 6-5. GUI frameworks in python*

Below is the list of some popular python GUI frameworks, while Table 1 shows a comparative view of the pros and cons of each of them:

- **Tkinter:** Tkinter, a widely adopted python GUI framework, is known for its simplicity and cross-platform compatibility on Windows, MacOS, and Linux. Its popularity is attributed to being bundled with python as a standard library, providing easy accessibility for all developers. Tkinter is often favored for quick prototyping, internal tools, and educational purposes. However, for more extensive or complex applications, developers may opt for frameworks like PyQt, wxPython, or Kivy, offering additional features, better aesthetics, or platform-specific advantages.
- **PyQt:** PyQt, a python binding for the Qt framework, is feature-rich and suitable for projects of varying sizes. It provides customizable widgets and offers a visual design tool, Qt designer. PyQt's cross-platform nature and good documentation make it a robust choice. However, its GPL license for PyQt5 may require consideration for commercial projects.
- **Kivy:** Kivy, designed for cross-platform development, is known for its touch-friendly interfaces. It supports various platforms, including Android and iOS. While open-source and python-based, Kivy may lack a native look and feel on all platforms. Its strengths lie in its versatility, making it a choice for applications requiring touch interaction.
- **wxPython:** wxPython, providing a native look and feel on multiple platforms, is a versatile and customizable GUI framework. Although not as widely used as Tkinter or PyQt, it offers good documentation and ease of customization. Installation may be required, and wxPython may not have the same level of community support as the more established frameworks.

- Dear PyGUI:** PyGUI, encompassing frameworks like Tkinter, PyQt, and Kivy, stands out as an excellent choice for Graphical User Interface (GUI) development in python. Its appeal lies in its user-friendly development environment, cross-platform compatibility, and a diverse set of pre-built widgets that streamline the creation of interactive interfaces. With robust community support and integration capabilities with other Python libraries, PyGUI allows developers to seamlessly blend GUI applications with various functionalities, from data processing to machine learning. Its open-source nature fosters flexibility, enabling customization, while the rapid prototyping features and scalability make it suitable for projects of varying sizes. Industry-wide adoption further solidifies PyGUI's position as a reliable solution for building desktop applications and diverse software tools.
- Gtk (PyGObject):** Gtk, with PyGObject as its Python binding, is a mature toolkit that provides a native look on Linux systems. While GTK+ has been widely used, its usage in Python may require additional installation on non-Linux systems. It may not have the same level of cross-platform integration as other frameworks.
- Eel:** Eel allows the use of HTML, CSS, and Javascript for GUI, providing a lightweight and straightforward approach. While suitable for simple applications, it may lack the features and extensibility of more traditional frameworks. It offers a unique approach by combining web technologies with python for GUI development.

Framework	Pros	Cons
Tkinter	- Built-in with Python.	- Limited modern UI features.
	- Simple and easy to use.	- GUIs may look outdated.
	- Cross-platform compatibility.	
PyQt	- Feature-rich.	- GPL license for PyQt5.
	- Good documentation.	- Requires installation.
	- Visual design with Qt Designer.	
Kivy	- Cross-platform (including mobile).	- May lack a native look on some platforms.
	- Open source and Python-based.	
wxPython	- Native look and feel.	- Requires installation.
	- Versatile and customizable.	- May have less community support.
Dear PyGui	- Fast and simple.	- Relatively new with less community adoption.
	- Immediate mode GUI.	
Gtk (PyGObject)	- Mature and widely used.	- Requires installation on non-Linux systems.
	- Native on Linux systems.	- May lack cross-platform integration.
Eel	- Uses HTML, CSS, and JavaScript for GUI.	- Limited features compared to traditional frameworks.
	- Lightweight.	- Less extensibility.

Table 1: Comparative analysis of popular python frameworks



IBM ICE (Innovation Centre for Education)

## Widgets and event handling

- A widget is an element of the Graphical User Interface (GUI) that displays or illustrates information and provides a means for the user to interact with the operating system.
- In Tkinter, widgets are represented as objects, they are instances of classes that depict buttons, frames, and so on. these widgets are configured to be responsive to specific user interactions, such as button clicks or keypresses.
- Events are bound to widgets, and corresponding functions or methods are defined to handle these events.
- During the execution of the application, an event loop continuously monitors user interactions, and when an event occurs, the associated handling function is invoked. event parameters are often received by these functions, providing information about the nature of the event.

*Figure 6-6. Widgets and event handling*

Python has several GUI frameworks, but Tkinter is the only framework that is built into the python standard library. Visual elements are rendered using native operating system elements, resulting in applications built with Tkinter appearing integrated with the platform where they are executed.

### **Tkinter module**

#### **Purpose:**

Tkinter is the standard GUI (Graphical User Interface) toolkit included with Python. It provides a fast and easy way to create GUI applications.

#### **Features:**

- Simple and intuitive interface for building GUI applications. Provides various widgets such as buttons, labels, entry fields, and more.
- Supports event-driven programming, allowing actions to be triggered by user interactions.
- Offers geometry management to arrange widgets within the application.
- Cross-platform compatibility, as Tkinter is available on most operating systems.

#### **Usage:**

- Tkinter is widely used for creating desktop applications, utilities, and tools with graphical interfaces.
- It is suitable for both simple and complex applications.

#### **Getting started:**

To use Tkinter, import it into your Python script with: `import tkinter as tk`. Then, create a root window and add widgets to it. Finally, start the event loop with `tk.mainloop()`.

**Documentation:**

Tkinter's official documentation provides comprehensive information about its usage, widgets, methods, and examples. It can be found at: [Tkinter Documentation](#).



IBM ICE (Innovation Centre for Education)

## Creating widgets (1 of 2)

- All Tkinter widgets have access to specific geometry management methods, which have the purpose of organizing widgets throughout the parent widget area.
- Tkinter exposes the following geometry manager classes:
  - [Pack Geometry Manager](#).
  - [Grid Geometry Manager](#).
  - [Place Geometry Manager](#).

*Figure 6-7. Creating widgets (1 of 2)*

**Pack geometry manager:**

- The pack() method is used to organize widgets in blocks before placing them in the parent widget.
- It arranges widgets sequentially either horizontally or vertically.

**Example:**

```
widget.pack()
```

**Grid geometry manager:**

- The grid() method organizes widgets in a table-like structure within the parent widget.
- Widgets are placed in rows and columns, allowing for more precise control over their positioning.

**Example:**

```
widget.grid(row=0, column=0)
```

**Place geometry manager:**

- The place() method allows precise placement of widgets at specific positions within the parent widget.
- Widgets are placed using x and y coordinates or by specifying anchor points.

**Example:**

```
widget.place(x=100, y=100)
```

These geometry manager classes provide different ways to manage the layout of widgets within a Tkinter application, offering flexibility and control over their positioning and arrangement on the screen. Developers can choose the most suitable geometry manager based on the desired layout requirements for their GUI application.

### Example

```
from tkinter import *
window = tk.Tk()
label = tk.Label(text="Python rocks!")
label.pack()
window.mainloop()
```

Figure 2 comprehensively presents an illustrative list of various widgets that Tkinter offers. These widgets serve as building blocks for constructing interactive graphical user interfaces, each designed to cater to specific functionalities and user interactions. The diversity of widgets available in Tkinter allows developers to tailor interfaces for an extensive range of applications, ensuring a versatile and dynamic user experience.

Widget	Purpose
<code>tk.Button , ttk.Button</code>	Execute a specific task; a "do this now" command.
<code>tk.Menu</code>	Implements toplevel, pulldown, and popup menus.
<code>ttk.Menubutton</code>	Displays popup or pulldown menu items when activated.
<code>tk.OptionMenu</code>	Creates a popup menu, and a button to display it.
<code>tk.Entry , ttk.Entry</code>	Enter one line of text.
<code>tk.Text</code>	Display and edit formatted text, possibly with multiple lines.
<code>tk.Checkbutton , ttk.Checkbutton</code>	Set on-off, True-False selections.
<code>tk.Radiobutton , ttk.Radiobutton</code>	Allow one-of-many selections.
<code>tk.Listbox</code>	Choose one or more alternatives from a list.
<code>ttk.Combobox</code>	Combines a text field with a pop-down list of values.
<code>tk.Scale , ttk.Scale</code>	Select a numerical value by moving a "slider" along a scale.

Name of Widget	Description
<a href="#">Button</a>	If you want to add a button in your application then the <b>Button widget</b> will be used. A button is clickable and the user can click the button widget to perform any action.
<a href="#">Canvas</a>	To draw a complex layout and pictures (like graphics, text, etc.) the <b>Canvas Widget</b> can be used.
<a href="#">Checkbutton</a>	If you want to display a number of options as checkboxes then the <b>Checkbutton widget</b> can be used. It allows you to select multiple options at a time.
<a href="#">Entry</a>	To display a <b>single-line text field</b> that accepts values from the user <b>Entry widget</b> can be used.
<a href="#">Frame</a>	In order to group and organize other widgets, the Frame widget can be used. Basically, it acts as a <b>container that holds other widgets</b> .
<a href="#">Label</a>	To Provide a <b>single-line caption</b> to another widget Label widget can be used. It can <b>contain images</b> too.
<a href="#">Listbox</a>	To provide a user with a list of options the <b>Listbox widget</b> can be used.
<a href="#">Menu</a>	To provide commands to the user <b>Menu widget</b> can be used. Basically, these <b>commands are inside the Menu button</b> . This widget mainly creates <b>all kinds of Menus</b> required in the application.
<a href="#">Menubutton</a>	The <b>Menubutton widget</b> is used to display the menu items to the user.
<a href="#">Message</a>	The <b>Message widget</b> mainly displays a message box to the user. Basically, it is a <b>multi-line text which is non-editable</b> .
<a href="#">Radiobutton</a>	If you want the number of options to be displayed as radio buttons then the <b>Radiobutton widget</b> can be used. You can select one at a time.

<a href="#">Scale</a>	<b>Scale widget</b> is mainly a <b>graphical slider</b> that allows you to select values from the scale.
<a href="#">Scrollbar</a>	To scroll the window up and down the <b>Scrollbar widget</b> in Python can be used.
<a href="#">Text</a>	The <b>Text widget</b> mainly provides a <b>multi-line text field</b> to the user where users enter or edit the text and it is different from Entry.
<a href="#">Toplevel</a>	The <b>Toplevel widget</b> is mainly used to provide us with a separate window container
<a href="#">SpinBox</a>	The <b>SpinBox</b> acts as an entry to the "Entry widget" in which value can be input just by <b>selecting a fixed value of numbers</b> .
<a href="#">PanedWindow</a>	The <b>PanedWindow</b> is also a <b>container widget</b> that is mainly used to <b>handle different panes</b> . Panes arranged inside it can either be <b>horizontal or vertical</b>
<a href="#">LabelFrame</a>	The <b>LabelFrame widget</b> is also a container widget used to mainly handle complex widgets.
<a href="#">MessageBox</a>	The <b>MessageBox widget</b> is mainly used to display messages in desktop applications.

Table 2: Tkinter widget and purpose

[<https://www.studytonight.com/tkinter/python-tkinter-widgets>]**Button:**

The Button widget in Tkinter is used to create a clickable button in a graphical user interface (GUI) application. It allows users to trigger actions or events when clicked. Here are some key points about the Button widget:

- **Purpose:** The Button widget is used to create interactive buttons that users can click on to perform specific actions or trigger events in a Tkinter application.

- **Appearance:** Buttons typically have a text label or an image displayed on them, indicating the action they perform when clicked.
- **Usage:** Buttons are commonly used for various purposes such as submitting forms, executing commands, navigating between screens, or triggering functions.
- **Configuration options:** The Button widget in Tkinter provides various configuration options to customize its appearance and behavior, including text, font, background color, foreground color, width, height, and command (the function to be called when the button is clicked).
- **Event handling:** Buttons can be associated with event handlers or callback functions that are executed when the button is clicked. This allows developers to define the desired behavior or action to be performed in response to the button click event.

A button is instantiated by utilizing the Button class in the tkinter. The button is then configured with text and a callback function for handling events.

```
import tkinter as tk

def button_click():
    label.config(text="Button Clicked!")

root = tk.Tk()

button = tk.Button(root, text="Click Me", command=button_click)
button.pack()

label = tk.Label(root, text="Hello, Tkinter!")
label.pack()

root.mainloop()
```

A button is created within the tkinter framework for Python. The process involves instantiating a button object using the Button class, and the button is configured with specific properties, such as the displayed text and a callback function (button\_click) that executes when the button is clicked. The button is then packed or positioned within the graphical user interface. Additionally, a label widget is created and packed to display an initial text message.

The main event loop (root. mainloop()) is initiated, allowing the graphical user interface to be displayed and enabling user interaction. When the button is clicked, the callback function updates the label text, demonstrating a simple interaction between widgets.

## Label

The Label widget in Tkinter is used to display text or images on a graphical user interface (GUI) without any interaction from the user. It is typically used to provide information or instructions to the user or to display static content such as headings, titles, or descriptions. Here are some key points about the Label widget:

- **Purpose:** The Label widget is primarily used to display text or images on a Tkinter GUI application.
- **Text display:** Labels can display static text strings, variable text generated at runtime, or even formatted text using HTML tags.
- **Image display:** Labels can also display images such as PNG, GIF, or JPEG files.
- **Appearance:** Labels can be customized in terms of font, size, color, alignment, and background color to match the overall design of the GUI.
- **Usage:** Labels are commonly used to provide informative text, headings, titles, descriptions, or labels for other widgets in the GUI.

Example:

```
from tkinter import Tk, Label

root = Tk()

label = Label(root, text="Hello, Tkinter!")
```

```

label.pack()
root.mainloop()

```

In this example, a Label widget displaying the text "Hello, Tkinter!" is created and packed into the root window. The label is displayed on the GUI when the application is run.

**Dynamic Updating:** While Labels are typically static, their text or image content can be updated dynamically at runtime to reflect changing data or states within the application.

## Entry

The Entry widget in Tkinter is used to create a single-line text entry field where users can input text or data. It provides a user interface element for users to type in textual information, such as usernames, passwords, search queries, or any other form of input. Here are some key points about the Entry widget:

- **Purpose:** The Entry widget is primarily used to allow users to input text or data in a Tkinter GUI application.
- **Text input field:** Entry widgets provide a single-line text field where users can type in text using the keyboard.
- **Appearance:** Entry widgets typically appear as rectangular boxes where users can enter text. The appearance can be customized in terms of width, height, font, color, and border.
- **Text editing:** Users can edit the text within the Entry widget using standard keyboard input and editing operations such as typing, deleting, selecting, copying, and pasting.
- **Validation:** Entry widgets support various validation mechanisms to ensure that the entered text meets specific criteria, such as limiting the length of input, restricting input to certain characters, or validating input against a predefined format.
- **Usage:** Entry widgets are commonly used in forms, dialogs, input fields, search boxes, or any other scenario where user input is required.

Example:

```

from tkinter import Tk, Entry
root = Tk()
entry = Entry(root, width=30)
entry.pack()
root.mainloop()

```

In this example, an Entry widget with a width of 30 characters is created and packed into the root window. Users can input text into the entry field when the application is run.

**Accessing Input:** Developers can access the text entered by the user in the Entry widget using methods such as `get()` to retrieve the current text content or `insert()` to programmatically insert text into the entry field.

## Example 2-

The Entry widget allows users to input text. It provides a single-line text entry field.

Example:

```

import tkinter as tk

def show_entry_text():
    label.config(text=entry.get())

root = tk.Tk()
entry = tk.Entry(root)
entry.pack()

button = tk.Button(root, text="Show Entry Text", command=show_entry_text)
button.pack()

```

```
label = tk.Label(root, text="")
label.pack()
root.mainloop()
```

In the provided code snippet, the tkinter module is imported as tk. A function named show\_entry\_text is defined, and within this function, the text of a label is configured based on the content retrieved from an entry widget. A tkinter root window is created, and an entry widget is instantiated within this root window. Subsequently, a button widget is created with the text "Show Entry Text," and the command attribute is set to the show\_entry\_text function. Both the button and an initially empty label are packed within the root window. Finally, the main event loop is initiated to display the graphical user interface.

## Creating widgets (2 of 2)



IBM ICE (Innovation Centre for Education)

- Text: The Text widget in Tkinter is used to create a multi-line text entry and display area where users can input and view text or data.
- It provides a user interface element for working with large bodies of text, such as paragraphs, documents, logs, or code snippets.

---

Figure 6-8. Creating widgets (2 of 2)

Here are some key points about the Text widget:

- **Purpose:** The Text widget is primarily used to allow users to input, edit, and display multi-line text or data in a Tkinter GUI application.
- **Multi-line text field:** Text widgets provide a scrollable area where users can enter or view text spanning multiple lines.
- **Text editing:** Users can edit the text within the Text widget using standard keyboard input and editing operations such as typing, deleting, selecting, copying, cutting, and pasting.
- **Appearance:** Text widgets typically appear as rectangular areas with vertical and horizontal scrollbars, allowing users to navigate through large amounts of text.
- **Text formatting:** Text widgets support basic text formatting options such as font styles (e.g., bold, italic, underline), text alignment, indentation, and bullet points.
- **Syntax highlighting:** Developers can implement syntax highlighting for code snippets or structured text within Text widgets to improve readability and visual clarity.
- **Usage:** Text widgets are commonly used for tasks involving large bodies of text, such as text editors, word processors, code editors, chat applications, log viewers, and document viewers.

**Example:**

```
from tkinter import Tk, Text, Scrollbar
```

```

root = Tk()
# Create a Text widget
text_widget = Text(root, height=10, width=40)
# Create a Scrollbar and attach it to the Text widget
scrollbar = Scrollbar(root, command=text_widget.yview)
text_widget.config(yscrollcommand=scrollbar.set)
# Pack the Text widget and Scrollbar
text_widget.pack(side="left", fill="both", expand=True)
scrollbar.pack(side="right", fill="y")
root.mainloop()

```

In this example, a Text widget with a height of 10 lines and a width of 40 characters is created and packed into the root window. A Scrollbar is also created and attached to the Text widget to allow scrolling through the text when it exceeds the visible area.

The Text widget is used for multiline text input or display.

### **Example 2:**

```

import tkinter as tk
root = tk.Tk()
text = tk.Text(root, height=5, width=30)
text.pack()
text.insert(tk.END, "Enter your text here.")
root.mainloop()

```

In the above code, a Text widget is instantiated within the root window, configured with a height of 5 lines and a width of 30 characters, and subsequently packed within the root window. The insert method is then used to add the text "Enter your text here." at the end of the Text widget. Finally, the main event loop is initiated to display the graphical user interface.

### **Checkbutton**

The Checkbutton widget in Tkinter is used to create a checkable button or checkbox in a graphical user interface (GUI) application. It allows users to toggle between two states: checked (selected) and unchecked (deselected). Here are some key points about the Checkbutton widget:

- **Purpose:** The Checkbutton widget is primarily used to provide users with a binary choice or option in a Tkinter GUI application.
- **Appearance:** Checkbuttons typically appear as small squares or rectangles with a label or text beside them to indicate the option they represent. When checked, a mark (such as a checkmark or an 'X') appears inside the checkbox.
- **State:** Checkbuttons can be in one of two states: checked (selected) or unchecked (deselected). Users can toggle the state of the Checkbutton by clicking on it.
- **Usage:** Checkbuttons are commonly used to represent boolean options or settings that can be turned on or off independently. For example, they can be used for enabling/disabling features, selecting items from a list, or setting preferences.
- **Configuration options:** Checkbuttons provide various configuration options to customize their appearance and behavior, including text, font, background color, foreground color, width, height, and command (the function to be called when the Checkbutton state changes).

### **Example:**

```
from tkinter import Tk, Checkbutton, IntVar
```

```

root = Tk()

# Create a variable to store the Checkbutton state
check_var = IntVar()

# Create a Checkbutton
checkbutton = Checkbutton(root, text="Enable Feature", variable=check_var)

# Pack the Checkbutton
checkbutton.pack()

root.mainloop()

```

In this example, a Checkbutton labeled "Enable Feature" is created and packed into the root window. The state of the Checkbutton is controlled by an IntVar variable named check\_var.

**Accessing State:** Developers can access the state of the Checkbutton (checked or unchecked) by associating it with a Tkinter variable (e.g., IntVar, BooleanVar, or StringVar) and retrieving the value of that variable.

## Example 2

The Checkbutton widget is a checkbox that allows users to select or deselect an option .

Example:

```

import tkinter as tk

def checkbutton_callback():
    label.config(text=f"Checked: {var.get() }")

root = tk.Tk()
var = tk.IntVar()

checkbutton = tk.Checkbutton(root, text="Check me", variable=var,
command=checkbutton_callback)

checkbutton.pack()

label = tk.Label(root, text="")
label.pack()

root.mainloop()

```

In the provided code snippet, the tkinter module is imported as tk, and a function named checkbutton\_callback is defined. This function configures the text of a label based on the current state of a checkbutton. A tkinter root window is created using the Tk() constructor. An IntVar named var is instantiated to store the state of the checkbutton. A checkbutton is created within the root window, configured with the text "Check me," associated with the var variable and linked to the checkbutton\_callback function for event handling. Both the checkbutton and an initially empty label are packed within the root window. Finally, the main event loop is initiated to display the graphical user interface.

## Radiobutton

The Radiobutton widget in Tkinter is used to create a group of radio buttons where users can select one option from a set of mutually exclusive choices. It allows users to choose a single option from a predefined list of options presented as radio buttons. Here are some key points about the Radiobutton widget:

- **Purpose:** The Radiobutton widget is primarily used to present users with a set of options or choices in a Tkinter GUI application, where only one option can be selected at a time.
- **Appearance:** Radiobuttons typically appear as small circular buttons with a label or text beside them to indicate the option they represent. When one Radiobutton is selected, the previously selected Radiobutton in the same group is automatically deselected.
- **Mutual exclusivity:** Radiobuttons within the same group are mutually exclusive, meaning that selecting one Radiobutton automatically deselects all other Radiobuttons in the same group.

- **Usage:** Radiobuttons are commonly used for scenarios where users need to make a single choice from a predefined list of options. For example, they can be used for selecting a gender, choosing a payment method, or specifying preferences.
- **Configuration options:** Radiobuttons provide various configuration options to customize their appearance and behavior, including text, font, background color, foreground color, width, height, and command (the function to be called when the Radiobutton selection changes).

### **Example 1:**

```
from tkinter import Tk, Radiobutton, IntVar
root = Tk()
# Create a variable to store the selected Radiobutton value
radio_var = IntVar()
# Create Radiobuttons
radio_button1 = Radiobutton(root, text="Option 1", variable=radio_var, value=1)
radio_button2 = Radiobutton(root, text="Option 2", variable=radio_var, value=2)
radio_button3 = Radiobutton(root, text="Option 3", variable=radio_var, value=3)
# Pack the Radiobuttons
radio_button1.pack()
radio_button2.pack()
radio_button3.pack()
root.mainloop()
```

In this example, three Radiobuttons labeled "Option 1", "Option 2", and "Option 3" are created and packed into the root window. The selected option is stored in an IntVar variable named `radio_var`.

**Accessing Selection:** Developers can access the value of the selected Radiobutton by associating them with a Tkinter variable (e.g., `IntVar`, `StringVar`) and retrieving the value of that variable.

### **Example 2:**

```
import tkinter as tk
# The tkinter module is imported as tk.
# A function named radiobutton_callback is defined to configure the text of a
label based on the selected option of a group of radiobuttons.
def radiobutton_callback():
    label.config(text=f"Selected: {var.get()}")
# A tkinter root window is created.
root = tk.Tk()
# An IntVar named var is instantiated to store the value of the selected
radiobutton.
var = tk.IntVar()
# Two radiobuttons, radiobutton1 and radiobutton2, are created within the root window.
# They are associated with the same var variable, each having a unique value (1 and 2, respectively), and
linked to the radiobutton_callback function for event handling.
radiobutton1 = tk.Radiobutton(root, text="Option 1", variable=var, value=1, command=radiobutton_callback)
radiobutton1.pack()
radiobutton2 = tk.Radiobutton(root, text="Option 2", variable=var, value=2, command=radiobutton_callback)
```

```
radiobutton2.pack()
# An initially empty label is packed within the root window.
label = tk.Label(root, text="")
label.pack()
# The main event loop is initiated to display the graphical user interface.
root.mainloop()
```

In the provided code snippet, the tkinter module is imported as tk, and a function named radiobutton\_callback is defined. This function configures the text of a label based on the selected option of a group of radiobuttons. A tkinter root window is created using the Tk() constructor. An IntVar named var is instantiated to store the value of the selected radiobutton. Two radiobuttons, radiobutton1 and radiobutton2, are created within the root window. They are associated with the same var variable, each having a unique value (1 and 2, respectively), and linked to the radiobutton\_callback function for event handling. An initially empty label is also packed within the root window. Finally, the main event loop is initiated to display the graphical user interface.



IBM ICE (Innovation Centre for Education)

# Handling user events (clicks, input)

- Handling user events, such as clicks and input, in widgets is a crucial aspect of event handling in python.
- The process involves configuring widgets to respond to specific user actions and implementing event handlers to manage the associated functionalities.
- The following in-depth explanation outlines the approach to handling user events in python:
  - Widget configuration.
  - Event binding.
  - Event handlers.
  - User interaction.
- Example: To-do list
  - A to-do list application using python and Tkinter. In this application, users can add tasks, mark them as completed, and clear the task list. This example demonstrates handling user events in a to-do list application, emphasizing the configuration, binding, and execution of events.

*Figure 6-9. Handling user events (clicks, input)*

## Widget configuration

Widgets, representing graphical elements like buttons or input fields, are configured to detect user interactions.

# Widgets configured to respond to user events

```
button = Button(text="Click Me")
entry = Entry()
```

## Event binding

Events are bound to widgets, establishing a connection between user actions and specific functions. The below code is used to convey the action of linking events to corresponding handlers.

```
# Event binding for a button click
button.bind("<Button-1>", on_button_click)
# Event binding for text entry
entry.bind("<Return>", on_enter_pressed)
```

## Event handlers

Functions known as event handlers are implemented to define the behavior when specific events occur.

```
def on_button_click(event):
    # Action performed when the button is clicked
```

```

        print("Button Clicked!")

def on_enter_pressed(event):
    # Action performed when the 'Return' key is pressed in the entry field
    user_input = entry.get()
    print("Entered Text:", user_input)

```

## User interaction

Users interact with the graphical interface, triggering events on widgets. The process of users interacting with the application:

- **Clicking the button:** The button responds to a click event, invoking the associated event handler.
- **Entering text:** Users enter text in the input field, and the 'Return' key press event is detected.
- **Event execution:** When events are detected, the associated event handlers are executed, performing the specified actions.
- **Button click event:** The function `on_button_click` is executed when the button is clicked.
- **Enter key press event:** The function `on_enter_pressed` is executed when the 'Return' key is pressed.

## Example: To-do List

A to-do list application using python and Tkinter. In this application, users can add tasks, mark them as completed, and clear the task list. This example demonstrates handling user events in a to-do list application, emphasizing the configuration, binding, and execution of events.

```

import tkinter as tk

class TodoListApp:

    def __init__(self, master):
        self.master = master
        self.master.title("Todo List App")
        # Entry widget for new task
        self.new_task_var = tk.StringVar()
        entry = tk.Entry(master, textvariable=self.new_task_var, font=('Arial', 12))
        entry.grid(row=0, column=0, columnspan=2, pady=10, padx=10, sticky='ew')
        # Button to add a new task
        add_button = tk.Button(master, text="Add Task", command=self.add_task,
                               font=('Arial', 12), padx=10, pady=5)
        add_button.grid(row=0, column=2, sticky='ew')
        # Listbox to display tasks
        self.task_listbox = tk.Listbox(master, font=('Arial', 12),
                                      selectmode=tk.SINGLE)
        self.task_listbox.grid(row=1, column=0, columnspan=3, pady=10, padx=10,
                              sticky='nsew')
        # Button to mark task as completed
        complete_button = tk.Button(master, text="Mark Completed",
                                     command=self.mark_completed,
                                     font=('Arial', 12), padx=10, pady=5)
        complete_button.grid(row=2, column=0, sticky='ew')

```

```

# Button to clear completed tasks
    clear_button = tk.Button(master, text="Clear Completed",
command=self.clear_completed,
                                font=('Arial', 12), padx=10, pady=5)
    clear_button.grid(row=2, column=1, columnspan=2, sticky='ew')
# Configure row and column weights
for i in range(3):
    master.grid_rowconfigure(i, weight=1)
    master.grid_columnconfigure(i, weight=1)

def add_task(self):
    new_task = self.new_task_var.get()
    if new_task:
        self.task_listbox.insert(tk.END, f"Task: {new_task}")
        self.new_task_var.set("")

def mark_completed(self):
    selected_index = self.task_listbox.curselection()
    if selected_index:
        self.task_listbox.itemconfig(selected_index, {'bg': 'light green'})

def clear_completed(self):
    completed_indices = [i for i in range(self.task_listbox.size()) if 'light
green' in self.task_listbox.itemcget(i, 'bg')]
    for index in reversed(completed_indices):
        self.task_listbox.delete(index)

# Create the main application window
root = tk.Tk()
app = TodoListApp(root)
# Run the Tkinter event loop
root.mainloop()

```

### **Explanation of the example:**

- Widget configuration: Widgets such as the Entry widget for new tasks, buttons for adding, completing, and clearing tasks, and the Listbox to display tasks are configured in the `__init__` method.
- Event binding: Buttons are bound to their respective event handlers (`add_task`, `mark_completed`, `clear_completed`) using the `command` parameter.
- Event handlers: The `add_task` method adds a new task to the list, `mark_completed` marks the selected task as completed, and `clear_completed` removes completed tasks from the list.
- User interaction: Users interact with the to-do list by entering new tasks, clicking buttons to add, marking as completed, and clear tasks.
- Event execution: When a button is clicked, the associated event handler is executed, updating the task list or performing actions on selected tasks.

# GUI development in python: Example codes



IBM ICE (Innovation Centre for Education)

- In this section, two GUI-based applications named "Calculator GUI" and "ScholarlyMarks"
- Dynamic submission system have been developed using the Tkinter GUI framework in Python.
- Both applications showcase the versatility of Tkinter in creating intuitive graphical interfaces for diverse purposes, ranging from basic calculators to sophisticated academic submission systems.
  - [Example 1: Calculator GUI.](#)
  - [Example 2: ScholarlyMarks: Dynamic submission system.](#)

*Figure 6-10. GUI development in python: Example codes*

## Example 1: Calculator GUI

The calculator GUI leverages Tkinter's widget capabilities to create a graphical interface for performing mathematical operations interactively. Users can input values through the GUI, and the application responds with calculated results. The below code creates a simple calculator GUI with numeric buttons, an entry widget for input/output, and basic arithmetic functionality. The use of the grid method organizes the layout of the buttons on the Tkinter grid. The lambda functions in button commands facilitate the association of actions with button clicks. The screenshot of the output of the below program is presented in Figure 3.

Import the tkinter module and create the main window:

```
from tkinter import *
root = Tk()
root.title('Simple Calculator')

Create an entry widget for displaying input and output:
e = Entry(width=35, fg='white', bg='black', borderwidth=5)
e.grid(row=0, column=0, columnspan=3, padx=10, pady=10)

Define functions for button actions:
def button_click(number):
    current = e.get()
```

```
e.delete(0, END)
e.insert(0, str(current) + str(number))
def button_clear():
    e.delete(0, END)
def button_add():
    first = e.get()
    global f_num
    f_num = int(first)
    e.delete(0, END)
def button_equal():
    second = e.get()
    e.delete(0, END)
    e.insert(0, f_num + int(second))

Create numeric buttons (1 to 9 and 0) with lambda functions:
button_1      = Button(root,      text='1',      padx=40,      pady=20,      command=lambda:
button_click(1))

# ... Repeat for buttons 2 to 9 and 0

Create operator buttons (addition, equal, clear) with corresponding commands:
button_add = Button(root, text='+', padx=39, pady=20, command=button_add)
button_equal = Button(root, text='=', padx=90, pady=20, command=button_equal)
button_clear     = Button(root,      text='clear',      padx=80,      pady=20,
command=button_clear)

Place buttons on the grid using the grid method:
button_1.grid(row=3, column=0)
# ... Repeat for other numeric buttons
button_clear.grid(row=4, column=1, columnspan=2)
button_add.grid(row=5, column=0)
button_equal.grid(row=5, column=1, columnspan=2)
Start the main event loop to display the GUI:
root.mainloop()
```



Figure 2: Output of example 1

### **Example 2. Scholarlymarks: Dynamic submission system**

- ScholarlyMarks: Dynamic submission system, implemented with Tkinter, provides an interactive platform for dynamic submission management in an academic context. This example involves creating a Tkinter GUI application for a marks submission system with scholarship eligibility evaluation. Let's break down the key components and functionality:

#### **First part:**

##### **Button removal:**

- A simple window (root) is created.
- A button ("Delete me") is created with a lambda function assigned to its command.
- The lambda function uses pack\_forget() to remove the button when clicked dynamically.

##### **Main event loop:**

- The main event loop (root.mainloop()) is started, allowing the GUI to be displayed.

#### **Second part:**

##### **Initialization:**

- A new window (master) is created for the marks submission system.
- The window title and geometry are set.

##### **Functions for scholarship and clearing labels:**

- Two functions, scholarship and clearness, are defined.
- Scholarship displays a label indicating eligibility for a 30% scholarship based on gender.
- Clear checks the gender selection and displays appropriate messages.

##### **Labels, Radiobuttons, and Listbox:**

- Labels are created for various information such as Name, Reg.No, and Roll.No.
- A tkinter variable (r) is declared for radiobuttons representing gender.
- Radiobuttons for males and Females are created, each linked to the variable and assigned commands.
- A listbox is used for selecting the semester, displaying options I to V.

##### **Labels for subjects and marks:**

- Labels are created for Serial Numbers, Subjects, and Marks.

- Entry fields are provided for user input for Name, Reg.No, Roll.No, and subject-wise marks.

**ListBox for semester:**

- A listbox is created to select the semester from options I to V.

**Display function and calculation:**

- The display function is defined to calculate and display the total and average marks.
- Entry fields for subject-wise marks are converted to integers, and total and average are calculated.
- Labels for Total and Average are updated dynamically based on the calculated values.

**Submit button:**

- A button ("Submit") triggers the display function when clicked.
- The button has a green background for visual appeal.
- Main Event Loop for the Marks Submission System:
- The main event loop (master.mainloop()) is started to display the complete GUI for the marks submission system.

# Web programming with CGI



IBM ICE (Innovation Centre for Education)

- Web programming with CGI (Common Gateway Interface) in Python is a method that enables the integration of dynamic content and scripts into web servers.
- "Building Interactive Web Applications" encompasses various techniques, frameworks, and tools to create dynamic and engaging web experiences.
- While CGI serves as a server-side mechanism for handling dynamic content, user input, and server interactions, it is a specific subset within the overarching goal of creating interactive web applications.
- The current version is CGI/1.1 and CGI/1.2 is under progress.
- Mention the latest version of CGI. The current version is CGI/1.1 and CGI/1.2 is under progress updated .

*Figure 6-11. Web programming with CGI*

## Introduction to CGI

### Purpose of CGI (Common Gateway Interface) in Python:

- CGI serves as a mechanism enabling the execution of Python scripts on web servers.
- It facilitates the dynamic generation of content in response to HTTP requests.

### Communication process:

- CGI involves communication between web servers and external Python programs or scripts.
- This collaborative process enables the dynamic generation of content.

### Python's suitability for CGI scripting:

- Python's versatility and user-friendly syntax make it a popular choice for CGI scripting.
- Its readability and ease of use contribute to its popularity in this context.

### CGI process flow:

- The CGI process flow includes the identification and execution of Python scripts by the web server.
- This leads to dynamic content generation and the subsequent delivery of output as an HTTP response to the user's browser.

### Key components of cgi:

- HTTP headers, environment variables, and the utilization of the CGI module are essential components of CGI in Python.

- These components contribute to the effective implementation of CGI functionality.

**Deployment and configuration considerations:**

- Deployment and configuration of CGI scripts require careful consideration of security measures.
- Proper script location and server configuration are necessary to ensure a secure and efficient interaction between users and web applications.

**CGI program types:**

- CGI programs can be written in various languages, including Python, PERL, Shell Script, C, or C++.
- These programs are executed by the web server to generate dynamic content in response to client requests.

**Example of web browsing:**

- Suppose a user wants to visit a website hosted on a remote server.
- The user opens a web browser (such as Chrome, Firefox, or Safari) and enters the URL of the website.
- The web browser sends an HTTP request to the web server hosting the website.
- The web server receives the request and processes it, possibly invoking CGI programs to generate dynamic content.
- The server sends back an HTTP response containing the requested web page, along with any other necessary resources (such as images, CSS files, or JavaScript files).

The web browser renders the received HTML content and displays the webpage to the user.



IBM ICE (Innovation Centre for Education)

## What is CGI and its purpose

- CGI (common gateway interface) is a protocol used in web development to facilitate communication between web servers and external programs or scripts. Here are points about CGI:
  - Dynamic content generation.
  - Language independence.
  - User interaction.
  - Form handling.
  - Server-side processing.
  - Personalization.
  - Database interaction.
  - Customization and extension.
  - Scalability.
  - Legacy usage.
  - Request handling.
  - Python script invocation.
  - Data extraction with CGI.
  - Processing based on input.
  - Dynamic content generation.
  - HTTP response.
  - User interaction and engagement .
  - Relevance and considerations.

Figure 6-12. What is CGI and its purpose

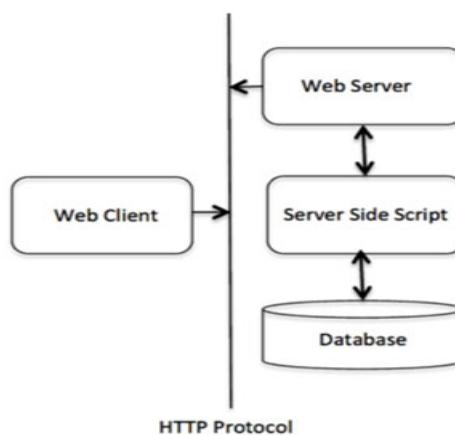


Figure 3: CGI Architecture Diagram

[[https://www.tutorialspoint.com/python/python\\_cgi\\_programming.htm](https://www.tutorialspoint.com/python/python_cgi_programming.htm)]

The primary purpose of CGI is to enable the execution of these programs in response to HTTP requests, thereby allowing for dynamic content generation on the web.

Here are points about CGI and its purpose:

#### **Dynamic content generation:**

- CGI enables the creation of dynamic web content by allowing web servers to execute programs or scripts on the server side. This dynamic content can include HTML pages, images, or other multimedia elements that change based on user input or other parameters.

#### **Language independence:**

- CGI is language-independent, meaning it can be used with various programming languages, such as Python, Perl, or C. This flexibility allows developers to choose the language that best suits their needs.

#### **User interaction:**

- CGI plays a crucial role in enhancing user interaction with web applications. It allows for the processing of user input, such as form submissions, and generating responses dynamically.

#### **Form handling:**

- CGI is commonly used for handling HTML forms on the web. When a user submits a form, the data is sent to a CGI script, which processes the input and generates a response.

#### **Server-side processing:**

- CGI scripts are executed on the server side, allowing for server-side processing of data and reducing the workload on the client (user's browser).

#### **Personalization:**

- CGI facilitates the creation of personalized web experiences by allowing applications to tailor content based on user-specific parameters or preferences.

#### **Database interaction:**

- CGI scripts often interact with databases to retrieve or store information. This enables the development of dynamic web applications with data-driven content.

#### **Customization and extension:**

- CGI provides a customizable and extensible framework for web development. Developers can create modular CGI scripts to add specific functionalities to their web applications.

#### **Scalability:**

- CGI is scalable, allowing developers to build web applications that can handle a large number of simultaneous requests. This makes it suitable for a wide range of applications, from small websites to enterprise-level systems.

#### **Legacy usage:**

- While newer technologies like FastCGI and server-side frameworks have emerged, CGI is still used in certain scenarios, especially for legacy systems or simpler web applications.

### **CGI in web development**

In the realm of web development with Python, the Common Gateway Interface (CGI) plays a crucial role in facilitating communication between web servers and Python scripts, allowing for dynamic and interactive web applications. This process involves several key steps that contribute to the handling of requests and the generation of responses:

- **Request handling:** When a user interacts with a web page, an HTTP request is generated and sent to the web server. The server is configured to recognize certain types of requests and direct them to specific Python scripts through CGI.
- **Python script invocation:** Upon receiving a request, the associated CGI Python script is invoked by the server. The script contains Python code responsible for processing the incoming request.
- **Data extraction with CGI:** The CGI module in Python is commonly used within CGI scripts to extract data from the incoming request. The FieldStorage class is often employed to parse and retrieve data sent via methods like GET or POST.

- **Processing based on input:** Once the data is extracted, the Python script performs actions based on the received input. This may involve data processing, database interactions, or the execution of specific functionalities tied to the nature of the request.
- **Dynamic content generation:** The outcome of the processing is used to dynamically generate content that will be sent back to the client's web browser.
- **HTTP response:** The Python script generates an HTTP response, including the processed data and relevant HTTP headers. The "Content-type" header specifies the type of data being sent, such as HTML, JSON, or other formats.
- **User interaction and engagement:** The dynamic content generated by the CGI Python script contributes to an interactive user experience. User interactions, such as form submissions, trigger requests and responses, allow for seamless interaction with the web application.
- **Relevance and considerations:** While CGI is historically significant, modern web development often leans towards more advanced frameworks like Flask or Django. CGI remains relevant in scenarios where simplicity and direct control over server interactions are desired.

In summary, CGI in web development with Python is a process that involves receiving and processing requests, leveraging the Common Gateway Interface to generate content and provide an interactive user experience dynamically. Understanding CGI provides a foundational knowledge of server-side interactions, which can be valuable in specific development scenarios.



# Handling HTTP requests

- Handling HTTP requests is a fundamental aspect of web development, and within the context of CGI python scripts, this process involves several steps.
- Let's delve into the details of how HTTP requests are managed:
  - [Server configuration](#).
  - [User interaction](#).
  - [Request routing](#).
  - [Directory structure](#).
  - [CGI scripts](#).

*Figure 6-13. Handling HTTP requests*

## Server configuration

Server configuration for handling HTTP requests involves setting up the webserver to recognize specific types of requests, defining rules for routing these requests to the appropriate handlers (such as CGI scripts) and configuring various server settings. Here is an overview of how server configuration is typically done:

- Web server installation: Choose and install web server software on your server machine. Common web servers include Apache, Nginx, or Microsoft Internet Information Services (IIS).
- Configuration files: Web servers use configuration files to define how they should operate. The location of these files depends on the server software. For example:
  - Apache: httpd.conf or .htaccess files.
  - Nginx: nginx.conf file.
  - IIS: Configuration settings are often managed through the IIS Manager.
- CGI configuration: Ensure that CGI is enabled in the server configuration. This involves specifying the directory where CGI scripts are located and configuring the server to execute them.
- Here's a simple example for configuring CGI in Apache's httpd.conf file:

```
<Directory "/path/to/cgi/scripts">
```

```
  Options +ExecCGI
```

```
  AddHandler cgi-script .py
```

```
</Directory>
```

This configuration enables the execution of CGI scripts with the .py extension in the specified directory.

Remember that the specific steps and configuration syntax can vary depending on the web server software being used. Always refer to the documentation of the chosen server for detailed instructions on server configuration.

### User interaction

When a user interacts with a web page by submitting a form, clicking a link, or making any other request, an HTTP request is generated. This request contains information about the user's action, such as form data or the requested URL.

### Request routing

The web server receives the HTTP request and uses its configuration to route the request to the appropriate CGI Python script. The routing is based on factors such as the type of request, the URL, or other specified criteria.

**Server configuration:** Assuming Apache is being used, the configuration file (e.g., httpd.conf) or a .htaccess file is employed. CGI is enabled, and a handler for CGI scripts is set up.

```
# CGI is enabled
```

```
Options +ExecCGI
```

```
AddHandler cgi-script .cgi
```

**Request routing:** Rules are set up to route requests to the appropriate CGI scripts based on the URL.

```
# Requests for articles are routed to the article.cgi
```

```
<Location "/articles">
```

```
    CGI scripts are set as handlers
```

```
    ScriptAlias "/articles" "/path/to/article.cgi"
```

```
</Location>
```

```
# Requests for profiles are routed to the profile.cgi
```

```
<Location "/profiles">
```

```
    CGI scripts are set as handlers
```

```
    ScriptAlias "/profiles" "/path/to/profile.cgi"
```

```
</Location>
```

### In this example:

Requests to /articles are routed to the article.cgi script.

Requests to /profiles are routed to the profile.cgi script.

**Directory structure:** The directory structure is organized to reflect the configured paths.

```
/path/to/
```

```
|—— article.cgi
```

**CGI scripts:** CGI scripts (article.cgi and profile.cgi) are created to handle the logic for each section.

article.cgi:

```
#!/usr/bin/env python
```

## Receiving and processing requests



IBM ICE (Innovation Centre for Education)

- In the realm of CGI python, the process of receiving and processing requests involves the utilization of the common gateway interface to facilitate communication between web servers and python scripts.
- Upon receipt of a request, the CGI script is invoked, and the associated python code executes. Within this script, the CGI module is commonly employed to parse and extract information from the incoming request.
- The FieldStorage class from the CGI module facilitates the retrieval of data sent via methods like GET or POST. This data often includes user inputs from HTML forms. Once the data is extracted, the python script can perform various actions based on the received input.

*Figure 6-14. Receiving and processing requests*

This may involve processing the data, interacting with databases, or executing specific functionalities tied to the nature of the request. For instance, a CGI python script might process form data, perform calculations, or generate a dynamic HTML response.

- CGI module in python.
- Understanding CGI environment variables.
- HTTP header.
- Handling HTTP methods: GET and POST.

### CGI module in python

The CGI module is utilized in Python, offering numerous built-in functions, and can be employed by importing the CGI module.

```
import cgi
```

Subsequently, further script development can be performed:

```
import cgi
cgitb.enable()
```

The above script will activate an exception handler to display a detailed report of any errors that occur in the web browser. Additionally, the report can be saved using the following script:

```
import cgitb
```

```
cgitb.enable(display=0, logdir="/path/to/logdir")
```

This feature of the CGI module proves beneficial during script development, providing detailed reports for effective debugging. Once the expected output is achieved, this feature can be removed.

To retrieve that information from the form, Python offers the FieldStorage class. The encoding keyword parameter can be applied to the document if the form contains non-ASCII characters. The content within the <META> tag in the <HEAD> section of the HTML document can be found.

The FieldStorage class reads form information from the standard input or the environment. A FieldStorage instance is similar to a Python dictionary. The len() function and other dictionary functions can be used with the FieldStorage instance, overlooking fields with empty string values. Empty values can also be considered using the optional keyword parameter keep\_blank\_values by setting it to True.

```
form = cgi.FieldStorage()
if ("name" not in form or "addr" not in form):
    print("<H1>Error</H1>")
    print("Please enter information in the name and address fields.")
    return
print("<p>name:", form["name"].value)
print("<p>addr:", form["addr"].value)
# The next lines of code will execute here...
```

In the above example, the form ["name"] is used, where "name" is the key, to extract the value entered by the user. The getvalue() method can be used to fetch the string value directly. This function also takes an optional second argument as a default. If the key is not present, it returns the default value. Suppose the submitted form data has more than one field with the same name, the form.getlist() function should be used. It returns a list of strings. In the following code, any number of username fields, separated by commas, can be added.

```
value1 = form.getlist("username")
usernames1 = ",".join(value)
```

If the field is an uploaded file, it can be accessed by the value attribute or the getvalue() method, and the uploaded file can be read as bytes.

## Understanding CGI environment variables

Variable Name	Description
CONTENT_TYPE	The data type of the content used in POST requests for file uploads.
CONTENT_LENGTH	The length of the query information, available only for POST requests.
HTTP_COOKIE	Returns set cookies in the form of key-value pairs.
HTTP_USER_AGENT	Contains information about the user agent (web browser).
PATH_INFO	The path for the CGI script.
QUERY_STRING	URL-encoded information sent with GET method requests.
REMOTE_ADDR	The IP address of the remote host making the request.
REMOTE_HOST	The fully qualified name of the host making the request.
REQUEST_METHOD	The method used to make the request, commonly GET or POST.
SCRIPT_FILENAME	The full path to the CGI script.
SCRIPT_NAME	The name of the CGI script.
SERVER_NAME	The server's hostname or IP Address.
SERVER_SOFTWARE	The name and version of the software the server is running.

Table 3: CGI environment variables

## HTTP header

The HTTP header includes crucial information sent to the browser to define the content type and other attributes. The Content-type: text/html\r\n\r\n line, for instance, specifies that the content type is HTML. Various HTTP headers include:

Header Name	Description
Content-Type	A MIME string defining the format of the returned file (e.g., text/html).
Expires	The date the information becomes invalid.
Location	The URL returned instead of the requested URL, useful for redirection.
Last-Modified	The date of the last modification of the resource.
Content-Length	The length, in bytes, of the data being returned.
Set-Cookie	Sets a cookie passed through the string.

Table 4: HTTP header

## Handling HTTP methods: GET and POST

### GET method

The GET method involves sending encoded user information appended to the page request. The information is separated by the? Character in the URL, like:

<http://www.test.com/cgi-bin/hello.py?key1=value1&key2=value2>

GET is the default method for passing information, but it's unsuitable for sensitive data and has a size limitation of 1024 characters. Information sent via GET is accessible in a CGI program through the QUERY\_STRING environment variable. Here's an example using the GET method:

/cgi-bin/hello\_get.py?first\_name=Malhar&last\_name=Lathkar

The given URL "/cgi-bin/hello\_get.py?first\_name=Malhar&last\_name=Lathkar" can be broken down into the following components:

- **Path: "/cgi-bin/hello\_get.py"**

This part of the URL specifies the path to the resource on the server's filesystem or within the web application. In this case, it indicates that the resource is located in the "/cgi-bin/" directory and is named "hello\_get.py".

- **Query Parameters: "?first\_name=Malhar&last\_name=Lathkar"**

The query parameters are appended to the URL after a question mark ("?"). They provide additional data to be sent to the server for processing. In this example, there are two query parameters:

- "first\_name" with the value "Malhar"
  - "last\_name" with the value "Lathkar"
- Overall, this URL is likely used to access a Python script named "hello\_get.py" located in the "/cgi-bin/" directory of a web server. The script may process the provided query parameters (first\_name and last\_name) and generate a response accordingly.

The hello\_get.py script retrieves and handles this input:

```
import cgi
form = cgi.FieldStorage()
first_name = form.getvalue('first_name')
last_name = form.getvalue('last_name')
print("Content-type:text/html\r\n\r\n")
print("<html>")
```

```

print("<head>")
print("<title>Hello - Second CGI Program</title>")
print("</head>")
print("<body>")
print("<h2>Hello %s %s</h2>" % (first_name, last_name))
print("</body>")
print("</html>")

[https://www.tutorialspoint.com/common-gateway-interface#:~:text=CGI%20programs%20are%20written%20in,in%20writing%20the%20CGI%20program.]

```

### **POST method:**

The POST method provides a more reliable way to pass information to a CGI program. It sends information as a separate message rather than appending it to the URL. This information comes into the CGI script as standard input.

The same hello\_get.py script can handle both GET and POST methods:

```

import cgi

form = cgi.FieldStorage()

first_name = form.getvalue('first_name')
last_name = form.getvalue('last_name')

print("Content-type:text/html\r\n\r\n")

[[https://www.tutorialspoint.com/common-gateway-interface#:~:text=CGI%20programs%20are%20written%20in,in%20writing%20the%20CGI%20program.]

```

The code imports the CGI module and initializes a form object using the FieldStorage() method. Subsequently, the values of the 'first\_name' and 'last\_name' fields are retrieved from the form using the getvalue() method. Finally, the code prints the content type for an HTML response with "Content-type: text/html\r\n\r\n". "Content-type: text/html\r\n\r\n" is an HTTP header that tells the recipient (typically a web browser) that the content of the response is in HTML format.

The carriage return and line feed characters separate this header from the actual HTML content in the HTTP response. This helps the receiving browser correctly interpret and render the content of the web page.

Below are Python CGI programs demonstrating how to pass different types of form data (checkbox, radio button, text area, and drop-down box) to a CGI program:

### **Passing Checkbox Data to CGI Program:**

```

#!/usr/bin/env python3

import cgi

print("Content-type:text/html\r\n\r\n")

print("<html>")
print("<head>")
print("<title>Checkbox Data</title>")
print("</head>")
print("<body>")
print("<h2>Checkbox Data</h2>")

form = cgi.FieldStorage()

if "checkbox_data" in form:
    checkbox_values = form.getlist("checkbox_data")

```

```

print("<p>Selected checkboxes:</p>")
print("<ul>")
for value in checkbox_values:
    print("<li>{}</li>".format(value))
print("</ul>")
else:
    print("<p>No checkboxes selected.</p>")
print("</body>")
print("</html>")

```

### **Passing Radio Button Data to CGI Program:**

```

#!/usr/bin/env python3
import cgi
print("Content-type:text/html\r\n\r\n")
print("<html>")
print("<head>")
print("<title>Radio Button Data</title>")
print("</head>")
print("<body>")
print("<h2>Radio Button Data</h2>")
form = cgi.FieldStorage()
if "radio_data" in form:
    radio_value = form.getvalue("radio_data")
    print("<p>Selected radio button: {}</p>".format(radio_value))
else:
    print("<p>No radio button selected.</p>")
print("</body>")
print("</html>")

```

### **Passing Text Area Data to CGI Program:**

```

#!/usr/bin/env python3
import cgi
print("Content-type:text/html\r\n\r\n")
print("<html>")
print("<head>")
print("<title>Text Area Data</title>")
print("</head>")
print("<body>")
print("<h2>Text Area Data</h2>")
form = cgi.FieldStorage()
if "textarea_data" in form:
    textarea_value = form.getvalue("textarea_data")

```

```
    print("<p>Text area value: {}</p>".format(textarea_value))
else:
    print("<p>No data entered in the text area.</p>")
print("</body>")
print("</html>")

Passing Drop Down Box Data to CGI Program:
#!/usr/bin/env python3

import cgi

print("Content-type:text/html\r\n\r\n")
print("<html>")
print("<head>")
print("<title>Drop Down Box Data</title>")
print("</head>")
print("<body>")
print("<h2>Drop Down Box Data</h2>")

form = cgi.FieldStorage()

if "dropdown_data" in form:
    dropdown_value = form.getvalue("dropdown_data")
    print("<p>Selected option from drop-down box: {}</p>".format(dropdown_value))
else:
    print("<p>No option selected from the drop-down box.</p>")

print("</body>")
print("</html>")
```

These programs demonstrate how to retrieve data from different types of form elements in a CGI program written in Python.



IBM ICE (Innovation Centre for Education)

# Generating HTTP responses

- In the realm of web development, generating HTTP responses is a fundamental aspect that involves sending data from the server to the client (usually a web browser).
- In python, Common Gateway Interface (CGI) scripting is one of the ways to accomplish this.
- Let's delve into the process of generating HTTP responses using python CGI in more detail:
  - [HTTP response basics](#).
  - [HTTP header types](#).
  - [CGI script execution flow](#).
  - [Handling dynamic data](#).
  - [Error handling and debugging](#).

---

Figure 6-15. Generating HTTP responses

## HTTP response basics

An HTTP response is composed of two main parts: headers and body. The headers provide essential information about the response, such as the content type, status code, and additional metadata. The body contains the actual content that the browser will render.

## HTTP Header types

The common types of HTTP headers used in Python CGI programming are presented in a table format:

Header Type	Description	Example
Content-Type	Specifies the media type (MIME type) of the response content.	'Content-Type: text/html'
Location	Specifies the URL to which the client should be redirected.	'Location: http://example.com/new_location'
Set-Cookie	Sets cookies in the client's browser.	'Set-Cookie: name=value; expires=Sun, 06 Nov 1994 08:49:37 GMT'
Cache-Control	Specifies caching directives for the client or intermediate proxies.	'Cache-Control: no-cache'
Content-Disposition	Provides a suggested filename and disposition for downloaded files.	'Content-Disposition: attachment; filename="example.txt"'
Expires	Specifies the expiration date/time of the response content.	'Expires: Sat, 01 Jan 2025 00:00:00 GMT'
ETag	Provides entity tagging to uniquely identify versions of a resource.	'ETag: "123456789"'
Access-Control-Allow-Origin	Specifies which origins are allowed to access the resource in Cross-Origin Resource Sharing (CORS).	'Access-Control-Allow-Origin: *'
WWW-Authenticate	Used for HTTP authentication challenges.	'WWW-Authenticate: Basic realm="Access to the site"'

Table 5: Common HTTP Header types

## HTTP Header types

### Content-type header:

- Description: The Content-Type header specifies the media type (MIME type) of the response content. It informs the client about the type of data being sent, such as HTML, JSON, XML, images, etc.
- Why it's used: This header is essential for the client to interpret the response correctly. For example, if the Content-Type is set to "text/html", the client will render the response as an HTML document.

### Location header:

- Description: The Location header is used in redirection to specify the URL to which the client should be redirected.
- Why it's used: This header is crucial for implementing URL redirection. It instructs the client's browser to navigate to a different location, often used for handling page redirects or implementing URL shortening services.

### Set-cookie header:

- Description: The Set-Cookie header sets cookies in the client's browser. Cookies are small pieces of data sent from a website and stored in the user's browser while the user is browsing.
- Why it's used: Cookies enable websites to store user-specific information, such as login sessions, shopping cart contents, or user preferences. They are commonly used for session management, personalization, and tracking user behavior.

### Cache-control header:

- Description: The Cache-Control header specifies caching directives for the client or intermediate proxies, controlling how responses are cached.

- Why it's used: Caching improves performance and reduces server load by storing copies of frequently accessed resources. This header allows servers to control caching behavior, such as specifying whether a response can be cached, for how long, and under what conditions.

#### **Content-disposition header:**

- Description: The Content-Disposition header provides a suggested filename and disposition for downloaded files. It is often used for file downloads initiated by the server.
- Why it's used: This header allows servers to suggest filenames for downloaded files and specify whether they should be displayed inline or downloaded as attachments. It enhances the user experience by providing meaningful filenames and controlling how files are handled by the browser.

#### **Expires header:**

- Description: The Expires header specifies the expiration date/time of the response content. It indicates to the client when the content becomes invalid and should be re-fetched from the server.
- Why it's used: By setting an expiration date for content, servers can optimize caching and reduce unnecessary requests. Clients can store cached responses until they expire, reducing network traffic and improving performance.

#### **ETag header:**

- Description: The ETag header provides entity tagging to uniquely identify versions of a resource. It is used for cache validation and conditional requests.
- Why it's used: ETags allow servers and clients to efficiently determine if a cached resource is still valid by comparing its ETag value with the current version on the server. This enables conditional requests, reducing bandwidth usage and server load.

#### **Access-control-allow-origin header:**

- Description: The Access-Control-Allow-Origin header specifies which origins are allowed to access the resource in Cross-Origin Resource Sharing (CORS). It controls access to resources from different domains.
- Why it's used: CORS is a security feature that restricts cross-origin requests to protect against cross-site scripting attacks. This header allows servers to explicitly specify which domains are allowed to access their resources, enhancing security while enabling controlled data sharing between origins.

#### **WWW-authenticate header:**

- Description: The WWW-Authenticate header is used for HTTP authentication challenges. It prompts the client to authenticate itself using a supported authentication method.
- Why it's used: This header is essential for securing restricted resources and controlling access to protected areas of a website. It initiates the authentication process by informing the client which authentication schemes are supported by the server.
- These HTTP headers play critical roles in controlling various aspects of the HTTP communication between clients and servers, enhancing security, performance, and functionality of web applications.

#### **Real-Time example: Shopping website**

##### **Personalized recommendations:**

- When a user visits a shopping website, cookies can be used to track their browsing behavior and product interactions.
- Based on this information, the website can recommend personalized products to the user, such as items similar to those they have viewed or purchased in the past.
- For example, if a user frequently browses electronics, the website may recommend new gadgets or accessories that match their interests.

##### **Shopping cart persistence:**

- Cookies can store information about items added to the shopping cart, allowing users to continue their shopping session across multiple visits or pages.

- When a user adds items to their cart and navigates to different pages or even closes the browser, the cart contents can be preserved using cookies.
- This ensures a seamless shopping experience, as users can easily pick up where they left off without losing their selected items.

### **User preferences and settings:**

- Cookies can store user preferences and settings, such as preferred language, currency, or display options.
- By storing these preferences in cookies, the website can customize the user experience based on their individual preferences without requiring them to re-enter the information each time they visit.
- For example, if a user selects a specific currency for pricing, the website can remember this preference using cookies and display prices in the chosen currency during future visits.

### **Login sessions and account management:**

- Cookies play a crucial role in managing user login sessions and maintaining authentication status.
- When a user logs in to their account on the shopping website, a session cookie is typically set to identify them as a logged-in user.
- This session cookie allows the user to access restricted areas of the website, such as their order history, saved addresses, or payment methods, without needing to log in repeatedly.

### **Persistent login and remember me functionality:**

- Cookies can provide the "Remember Me" functionality, allowing users to stay logged in across multiple sessions.
- By storing a persistent login token in a cookie, users can bypass the login screen and access their account directly during subsequent visits to the website.
- This feature enhances convenience for users who prefer not to enter their credentials each time they visit the website, while still maintaining security through encrypted tokens.

In summary, cookies play a vital role in enhancing the user experience on shopping websites by enabling personalized recommendations, preserving shopping cart contents, storing user preferences, managing login sessions, and providing convenient login options. However, it's essential for websites to handle cookies responsibly, respect user privacy, and comply with relevant regulations such as GDPR and CCPA to maintain trust and transparency with users.

### **Code**

Suppose we have a simple shopping website with three products: Laptop, Smartphone, and Headphones. Users can click on a button to add items to their shopping cart. We'll use cookies to store the shopping cart information.

Here's the Python CGI script for our shopping website:

```
#!/usr/bin/env python3

import os

# Function to retrieve cookies from the HTTP_COOKIE environment variable
def get_cookies():
    cookie_string = os.environ.get('HTTP_COOKIE', '')
    cookies = {}
    if cookie_string:
        for cookie in cookie_string.split('; '):
            key, value = cookie.split('=')
            cookies[key] = value
```

```

        return cookies

# Function to set a cookie
def set_cookie(name, value):
    print(f"Set-Cookie: {name}={value}; Path=/")

# Function to display the shopping cart
def display_shopping_cart():
    print("<h2>Shopping Cart:</h2>")
    print("<ul>")
    for item in shopping_cart:
        print(f"<li>{item}</li>")
    print("</ul>")

# Retrieve the shopping cart from cookies
cookies = get_cookies()
shopping_cart = cookies.get('shopping_cart', '').split(',')
# Handle adding items to the shopping cart
form_data = os.environ.get('QUERY_STRING', '')
if form_data:
    product = form_data.split('=')[1] # Extract the product name from the form data
    shopping_cart.append(product)
    set_cookie('shopping_cart', ','.join(shopping_cart))

# Generate HTML response
print("Content-Type: text/html\r\n\r\n")
print("<html>")
print("<head><title>Shopping Website</title></head>")
print("<body>")
print("<h1>Welcome to our Shopping Website!</h1>")
print("<h2>Products:</h2>")
print("<ul>")
print("<li>Laptop <a href='?product=laptop'>Add to Cart</a></li>")
print("<li>Smartphone <a href='?product=smartphone'>Add to Cart</a></li>")
print("<li>Headphones <a href='?product=headphones'>Add to Cart</a></li>")
print("</ul>")
display_shopping_cart()
print("</body>")
print("</html>")

```

**Explanation:**

- The script defines functions to retrieve cookies (`get_cookies()`), set cookies (`set_cookie()`), and display the shopping cart (`display_shopping_cart()`).
- It retrieves the shopping cart information from cookies using the `get_cookies()` function.

- When a user clicks the "Add to Cart" link for a product, the product name is extracted from the query string (form\_data) and added to the shopping cart list.
- The updated shopping cart information is then stored in a cookie using the set\_cookie() function.
- The HTML response includes a list of products with "Add to Cart" links for each item, as well as the current contents of the shopping cart.

This example demonstrates a basic implementation of a shopping website using cookies to store and update the shopping cart information. Users can add items to their cart, and their selections are preserved across different page views using cookies.

### **CGI script execution flow**

The sequence of steps listed below illustrates how a CGI script written in Python processes an incoming HTTP request, generates a dynamic response, and sends it back to the client through interaction with the web server. Each step plays a crucial role in the overall execution flow, enabling the CGI script to handle client requests and provide dynamic content dynamically.

- **Web Server:** The process begins when the web server receives an HTTP request from a client, typically a web browser.
- **CGI Script:** Upon receiving the request, the web server identifies that the requested resource is a CGI script. It then executes the CGI script in response to the request. In this case, the CGI script is written in Python.
- **Environment variables:**

Before executing the CGI script, the web server sets various environment variables containing information about the request.

These environment variables include details such as the request method (REQUEST\_METHOD), query parameters (QUERY\_STRING), HTTP headers (HTTP\_\*), and more.

The CGI script accesses this information through environment variables to determine how to process the request.

- **Parse and process request data:**

The CGI script parses and processes the request data based on the information available in the environment variables.

This step involves extracting relevant data from the environment variables, such as form data, query parameters, or other request details.

The script may perform actions based on the received data, such as retrieving database records, performing calculations, or generating dynamic content.

- **Response generation:**

Based on the processed request data and application logic, the CGI script generates the content for the HTTP response dynamically.

This content could include HTML code, JSON data, or any other type of response relevant to the client's request.

The response content is typically generated using Python code within the CGI script, incorporating any necessary data manipulation or formatting.

- **HTTP header generation:**

In addition to generating the response content, the CGI script also generates HTTP headers as needed for the response.

These headers provide metadata about the response, such as the content type (Content-Type), cache-control directives, cookie settings (Set-Cookie), and more.

The CGI script sets these headers based on the requirements of the response and any specific instructions from the client or server.

- **HTTP response to client:**

Once the CGI script has generated both the response content and headers, it sends the complete HTTP response back to the web server.

The response includes both the headers and content, forming a complete HTTP response message ready for transmission.

- **Web server response to client:**

Finally, the web server sends the HTTP response received from the CGI script back to the client, typically a web browser.

The client receives the response and processes it accordingly, rendering the content to the user or performing any additional actions specified in the response.

When a client requests a CGI script, the server executes the script. The script, in turn, generates an HTTP response dynamically based on its logic and sends it back to the client. This dynamic generation of content is what distinguishes CGI scripts from static web pages.



- **Handling dynamic data**

In more complex scenarios, CGI scripts often involve handling dynamic data. This can include processing form submissions, interacting with databases, or generating content based on user input. The `cgi` module in Python provides functionalities for handling form data and interacting with the environment.

- **Error handling and debugging**

Proper error handling and debugging mechanisms are essential in CGI scripting. Techniques such as using the `cgitb` module to enable detailed error reports in the browser can aid in identifying and fixing issues during the development phase.

Generating HTTP responses in Python CGI involves setting appropriate headers and crafting the body content dynamically. Understanding the basics of HTTP, headers, and CGI execution flow is crucial for developing dynamic web applications using Python. Additionally, handling dynamic data, error reporting, and debugging are integral parts of the CGI scripting process.

## Building interactive web applications



IBM ICE (Innovation Centre for Education)

- Building interactive web applications through CGI (Common Gateway Interface) in python involves a server-side approach where python scripts are executed to handle dynamic content generation.
- While CGI is a traditional method for web development, it's worth noting that modern web frameworks like Flask and Django are more commonly used for building interactive web applications In python.

---

*Figure 6-16. Building interactive web applications*

To use CGI for web development, python scripts are written to handle different aspects of web interactions, such as form submissions and data processing. These scripts are executed on the server, and the results are sent back to the client's web browser.



IBM ICE (Innovation Centre for Education)

# Form handling with CGI

- In the realm of web development, creating interactive web applications often involves handling user input through forms.
- In the context of python CGI scripting, form handling is a crucial aspect that enables developers to collect and process data submitted by users.
- Let's explore the process of building interactive web applications with a focus on form handling using CGI:
  - [Introduction to form handling in CGI](#).
  - [Setting up the HTML form](#).
  - [Creating the CGI script for form processing](#).
  - [Handling form data](#) .
  - [Dynamic web application flow](#).

*Figure 6-17. Form handling with CGI*

## Introduction to form handling in CGI

Form handling is a mechanism that allows users to submit data to a server, typically via an HTML form. In CGI scripting, the CGI module in Python provides functionalities to retrieve and process form data. This enables the creation of dynamic web applications that can respond to user input.

### Setting up the HTML form

Begin by creating an HTML form that users will interact with. This form should specify the HTTP method (usually POST for form submissions) and the action, which is the URL of the CGI script that will process the form data.

```
html
Copy code
<form action="/cgi-bin/process_form.cgi" method="post">
  <label for="username">Username:</label>
  <input type="text" id="username" name="username" required>
  <br>
  <label for="password">Password:</label>
  <input type="password" id="password" name="password" required>
  <br>
```

```

<input type="submit" value="Submit">
</form>
```

### **Creating the CGI script for form processing**

Develop a CGI script, for example, process\_form.cgi, that will handle the form data. The script should extract the form parameters, perform any necessary processing, and generate an appropriate response.

```

#!/usr/bin/env python3
import cgi
# Enable detailed error reports in the browser during development.
cgi.cgitb.enable()
# Set content type to HTML.
print("Content-Type: text/html\n\n")
# Access form data using the FieldStorage class.
form = cgi.FieldStorage()
# Extracting values from the form.
username = form.getvalue("username")
password = form.getvalue("password")
# Process the form data.
# (Add the logic here based on the form data)
# Generate an HTML response.
print(f"""
<html>
    <head>
        <title>Form Processing Result</title>
    </head>
    <body>
        <h1>Form Processing Result</h1>
        <p>Username: {username}</p>
        <p>Password: {password}</p>
        <!-- Add more content based on your processing -->
    </body>
</html>
""")
```

### **Handling form data**

Form data is accessed and processed by the CGI script. Values are extracted from the form, and the processing logic is executed. The HTML response is generated dynamically based on the submitted data, providing an interactive experience for the user.

### **Dynamic web application flow**

- The user interacts with the HTML form presented in the browser.
- Upon submission, the form data is sent to the CGI script specified in the form's action attribute.
- The CGI script retrieves the form parameters using the FieldStorage class.
- Values are extracted and processed according to the script's logic.

- The script generates an HTML response dynamically, displaying the processed form data.

# Implementing data processing



IBM ICE (Innovation Centre for Education)

- Implementation of data processing.
- Web application description.
- Components of the implementation.
- Data processing tasks.
- Practical application of CGI scripting.

---

Figure 6-18. Implementing data processing

### Implementation of data processing:

- Achieved through the creation of interactive web applications using HTML and Python CGI scripting.
- Focuses on developing a web-based course registration system for college students.

### Web application description:

- Provides a user-friendly web form for students to register for courses.
- Allows students to select courses, submit the registration form, and receive a confirmation message.

### Components of the implementation:

- HTML Form: Named "registration\_form.html," it serves as the user interface.
- Allows students to input their name, email, and select multiple courses from a list.
- Python CGI Script: Named "process\_registration.py," it handles the form data and confirmation page generation.
- Processes the submitted form data.
- Generates a confirmation page dynamically.
- Displays the registered courses along with the student's name and email.

### Data processing tasks:

- Extraction of Values: Involves extracting values from the submitted form, such as student's name, email, and selected courses.

- HTML Content Generation: Dynamically generates HTML content for the confirmation page.
- Presentation of Finalized Registration Details: Presents the finalized registration details to the user on the confirmation page.

### Practical application of CGI scripting:

- Demonstrates the practical application of CGI scripting for:
  - Handling user input.
  - Processing data submitted through web forms.
  - Delivering dynamic content to users based on their input.

Overall, this example illustrates how HTML and Python CGI scripting can be used together to create interactive web applications that process user input, generate dynamic content, and provide a seamless user experience, as demonstrated through the course registration system described.

### HTML form (registration\_form.html)

```
<!-- File: registration_form.html -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Course Registration</title>
</head>
<body>
  <h1>Course Registration</h1>
  <form action="/cgi-bin/process_registration.py" method="post">
    <label for="name">Name:</label>
    <input type="text" id="name" name="name" required>
    <br>
    <label for="email">Email:</label>
    <input type="email" id="email" name="email" required>
    <br>
    <label for="courses">Select Courses:</label>
    <select id="courses" name="courses" multiple required>
      <option value="math101">Math 101</option>
      <option value="eng102">English 102</option>
      <option value="hist105">History 105</option>
      <!-- Add more courses as needed -->
    </select>
    <br>
    <input type="submit" value="Submit Registration">
  </form>
</body>
```

```

</html>

Python CGI script (process_registration.py)
Python
Copy code
#!/usr/bin/env python3
import cgi
# Enable detailed error reports in the browser during development.
cgi.cgitb.enable()
# Set content type to HTML.
print("Content-Type: text/html\n\n")
# Access form data using the FieldStorage class.
form = cgi.FieldStorage()
# Extracting values from the form.
name = form.getvalue("name")
email = form.getvalue("email")
selected_courses = form.getlist("courses")
# Generate an HTML response.
print(f"""
<html>
    <head>
        <title>Registration Confirmation</title>
    </head>
    <body>
        <h1>Registration Confirmation</h1>
        <p>Name: {name}</p>
        <p>Email: {email}</p>
        <p>Selected Courses:</p>
        <ul>
""") 
for course in selected_courses:
    print(f"            <li>{course}</li>")
print("""
        </ul>
        <p>Thank you for registering!</p>
    </body>
</html>
""")

```

**Steps to run:**

- Save files: Save the HTML form as registration\_form.html and the Python script as process\_registration.py.

- File placement: Place both files in the appropriate directory accessible by the web server, for example, /var/www/html.
- Permissions: Ensure the Python script has executable permissions (chmod +x process\_registration.py).
- Start web server: Initiate the web server.
- Access application: Open a web browser and navigate to [http://localhost/registration\\_form.html](http://localhost/registration_form.html) (replace localhost with the server address).

## Conclusion

Concluding the exploration across GUI development and web programming with CGI establishes a thorough understanding of these domains. The intricacies of GUI programming were explored, showcasing practical applications through examples that highlighted the potential of GUIs in real-world scenarios. Moving forward, the dynamics of handling HTTP requests and generating responses in web programming with CGI were uncovered, empowering developers to navigate the dynamic realm of web development. As a whole, this journey acts as a valuable compass for developers aiming to elevate their programming skills, encompassing a multitude of possibilities from GUI development to web programming with CGI.

## Self evaluation: Exercise 25



IBM ICE (Innovation Centre for Education)

- **Exercise 25:** Simple tkinter GUI
- **Estimated time:** 00:15 minutes
- **Aim:** Design a simple tkinter GUI application with buttons and labels. Implement functionality to update labels when buttons are clicked
- **Learning objective:**
  - The primary objective of this exercise is to introduce learners to building graphical user interfaces (GUIs) using the tkinter library in Python. Participants will gain hands-on experience in creating basic GUI components and implementing event-driven functionality
- **Learning outcome:**
  - Upon completing this exercise, learners will have acquired foundational skills in GUI development with Python's tkinter library. They will be able to design simple interactive applications, laying the groundwork for more complex GUI projects

---

Figure 6-19. Self evaluation: Exercise 25

Self evaluation exercise 25 is as stated above:



IBM ICE (Innovation Centre for Education)

## Self evaluation: Exercise 26

- **Exercise 26:** GUI Button Actions
- **Estimated time:** 00:15 minutes
- **Aim:** Create a Python program that responds to user button clicks in a tkinter GUI. Perform actions like displaying messages when buttons are clicked
- **Learning objective:**
  - This exercise aims to familiarize learners with handling button actions in a tkinter graphical user interface (GUI). Participants will gain practical experience in associating actions with button clicks and enhancing interactivity in GUI applications
- **Learning outcome:**
  - By completing this exercise, learners will acquire the skills to design tkinter GUIs with responsive buttons. They will understand how to define actions for buttons and improve the user experience in their Python GUI applications

---

Figure 6-20. Self evaluation: Exercise 26

Self evaluation exercise 26 is as stated above:



IBM ICE (Innovation Centre for Education)

## Self evaluation: Exercise 27

- **Exercise 27:** HTML Form Handling
- **Estimated time:** 00:15 minutes
- **Aim:** Build an HTML web form with input fields and a submit button. Create a Python CGI script to handle form submissions and display the entered data
- **Learning objective:**
  - This exercise aims to introduce learners to handling HTML forms using Python CGI (Common Gateway Interface) scripts. Participants will learn how to create a simple web form, submit data, and process it using Python on the server side
- **Learning outcome:**
  - By completing this exercise, learners will acquire the skills to design HTML forms and implement Python CGI scripts for form handling. They will understand the interaction between web forms and server-side scripts

---

Figure 6-21. Self evaluation: Exercise 27

Self evaluation exercise 27 is as stated above:



IBM ICE (Innovation Centre for Education)

## Self evaluation: Exercise 28

- **Exercise 28:** Form Data Validation
- **Estimated time:** 00:15 minutes
- **Aim:** Extend the previous exercise to validate form data in the Python CGI script. Check for required fields and display validation messages.
- **Learning objective:**
  - This exercise aims to enhance learners' understanding of form data validation in web applications. Participants will explore techniques to validate user input on the server side using Python CGI scripts
- **Learning outcome:**
  - Upon completing this exercise, learners will be equipped with the skills to implement data validation for web forms. They will understand how to check for required fields and provide feedback to users

---

Figure 6-22. Self evaluation: Exercise 28

Self evaluation exercise 28 is as stated above:



IBM ICE (Innovation Centre for Education)

## Self evaluation: Exercise 29

- **Exercise 29:** Dynamic Web Content with CGI
- **Estimated time:** 00:15 minutes
- **Aim:** Implement a Python CGI script that generates dynamic web content based on user requests. Create a simple web application that displays different content based on URL parameters.
- **Learning objective:**
  - This exercise aims to familiarize learners with dynamic web content generation using Python CGI scripts. Participants will understand how to handle user requests, extract parameters from URLs, and dynamically generate content
- **Learning outcome:**
  - Upon completing this exercise, learners will be proficient in creating dynamic web applications using CGI. They will gain insights into handling URL parameters and generating customized content based on user input

---

Figure 6-23. Self evaluation: Exercise 29

Self evaluation exercise 29 is as stated above:

## Checkpoint (1 of 2)



IBM ICE (Innovation Centre for Education)

### Multiple choice questions:

1. Which Python GUI framework is known for its simplicity and is included in the standard library?
  - a) PyQt
  - b) Tkinter
  - c) Kivy
  - d) wxPython
2. In GUI programming, what does the term "event handling" refer to?
  - a) Designing graphical elements
  - b) Handling user interactions
  - c) Defining colors and fonts
  - d) Structuring the application logic
3. Which of the following cannot be processed by a CGI script?
  - a) User form data
  - b) Server configuration files
  - c) Environmental variables
  - d) Database queries

Figure 6-24. Checkpoint (1 of 2)

Write your answers here:

- 1.
- 2.
- 3.

## Checkpoint (2 of 2)



IBM ICE (Innovation Centre for Education)

### Fill in the blanks:

1. In GUI development, the acronym GUI stands for \_\_\_\_\_ User Interface.
2. \_\_\_\_\_ and \_\_\_\_\_ are examples of widgets commonly used in GUI programming for creating buttons and displaying text, respectively.
3. Two types of CGI interfaces are \_\_\_\_\_ and \_\_\_\_\_
4. The header line "Content-type: text/html" within an HTTP response indicates the \_\_\_\_\_

### True/False:

1. In GUI programming, widgets are graphical elements like buttons and labels that users interact with. **True/False**
2. Event handling in GUI programming refers to the process of designing visual elements but does not involve user interactions. **True/False**
3. CGI stands for Common Gateway Interface. **True/False**

---

Figure 6-25. Checkpoint (2 of 2)

Write your answers here:

Fill in the blanks:

- 1.
- 2.
- 3.
- 4.

True or false:

- 1.
- 2.
- 3.

## Question bank (1 of 2)



IBM ICE (Innovation Centre for Education)

### Two mark questions:

1. You are developing a Python GUI application for a quiz game using Kivy. Write a brief code snippet to create a Kivy screen with a Label displaying a quiz question and multiple Button widgets representing answer options. I
2. For the above question, implement event handling to capture the user's selected answer when a button is clicked.
3. What is the primary difference between server-side scripting languages (like PHP or Python) and CGI in the context of dynamic web content generation
4. When a user submits a form on a web page that utilizes CGI, how is the data sent to the CGI script?

### Four mark questions:

1. Imagine you are developing a photo gallery application using a Python GUI framework. The application should display a grid of thumbnail images, and when a thumbnail is clicked, the corresponding full-sized image should be displayed. Follow the given guidelines:

#### Widgets and Layout (2 marks):

Write Python code to create the main window of the photo gallery application. Include a grid of thumbnail images, and ensure that the layout is visually appealing.

#### Event Handling (2 marks):

Implement event handling to respond to a thumbnail click. When a thumbnail is clicked, display the full-sized image in a separate window or section of the application.

Figure 6-26. Question bank (1 of 2)

## Question bank (2 of 2)



IBM ICE (Innovation Centre for Education)

2. Explain CGI and its purpose.
3. Briefly explain the role of a CGI script in processing user input on a web server. (2 marks)

What are the two main types of CGI interfaces, and how do they differ in terms of communication with the web server? (2 marks)

4. Describe the structure of an HTTP request and its key components relevant to CGI processing. (2 marks)

Explain the steps involved in generating an HTTP response from a CGI script. Include the function of common status codes. (2 marks)

### Eight mark questions:

1. Explain the essential steps involved in processing user data obtained through a CGI form submission. Include details about parsing form data, validating inputs, and handling potential errors (4 marks).

(2 marks) Discuss two different methods for storing and retrieving processed data in a CGI application. Consider factors like efficiency, security, and scalability.

(2 marks) Briefly describe how basic data manipulation tasks like calculations, aggregations, or filtering can be achieved within a CGI script. Provide an example scenario where such manipulation might be beneficial.

2. Explain the implementation of data processing using HTML and python CGI scripting.

---

Figure 6-27. Question bank (2 of 2)

## Unit summary



IBM ICE (Innovation Centre for Education)

- Having completed this unit, you should be able to:
  - Develop graphical user interfaces (GUI) using various widgets and events
  - Understand the basics of Common Gateway Interface (CGI) and create interactive web applications

---

Figure 6-28. Unit summary

Unit summary is as stated above.

---

# Unit 7. Python Applications

## Overview

This unit will provide an overview of how to explore Python applications, including networking, serialization, and data processing with NumPy and Pandas, along with database applications.

## How you will check your progress

- Checkpoint

## References

IBM Knowledge Center

# Unit objectives



IBM ICE (Innovation Centre for Education)

- After completing this unit, you should be able to:
  - Able to Explore Python applications, including networking, serialization, and data processing with NumPy and Pandas, along with database applications
- Learning outcomes:
  - Able to Implement Python applications, including networking, serialization, and data processing with NumPy and Pandas, along with database applications

---

*Figure 7-1. Unit objectives*

The unit objectives and outcomes are as stated above

# Introduction



IBM ICE (Innovation Centre for Education)

- Python applications play a crucial role in seamlessly integrating communication protocols and data structuring processes.
- The language excels in establishing connections and utilizing various networking protocols like HTTP or TCP/IP for efficient information transmission between distributed systems.
- Serialization, a fundamental process, is employed to convert complex data structures into transmission friendly formats, ensuring compatibility across diverse platforms.

---

*Figure 7-2. Introduction*

Python applications execute these operations by encapsulating data in a serialized form, facilitating its transmission over networks. Python's capabilities extend to receiving and deserializing data, contributing to coherent information exchange in networked environments. In data processing and analysis, Python leverages various tools and libraries, including NumPy and Pandas, for manipulating and deriving insights from datasets. This multifaceted process involves tasks like cleaning, transforming, and organizing data, with Python's rich ecosystem offering powerful solutions. Libraries like Matplotlib and Seaborn enable informative data visualization, and the workflow includes importing, cleaning, exploring data, and applying statistical methods and machine learning algorithms. Python's versatility positions it as a preferred language for data scientists and analysts, providing a seamless workflow for diverse data processing and analysis tasks.

## Networking and serialization



IBM ICE (Innovation Centre for Education)

- Within the domain of networking and serialization, Python applications play a pivotal role in orchestrating a seamless integration of communication protocols and data structuring processes.
- Python excels in establishing connections and leveraging various networking protocols, such as HTTP or TCP/IP, to facilitate the efficient transmission of information between distributed systems.

---

*Figure 7-3. Networking and serialization*

Serialization, a fundamental process in Python, is employed to convert complex data structures into formats suitable for transmission, ensuring compatibility across diverse platforms. Python applications execute these operations by encapsulating data in a serialized form, allowing for its transmission over networks. Moreover, Python's capabilities extend to the reception and deserialization of data on the receiving end, contributing to the coherent exchange of information in networked environments. In essence, Python's versatility and robust libraries make it an ideal language for crafting applications that excel in networking and serialization, enabling effective communication and data exchange across various systems.

# Networking basics in python



IBM ICE (Innovation Centre for Education)

- Exploration of networking basics in Python involves understanding fundamental principles such as communication protocols and data transmission mechanisms.
- Connections are established and protocols are employed to facilitate the seamless exchange of information between interconnected systems.
- In computer networking, communication between devices like servers and clients is facilitated by sockets, which act as doorways for data exchange.

---

*Figure 7-4. Networking basics in python*

- Sockets can be configured for different protocols like TCP, ensuring reliable delivery through a pre-established connection, or UDP, which transmits data quickly without such guarantees.
- Python's networking libraries are utilized to implement networking functionalities, focusing on the creation and management of sockets for data transmission.

# Introduction to networking protocols



IBM ICE (Innovation Centre for Education)

- In computer networks, communication between devices is made possible through a set of established rules and guidelines, known as networking protocols.
- These protocols govern the formatting, transmission, and interpretation of data exchanged over networks, ensuring seamless connectivity and interoperability.

---

Figure 7-5. Introduction to networking protocols

Within this framework, key concepts are introduced:

- Packets: Data is divided into smaller, manageable units called packets, which are individually routed through the network.
- Addresses: Each device on a network is assigned a unique address, enabling precise identification and routing of packets.
- Layers: Protocols are often organized into layers, each responsible for specific tasks, creating a structured and modular approach.

## Common protocol types are differentiated based on their functions

### Link-layer protocols:

- Definition: Link-layer protocols manage communication between devices that are directly connected to the same network segment.
- Example: Common examples of link-layer protocols include Ethernet and Wi-Fi.
- Functionality: These protocols handle the transmission of data frames between devices within the same local network. They are responsible for addressing, error detection, and collision detection on the physical network medium.

### Network-layer protocols:

- Definition: Network-layer protocols facilitate routing of packets across multiple networks.
- Example: The Internet Protocol (IP) is a prominent example of a network-layer protocol.

- Functionality: These protocols are responsible for logical addressing, packet forwarding, and routing decisions. They enable communication between devices on different networks by determining the optimal path for packet transmission.

### **Transport-layer protocols:**

- Definition: Transport-layer protocols handle reliable end-to-end data delivery.
- Examples: The primary choices for transport-layer protocols are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP).
- Functionality: TCP ensures reliable data delivery by establishing a connection-oriented communication and implementing mechanisms for error recovery and flow control. UDP, on the other hand, provides a connectionless communication with minimal overhead, suitable for applications where speed is prioritized over reliability.

### **Application-layer protocols:**

- Definition: Application-layer protocols support specific network applications.
- Examples: Examples of application-layer protocols include HTTP (Hypertext Transfer Protocol) for web browsing, FTP (File Transfer Protocol) for file transfers, and SMTP (Simple Mail Transfer Protocol) for email.
- Functionality: These protocols define the rules and conventions for communication between applications running on different devices. They enable users to interact with network services and access resources over the internet. Each application-layer protocol serves a specific purpose and operates at the highest level of the networking stack.

### **TCP/IP Stack:**

- The TCP/IP stack, short for Transmission Control Protocol/Internet Protocol, is a comprehensive set of networking protocols used for communication over the internet.
- It is composed of multiple layers, each serving specific functions in the process of transmitting data between devices.
- The TCP/IP stack is structured in a hierarchical manner, with higher layers building upon the services provided by lower layers.

### **Layers of the TCP/IP Stack**

- **Application layer:**
  - This layer contains application-specific protocols responsible for user interactions and data exchange.
  - Examples include HTTP (Hypertext Transfer Protocol), FTP (File Transfer Protocol), SMTP (Simple Mail Transfer Protocol), and DNS (Domain Name System).
- **Transport layer:**
  - The transport layer ensures end-to-end communication between hosts and provides reliable data delivery.
  - The primary protocols at this layer are TCP (Transmission Control Protocol) and UDP (User Datagram Protocol).
- **Internet layer:**
  - The internet layer, also known as the network layer, handles the routing and forwarding of data packets between networks.
  - The Internet Protocol (IP) is the key protocol at this layer, responsible for logical addressing and packet routing.
- **Link layer:**
  - The link layer, also referred to as the network access layer or data link layer, deals with the transmission of data frames over physical networks.

- Common protocols at this layer include Ethernet, Wi-Fi (IEEE 802.11), and PPP (Point-to-Point Protocol).

#### **Additional protocols in the TCP/IP stack:**

- Alongside the well-known protocols mentioned earlier, the TCP/IP stack encompasses a variety of other protocols for specific purposes:
  - FTP (File Transfer Protocol) for transferring files between hosts.
  - ICMP (Internet Control Message Protocol) for network diagnostics and error reporting.
  - ARP (Address Resolution Protocol) for mapping IP addresses to MAC addresses.
  - DHCP (Dynamic Host Configuration Protocol) for dynamically assigning IP addresses to devices on a network.

#### **Functionality:**

- The TCP/IP stack enables devices to communicate seamlessly over the internet by providing a standardized framework for data transmission.
- Each layer of the stack performs specific functions, from addressing and routing to error detection and correction, ensuring reliable and efficient communication between networked devices.

#### **Interoperability:**

- One of the key strengths of the TCP/IP stack is its interoperability across different types of networks and hardware platforms.
- This interoperability allows devices running on different operating systems and hardware architectures to communicate effectively, contributing to the global connectivity of the internet. The intricate interplay of these protocols enables the vast and interconnected world of digital communication we experience today.

# Creating client and server sockets



IBM ICE (Innovation Centre for Education)

- In socket programming, building networked applications revolves around crafting client and server sockets.
- For the server, this entails binding its socket to a specific address and port, essentially raising a flag announcing its readiness to receive connections.
- Clients seeking interaction connect to this flagged address, establishing a communication channel.
- Data then flows through these connected sockets, with servers responding to client requests or clients receiving server-initiated information.
- This intricate dance of socket creation, binding, connection establishment, and data exchange forms the very foundation of networked applications built with socket programming.

*Figure 7-6. Creating client and server sockets*

The socket module in Python provides interfaces for developing UDP and TCP clients and servers. The socket module in Python enables developers to create networked applications by providing the necessary tools to establish communication channels between clients and servers. This allows for the seamless exchange of data over networks.

Here's a breakdown of the key concepts:

- **Server:** Represents an application that is waiting for connections from clients. It typically listens for incoming connections on a specific port and IP address.
- **Client:** Represents an application that initiates connections to servers. It connects to the server's IP address and port to communicate with it.
- **Sockets:** Sockets serve as the communication endpoints through which clients and servers exchange data. They provide a bidirectional communication channel between two processes running on different devices. When a client application connects to a server, a socket is established for communication.

For example, when a web browser is opened, it automatically creates a socket to connect to the web server. This socket facilitates the exchange of HTTP requests and responses between the client (browser) and the server.

The steps to create the client and server sockets in Python are mentioned below:

- The Python socket server program executes first and waits for any incoming requests.
- The Python socket client program initiates the conversation by establishing a connection with the server.
- Upon receiving a request from the client, the server program responds accordingly.

- If the client program sends a "bye" message, it terminates. Optionally, the server program may also terminate when the client program terminates.

Alternatively, the server program can continue running indefinitely or terminate based on a specific command received in the client's request.

### **Example**

- **Constructing sockets:**

- Import the essential module: `import socket`
- Craft a socket object: `socket_object = socket.socket(socket.AF_INET, socket.SOCK_STREAM)`
  - `socket.AF_INET` specifies IPv4 for network communication.
  - `socket.SOCK_STREAM` indicates TCP for reliable connection-oriented communication.

- **Server-side socket configuration:**

- Bind the socket: `socket_object.bind((host_address, port_number))`
  - Assign the socket to a specific host address and port for listening.
- Initiate listening: `socket_object.listen()`
  - Signals readiness to accept incoming connections.
- Accept connections: `connection_object, client_address = socket_object.accept()`
  - Establishes a connection with a client, returning a new socket object for communication.

- **Client-side socket configuration:**

- Initiate connection: `socket_object.connect((server_address, server_port))`
  - Connects the client socket to the specified server address and port.

### **Code example (Server)**

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(("localhost", 8080)) # Bind to localhost on port 8080
server_socket.listen()
client_socket, client_address = server_socket.accept()
message = client_socket.recv(1024).decode()
print("Received message from client:", message)
client_socket.send("Hello from server!".encode())
client_socket.close()
server_socket.close()
```

### **Code example (Client)**

```
import socket

client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect(("localhost", 8080)) # Connect to server on port 8080
message = "Hello from client!"
client_socket.send(message.encode())
response = client_socket.recv(1024).decode()
print("Received message from server:", response)
client_socket.close()
```

**Code explanation**

- Socket objects are created to initiate network communication.
- Binding and listening are performed on the server-side to await connections.
- Clients initiate connections to servers using their addresses and ports.
- Data exchange is achieved through sending and receiving methods.
- Sockets are gracefully closed to terminate connections.

# Socket programming



IBM ICE (Innovation Centre for Education)

- Socket programming is a pivotal skill for building networked applications.
- It serves as the backbone for establishing communication channels between interconnected systems.
- This technique enables developers to create robust and responsive applications capable of seamless data transfer across networks.
- Socket programming involves the creation, configuration, and management of socket communication endpoints that facilitate the flow of data between different entities.

---

Figure 7-7. Socket programming

- It is particularly significant in the context of data transfer and communication.
- Understanding the intricacies of socket creation and utilization is crucial for crafting applications that can efficiently exchange information in real-time.
- Socket programming empowers developers to initiate connections and securely transmit data.
- It facilitates the architecture of networked applications that exhibit reliability, scalability, and responsiveness.



IBM ICE (Innovation Centre for Education)

# Building networked applications

- Robust and efficient networked applications can be built using socket programming through a solid understanding of the concepts and the implementation of best practices.
- It is important to remember that the specific techniques employed will be determined by the chosen application architecture, communication protocols, and data types involved.
- Building networked applications with sockets involves several key steps:

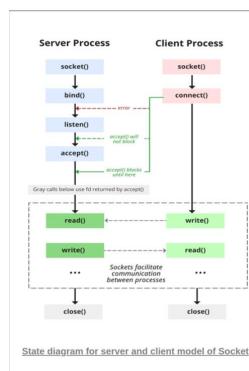


Figure 1: State diagram for server and client model of Socket

Source: [www.geeksforgeeks.org](http://www.geeksforgeeks.org)

Figure 7-8. Building networked applications

- **Defining application architecture:**

- Client-Server Architecture: This common model involves a server process listening for incoming client connections and providing services. Clients initiate connections and request data or services from the server. The ClientServer Architecture in socket programming is presented in Figure 1.
- Peer-to-Peer Architecture: Devices directly communicate with each other without a central server, often used for file sharing or distributed computing

- **Choosing communication protocols:**

Protocol	Description
TCP (Transmission Control Protocol)	Reliable, connection-oriented protocol ensuring ordered data delivery with error checking and retransmission. Ideal for critical applications like file transfers and financial transactions.
UDP (User Datagram Protocol)	Connectionless, faster protocol sacrificing reliability for speed. Suitable for real-time applications like video streaming and online gaming where slight data loss is tolerable.

Table 1: Choosing communication protocols

- **Implementing socket communication:**

- Socket creation: Utilize socket APIs to create sockets for sending and receiving data.
- Binding and listening (server-side): Bind the server socket to a specific address and port to listen for incoming connections.
- Connecting (client-side): Establish a connection with the server by specifying its address and port.
- Data exchange: Send and receive data using socket methods like send and receive.
- Closing connections: Gracefully close sockets to terminate communication.
- **Error handling and robustness:**
  - Implement robust error-handling mechanisms to handle network failures, invalid data, and potential security threats.
  - Utilize timeouts and retries to cope with temporary network issues.
- **Security considerations:**
  - Secure protocols like HTTPS and TLS should be used for sensitive data transmission.
  - Authentication and authorization mechanisms should be implemented to control access to resources.

### An example code

**Here's a Python code example demonstrating building a basic networked application using socket programming.**

#### **Client-server architecture-**

##### **Server-Side Code:**

```
import socket

# Create a TCP/IP socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind the socket to a specific address and port
server_address = ('localhost', 8080)
server_socket.bind(server_address)

# Listen for incoming connections
server_socket.listen(1)
print("Server listening on port 8080...")

while True:
    # Accept a connection
    client_socket, client_address = server_socket.accept()
    # Handle client requests in a separate thread
    # (this allows the server to handle multiple clients concurrently)
    thread = threading.Thread(target=handle_client, args=(client_socket,))
    thread.start()

    def handle_client(client_socket):
        # Receive data from the client
        data = client_socket.recv(1024).decode()
        print("Received message from client:", data)

    # Process the client's request (e.g., perform calculations, access a database)
    # ... # Send a response back to the client
    response = "Response from server!".encode()
    client_socket.sendall(response)
```

**# Close the connection**

```
client_socket.close()
```

**Client-Side Code:**

```
import socket
```

**# Create a TCP/IP socket**

```
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

**# Connect to the server**  
`server_address = ('localhost', 8080) # Replace 'localhost' with server's IP if needed`  
`client_socket.connect(server_address)`

**# Send a message to the server**

```
message = "Hello from client!".encode()
```

```
client_socket.sendall(message)
```

**# Receive the server's response**

```
response = client_socket.recv(1024).decode()
```

```
print("Received message from server:", response)
```

**# Close the connection**

```
client_socket.close()
```

**Key Points:**

- The server listens for incoming connections and spawns threads to handle clients concurrently.
- The client initiates a connection and sends requests to the server.
- The server processes requests and sends responses back to the client.
- Both sides gracefully close their sockets to terminate communication.

**Additional considerations:**

- Error handling: Implement robust error handling to manage network issues and exceptions.
- Security: Use secure protocols (HTTPS, TLS) and authentication mechanisms for sensitive data.
- Data serialization: Convert complex data structures into byte streams for transmission.
- Multithreading or Asynchronous Programming: Efficiently handle multiple connections.
- Encryption and decryption: Protect sensitive data during transmission.

# Data transfer and communication



IBM ICE (Innovation Centre for Education)

- In the context of socket programming in Python, data transfer and communication involve the establishment of communication channels between different entities, typically a client and a server.
- Sockets serve as endpoints for communication, allowing data to be transmitted bidirectionally.
- Several techniques can be employed for efficient data transfer and communication, which are crucial for networked applications.

---

Figure 7-9. Data transfer and communication

- **Serialization and deserialization:**
  - Convert complex data structures into byte streams for transmission over the network (serialization).
  - Reconstruct the original data structures from received byte streams at the receiving end (deserialization).
- **Buffering:**
  - Utilize buffers to temporarily store data before sending or receiving to optimize network I/O performance.
- **Compression:**
  - Compress data before transmission to reduce bandwidth usage and transmission time, especially for large data transfers.
- **Multithreading and asynchronous programming:**
  - Utilize multithreading or asynchronous programming paradigms to handle multiple connections and data transfers concurrently, improving application responsiveness and efficiency.
- **Encryption and decryption:**
  - Encrypt sensitive data before transmission and decrypt it at the receiving end to ensure confidentiality and data integrity.

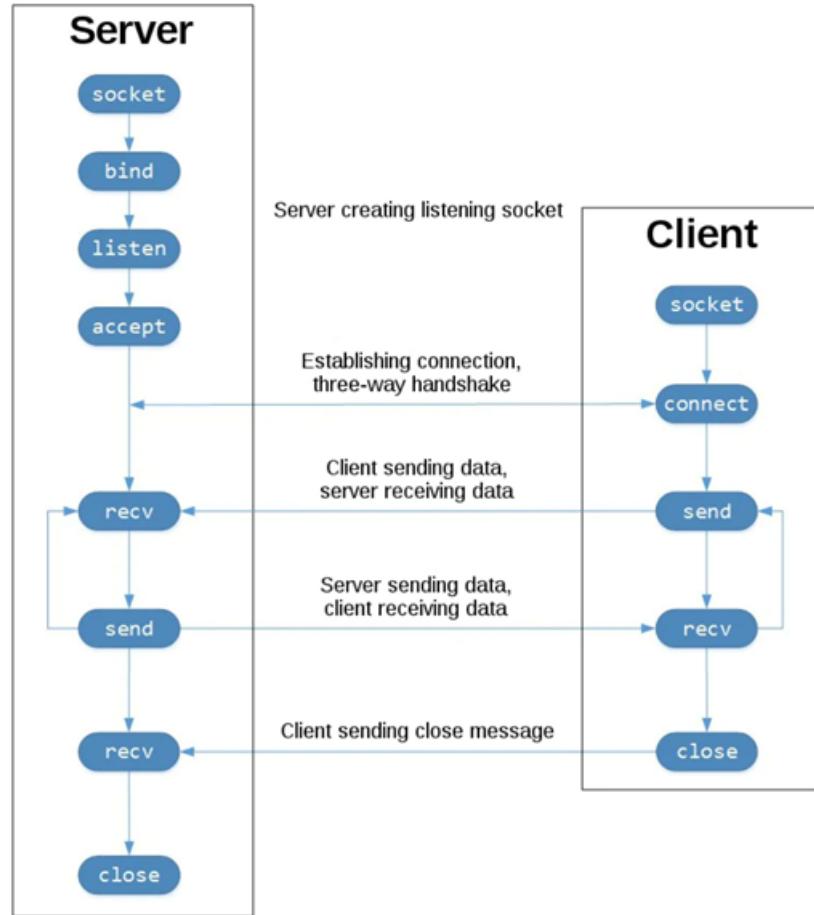


Figure 2: TCP Socket flow [<https://realpython.com/python-sockets/>]



IBM ICE (Innovation Centre for Education)

## An example code

- The server listens for incoming connections at a specified address and port.
- Each client connection is handled in a separate thread to allow concurrent processing.
- Upon receiving data from the client, the server deserializes it using Pickle.
- The server verifies the integrity of the received data using a SHA-256 checksum.
- If the data integrity check passes, the server sends a compressed response back to the client.
- On the client side, a message is sent to the server along with a computed checksum for data integrity.
- The client receives the server's compressed response, decompresses it, and prints the received message.
- Both client and server connections are appropriately closed after the communication is complete.
- The code incorporates threading for concurrent client handling, error-handling mechanisms, and basic security measures to ensure the integrity of the transmitted data.

*Figure 7-10. An example code*

### Additional notes:

- The use of try, except, and finally blocks provides error handling, ensuring that the code gracefully handles exceptions.
- The Hashlib module is used for computing SHA-256 checksums.
- The zlib module is employed for data compression.
- Threading is used to handle each client connection in a separate thread (handle\_client function).
- The server listens for incoming connections in an infinite loop (start\_server function).
- This code represents a simple client-server interaction with basic error handling, integrity checks, and compression for data security.

### Server-side code

```
import socket
import pickle
import zlib
import threading
import hashlib

def handle_client(client_socket):
    try:
```

```

# Step 1: Receive data from the client
data = client_socket.recv(1024)

# Step 2: Deserialize data
deserialized_data = pickle.loads(data)

# Step 3: Verify data integrity using checksum
    received_checksum = deserialized_data.pop('checksum', None)
computed_checksum = hashlib.sha256(pickle.dumps(deserialized_data)).hexdigest()
if received_checksum == computed_checksum:
    print("Received message from client:", deserialized_data)

# Step 4: Compress and send a response back to the client
    response = {'content': "Hello from server!"}
    compressed_response = zlib.compress(pickle.dumps(response))
    client_socket.sendall(compressed_response)
else:
    print("Data integrity check failed. Ignoring the message.")

except Exception as e:
    print("Error handling client request:", str(e))

finally:# Step 5: Close the connection
    client_socket.close()

def start_server():
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_address = ('localhost', 8080)
    server_socket.bind(server_address)
    server_socket.listen(5)
    print("Server listening on", server_address)
    while True:
        # Step 6: Accept a connection and handle it using a separate thread
        client_socket, client_address = server_socket.accept()
        client_thread = threading.Thread(target=handle_client,
args=(client_socket,))
        client_thread.start()

if __name__ == "__main__":
    start_server()

Client-side code:
import socket
import pickle
import zlib
import hashlib

def start_client():
    try:

```

**# Step 1: Create a TCP/IP socket**

```
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

**# Step 2: Connect to the server**

```
server_address = ('localhost', 8080)
```

```
client_socket.connect(server_address)
```

**# Step 3: Send a message to the server with checksum**

```
message = {'content': "Hello from client!"}
```

```
checksum = hashlib.sha256(pickle.dumps(message)).hexdigest()
```

```
message['checksum'] = checksum
```

```
compressed_message = zlib.compress(pickle.dumps(message))
```

```
client_socket.sendall(compressed_message)
```

**# Step 4: Receive the server's response**

```
compressed_response = client_socket.recv(1024)
```

```
response = pickle.loads(zlib.decompress(compressed_response))
```

```
print("Received message from server:", response)
```

```
except Exception as e:
```

```
    print("Error in client:", str(e))
```

```
finally:
```

**# Step 5: Close the connection**

```
client_socket.close()
```

```
if __name__ == "__main__":
```

```
    start_client()
```

**Explanation****Server-side code:**

Step 1: The server receives data from the client using `recv`.

Step 2: The received data is deserialized using `Pickle.loads`.

Step 3: Data integrity is verified by comparing the received checksum with the computed checksum using SHA-256.

Step 4: If data integrity is confirmed, the server sends a compressed response back to the client.

Step 5: The connection is closed in the `finally` block.

**Client-side code:**

Step 1: The client creates a TCP/IP socket.

Step 2: It connects to the server using `connect`.

Step 3: The client sends a message to the server with a computed checksum for data integrity.

Step 4: It receives the server's compressed response, decompresses it, and deserializes it.

Step 5: The connection is closed in the `finally` block.

# Serialization (JSON and Pickle)



IBM ICE (Innovation Centre for Education)

- Serialization is the process of converting a Python object into a byte stream, allowing it to be stored in a file or transmitted over a network.
- This enables the object's state to be preserved for later use or communication with other systems.

---

Figure 7-11. *Serialization (JSON and Pickle)*

## Example

```
import pickle

# Define a Python object
data = {'name': 'John', 'age': 30, 'city': 'New York'}

# Serialize the object using pickle.dumps()
serialized_data = pickle.dumps(data)

# Print the serialized data
print("Serialized Data:", serialized_data)
```

In this example, a dictionary data representing a person's information is serialized using the `pickle.dumps()` method. The serialized data is then printed, showing the byte stream representation of the Python object. This serialized data can be saved to a file, transmitted over a network, or stored in a database. Later, it can be deserialized back into a Python object using the `pickle.loads()` method.

Serialization is the process of converting complex data structures into a format that can be easily stored, transmitted, or reconstructed. Two commonly used serialization methods in Python are JSON (JavaScript Object Notation) and Pickle. A comparative analysis of JSON and Pickle is presented in Table 1.

Python Pickle	JSON
<p>Python Pickle is the process of converting python objects (list, dict, tuples, etc.) into byte streams which can be saved to disks or can be transferred over the network. The byte streams saved on the file contains the necessary information to reconstruct the original python object. The process of converting byte streams back to python objects is called deserialization.</p>	<p>JSON stands for JavaScript Object Notation. Although derived from JavaScript, JSON is a language interoperable. JSON is another way of storing python objects into a disk so that later on, they can be loaded without having the need to recreate the data again.</p>
<p>Pickle lets the user store data in binary format.</p>	<p>JSON lets the user store data in a human-readable text format.</p>
<p>Not only the data fields but with the pickle module, even classes and methods can be serialized and deserialized.</p>	<p>JSON is limited to certain python objects, and it cannot serialize every python object.</p>
<p>Pickle supports almost all the data types supported by Python:</p> <p>Following data types can be pickled:</p> <ul style="list-style-type: none"> <li>• None, Boolean values</li> <li>• Integer, float and complex numbers</li> <li>• Sequence data types – lists, tuples, strings</li> <li>• Collections – dictionaries, bytes, bytearray</li> <li>• Even functions and classes can be serialized</li> </ul>	<p>JSON supports only a subset of python data types.</p> <p>Following data types are supported by JSON:</p> <ul style="list-style-type: none"> <li>• Strings</li> <li>• Numbers</li> <li>• JSON object</li> <li>• An array</li> <li>• Boolean value</li> <li>• Null</li> </ul> <p>Below data types are not supported by JSON:</p> <ul style="list-style-type: none"> <li>• Classes</li> <li>• Functions</li> </ul>
<p>Pickling is specific to Python only, and it does not guarantee cross-language compatibility. Even different python versions are not compatible with each other. It means pickling done in python version 2.x may not work in python version 3.x</p>	<p>Almost all programming languages support JSON. JSON is language independent. So, even a non-Python programmer can use this for data interchange.</p>
<p>Serialization and deserialization with Pickle is a slower process when compared to other available alternatives.</p>	<p>JSON is a lightweight format and is much faster than Pickling.</p>
<p>There is always a security risk with Pickle. Unpickling data from unknown sources should be avoided as it may contain malicious or erroneous data.</p>	<p>There are no loopholes in security using JSON, and it is free from security threats.</p>

Table 2: JSON vs Pickle

# Serialization overview



IBM ICE (Innovation Centre for Education)

- Serialization is often likened to the careful packing of a suitcase for a journey, ensuring that information is organized and portable for travel through various digital landscapes.
- Serialization is recognized as a cornerstone of modern programming, enabling data persistence, communication, and interoperability.
- By comprehending its concepts and methods, effective management and exchange of data across diverse applications and systems can be achieved.

---

Figure 7-12. Serialization overview

## Why It's considered crucial:

- Data Storage: Serialization is employed for saving data to files or databases, preserving its structure, and enabling its retrieval at a later time.
- Network Communication: It's deemed essential for sending data over networks, enabling meaningful exchange of information between different devices and services.
- Cross-Language Compatibility: Certain serialization formats, such as JSON, facilitate data sharing between different programming languages, fostering collaboration and interoperability.

## Common methods employed:

- JSON: A human-readable, text-based format, widely utilized in web services and APIs.
- Pickle: A Python-specific, binary format, known for its efficiency in handling diverse data types.
- XML: Another text-based format, often employed in web development and configuration files.
- MessagePack: An efficient binary format, offering speed and compactness.
- Protocol buffers: A language-neutral format developed by Google, designed for high-performance data exchange.

## Key considerations noted:

- Data type compatibility: The selection of a serialization format that supports the specific data types being worked with is crucial.

- Human readability: JSON is acknowledged for its excellent human readability, while binary formats like Pickle are less readable but often more efficient.
- Cross-language compatibility: For scenarios involving data exchange with other languages, JSON or XML are recommended.
- Security: Mindfulness of security risks is paramount, especially when handling untrusted data with formats like Pickle.

# JSON and Pickle for data serialization (1 of 2)



IBM ICE (Innovation Centre for Education)

- JSON (JavaScript Object Notation):
  - JSON, or JavaScript Object Notation, is a lightweight and human-readable data interchange format.
  - It represents data using key-value pairs and supports diverse data types such as objects, arrays, numbers, strings, and boolean values.
  - JSON finds extensive use in web communication and configuration files.
  - Python utilizes the json module, which provides functions like dump and load for encoding and decoding JSON data.
  - JSON's simplicity, compatibility across multiple programming languages, and readability make it a preferred choice for data interchange.
  - It simplifies data serialization and transmission between servers and web applications.
- In Python, the json module allows encoding of Python objects into JSON format (json.dumps()) and decoding of JSON data into Python objects (json.loads()).

*Figure 7-13. JSON and Pickle for data serialization (1 of 2)*

## How It works

Serialization (to JSON):

Import the json module in Python.

Use json.dumps(data) to convert a Python object into a JSON string.

The "dump" and "dumps" methods are both part of the JSON (JavaScript Object Notation) module in Python and are used for JSON serialization. However, they have different purposes and usage

dump():

Purpose: Used to serialize Python objects to a JSON formatted file.

Syntax: json.dump(obj, fp)

Parameters:

obj: The Python object to be serialized.

fp: A file-like object where the JSON data will be written.

Example:

```
import json
data = {'name': 'John', 'age': 30, 'city': 'New York'}
```

```
with open('data.json', 'w') as f:
    json.dump(data, f)
```

In this example, the `dump()` method is used to serialize the Python dictionary data into JSON format and write it to the file 'data.json'.

`dumps() :`

**Purpose:** Used to serialize Python objects to a JSON formatted string.

**Syntax:** `json.dumps(obj)`

**Parameters:**

`obj:` The Python object to be serialized.

**Example:**

```
import json

data = {'name': 'John', 'age': 30, 'city': 'New York'}

json_string = json.dumps(data)

print(json_string)
```

In this example, the `dumps()` method is used to serialize the Python dictionary data into a JSON formatted string `json_string`.

Store or transmit the JSON string.

**Deserialization (from JSON):**

Retrieve the JSON string.

Use `json.loads(json_string)` to parse it back into a Python object.

The `load()` method is used in Python's JSON (JavaScript Object Notation) module to deserialize JSON data from a file-like object (such as a file or a network socket) into a Python object. Here's an explanation of the `load()` method:

`load():` This method reads JSON data from a file-like object and deserializes it into a Python object.

**Syntax:** `json.load(fp)`

**Parameters:**

`fp:` A file-like object (e.g., file, network socket) containing JSON data.

**Example:**

```
import json

# Open a JSON file for reading
with open('data.json', 'r') as f:
    # Load JSON data from the file
    data = json.load(f)

    # Print the deserialized Python object
    print(data)
```

In this example, the `load()` method is used to read JSON data from the file 'data.json' and deserialize it into a Python object `data`. The resulting Python object can then be accessed and manipulated like any other Python data structure.

The `load()` method is useful when you have JSON data stored in a file and you want to read it into your Python program as a Python object for further processing. It simplifies the process of reading and deserializing JSON data, making it easier to work with JSON-formatted data in Python applications.

- **Key considerations:**

- **Security:** Validate and sanitize JSON data, especially from untrusted sources, to prevent security vulnerabilities and injection attacks.
  - **Interoperability:** Leverage JSON's language-agnostic nature for seamless data exchange across various platforms and programming languages.
  - **Schema validation:** Enhance data integrity by utilizing JSON Schema to define and validate the expected structure of your JSON data.
  - **Efficiency and size:** Evaluate data size and transmission efficiency, considering compression techniques for network transmission, especially in scenarios with limited bandwidth.
- **Common use cases:**
    - **Web APIs (Application Programming Interfaces):** JSON is commonly used as the data interchange format for web APIs. Its lightweight and human-readable structure makes it ideal for transmitting data between web servers and clients in a format easily processed by JavaScript.
    - **Configuration Files:** JSON is frequently employed for storing configuration settings due to its simplicity and ease of readability. Many applications use JSON files to manage and store configuration parameters, allowing for straightforward customization.
    - **Data storage in NoSQL databases:** Many NoSQL databases, such as MongoDB, utilize JSON-like formats for storing and retrieving data. JSON's flexible structure aligns well with the dynamic nature of NoSQL databases, making it a prevalent choice for representing documents and records.
    - **Serialization in web development:** JSON is extensively used for serializing and deserializing data in web development. Whether storing data in cookies, local storage, or transmitting data between the client and server, JSON provides a versatile and widely supported format for representing structured information.
  - **When to choose JSON:**
    - **Interoperability across languages:** JSON is preferable to choose when a widely supported and language-agnostic data interchange format is required. JSON's simplicity and broad adoption make it an excellent choice for exchanging data between different programming languages and platforms.
    - **Human-readability and debugging:** Opt for JSON when human-readable data representation is crucial for debugging and troubleshooting. Its clear and concise format makes it easy for developers to understand and work with, enhancing the readability of the exchanged data.
    - **Web-based applications:** JSON is a natural choice for web-based applications due to its compatibility with JavaScript. Many web APIs and data exchanged between web servers and clients are formatted in JSON, making it well-suited for front-end and back-end communication in web development.
    - **Schema flexibility:** Choose JSON when dealing with data structures that may evolve over time. JSON's flexible schema allows for the representation of varying data structures without strict constraints, making it adaptable to changing requirements in dynamic environments.

# JSON and Pickle for data serialization (2 of 2)



IBM ICE (Innovation Centre for Education)

- Pickle:
  - Pickle, a Python-specific serialization format, offers greater versatility than JSON.
  - It can serialize a wider array of Python objects, including custom classes and functions.
  - Pickle plays a crucial role in saving and loading Python-specific data structures.
  - It is particularly valuable in Python-centric applications due to its compatibility with complex Python objects.
  - The pickle module introduces methods such as dump and load for serialization and deserialization.
  - Despite its versatility, Pickle sacrifices human-readability compared to JSON.
  - This becomes a consideration when transparency and manual inspection of serialized data are important.
  - In Python, the pickle module facilitates both serialization (Pickle.dumps()) and deserialization (Pickle.loads()) of Python objects.
  - Pickle proves advantageous when working within a Python environment and dealing with intricate Python objects.

*Figure 7-14. JSON and Pickle for data serialization (2 of 2)*

### Serialization (Pickling):

- Import the pickle module.
- Use Pickle.dumps(object) to serialize an object into a byte string.
- Store or transmit the pickled data.

### Deserialization (Unpickling):

- Retrieve the pickled data.
- Use Pickle.loads(pickled\_data) to deserialize it back into a Python object.

### Example codes:

#### Basic data Types:

```
import pickle

data = {"name": "Alice", "age": 30, "hobbies": ["coding", "reading"]}
# Serialization
pickled_data = Pickle.dumps(data)
# Deserialization
data_back = Pickle.loads(pickled_data)
print(data_back)
```

```
# Output: {'name': 'Alice', 'age': 30, 'hobbies': ['coding', 'reading']}  
Custom Classes and Functions:  
class Person:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
def greet(person):  
    print("Hello, " + person.name + "!")  
person = Person("Bob", 25)  
# Serialize object and function  
pickled_data = pickle.dumps((person, greet))  
# Deserialize back  
person_back, greet_back = pickle.loads(pickled_data)  
greet_back(person_back)  
# Output: Hello, Bob!
```

### Key considerations:

- Security: Be cautious when unpickling data from untrusted sources, as it can execute arbitrary code.
- Compatibility: Pickled data is specific to Python versions and environments, so ensure compatibility.
- Alternatives: Consider JSON for cross-language compatibility and human readability, or other formats like MessagePack or Protocol Buffers for specific needs.
- The pickle protocol is specific to Python, which means it may not be compatible with other programming languages, limiting its interoperability.
- Compatibility between different versions of Python is not guaranteed due to variations in supported data structures for serialization by the module.
- By default, the latest version of the pickle protocol is used, unless specified otherwise manually.
- It's important to note that the pickle module is not immune to security vulnerabilities arising from erroneous or maliciously constructed data, as documented.

### Common use cases:

- **Data serialization:** One of the primary use cases for Pickle is serializing Python objects, converting them into a byte stream. This is particularly valuable when you need to store complex data structures or custom objects persistently, making it easy to save and later retrieve the state of your program.
- **Model persistence in machine learning:** Pickle is widely used in machine learning for serializing trained models. After a machine learning model has been trained on a dataset, it can be pickled, allowing developers to save the model's state. Later, this pickled model can be loaded for making predictions on new data without the need to retrain the entire model.
- **Caching computations:** Pickle is employed for caching computationally expensive or time-consuming results. By pickling the output of certain operations, subsequent runs of the program can retrieve and reuse these precomputed results, saving valuable processing time and resources.
- **Configuration management:** Pickle facilitates the serialization of configuration settings or objects that define the configuration of a system. This is useful for saving and loading configuration data, making it easy to store and retrieve the state of an application's settings.

### When to Choose Pickle:

- Complex python objects:** Pickle is particularly useful when dealing with complex Python objects, custom classes, or nested data structures. It allows for seamless serialization and deserialization of these objects without the need for extensive manual conversion.
- Intra-python communication:** When working exclusively within a Python environment and needing to exchange data between different parts of a Python application, Pickle provides a straightforward solution for serialization and deserialization, facilitating efficient communication.
- Rapid development and prototyping:** During the prototyping phase or in scenarios where quick development iterations are crucial, Pickle enables developers to save and load Python objects swiftly. It is a convenient choice for situations where the focus is on speed and simplicity.
- State persistence and recovery:** Pickle is a suitable choice when there is a need to save and recover the state of a program or application. This is beneficial for scenarios such as saving game progress or preserving the state of long-running computations, allowing for easy recovery when needed.

	JSON	Pickle
<b>Advantages</b>		
Versatility	Supports various data types and structures	Can serialize almost any Python object
Human-readability	Easy to read and understand	Not human-readable, but compact binary format
Interoperability	Widely supported across programming languages	Python-specific, limited cross-language compatibility
Security	Safer for handling untrusted data	Less secure, susceptible to security vulnerabilities
<b>Disadvantages</b>		
Data Types	Limited support for complex Python objects	May not support all Python objects, versions, or cross-language compatibility
Size	Larger file size compared to Pickle	Compact binary format results in smaller file size
Performance	Slower serialization and deserialization	Faster serialization and deserialization

Table 3: JSON vs pickle

## Data processing and analysis



IBM ICE (Innovation Centre for Education)

- In Python, data processing and analysis involve leveraging various tools and libraries to manipulate and derive insights from datasets.
- This multifaceted process encompasses tasks such as cleaning, transforming, and organizing data for meaningful interpretation.

---

Figure 7-15. Data processing and analysis

Python, with its rich ecosystem of libraries, offers powerful solutions for data processing and analysis. Libraries such as NumPy and Pandas provide essential tools for working with numerical data and structured datasets, respectively. Additionally, visualization libraries like Matplotlib and Seaborn enable the creation of informative plots and charts. The process involves importing, cleaning, and exploring the data, followed by applying statistical methods and machine learning algorithms for in-depth analysis. Python's versatility makes it a preferred choice for data scientists and analysts, offering a seamless workflow for data processing and analysis tasks.

Imagine a retail company operating both online and offline stores. They collect vast amounts of data on customer transactions, including purchase history, product preferences, and demographic information. To optimize their business operations and enhance customer experience, they employ data processing and analysis techniques.

- **Data collection:** The company gathers data from various sources, such as point-of-sale systems, online transactions, customer surveys, and social media interactions.
- **Data cleaning and integration:** The collected data may contain errors, inconsistencies, or missing values. Data cleaning involves identifying and rectifying these issues to ensure accuracy. Additionally, data from different sources need to be integrated to create a comprehensive dataset.
- **Exploratory Data Analysis (EDA):** Analysts explore the dataset to gain insights into customer behavior, purchasing patterns, and product preferences. They use statistical methods, visualizations, and descriptive analytics to uncover trends and correlations within the data.

- **Customer segmentation:** Using clustering algorithms, the company segments customers based on their purchasing behavior, demographics, or geographic location. This segmentation helps in targeted marketing campaigns and personalized recommendations.
- **Sales forecasting:** By analyzing historical sales data and external factors like seasonality and market trends, the company can forecast future sales volumes. Accurate forecasting enables effective inventory management and resource allocation.
- **Sentiment analysis:** The company performs sentiment analysis on customer feedback and social media mentions to gauge customer satisfaction and identify areas for improvement. Positive sentiment can be leveraged for marketing and brand building, while negative sentiment prompts corrective actions.
- **Recommendation systems:** Utilizing collaborative filtering or machine learning algorithms, the company develops recommendation systems to suggest products to customers based on their past purchases and browsing behavior. This enhances the shopping experience and drives sales.
- **Marketing campaign optimization:** Data analysis helps the company evaluate the effectiveness of marketing campaigns and promotions. By analyzing campaign performance metrics and customer responses, they can refine their marketing strategies for better results. Overall, data processing and analysis empower the retail company to make data-driven decisions, optimize operations, and deliver a personalized shopping experience to customers, ultimately driving business growth and profitability.



IBM ICE (Innovation Centre for Education)

## Introduction to NumPy and pandas

- NumPy and Pandas are foundational libraries in the Python ecosystem, essential for data manipulation, analysis, and scientific computing.
- NumPy, short for Numerical Python, focuses on numerical operations and provides a powerful array object along with a collection of mathematical functions.
- It enables efficient manipulation of large datasets through its multidimensional array structures.

---

*Figure 7-16. Introduction to NumPy and pandas*

Pandas, on the other hand, excels in handling structured data through its key data structures: Series and DataFrame. A Series is a one-dimensional array-like object, while a DataFrame is a two-dimensional table with labeled axes, similar to a spreadsheet. Pandas simplifies data manipulation, cleaning, and analysis tasks, making it a go-to tool for data scientists and analysts. NumPy and Pandas work seamlessly together, with Pandas often relying on NumPy arrays for data representation. This combination provides a comprehensive toolkit for data exploration, cleaning, and preprocessing before feeding data into machine learning models or statistical analyses.



IBM ICE (Innovation Centre for Education)

## What is NumPy?

- NumPy, or Numerical Python, is a fundamental library in the Python ecosystem that specializes in numerical computing.
- It provides support for large, multidimensional arrays and matrices, along with an extensive collection of high-level mathematical functions to operate on these arrays.

---

Figure 7-17. What is NumPy?

### NumPy arrays are preferred over Python lists for several reasons:

- **Performance:** NumPy arrays are implemented in C and optimized for performance, whereas Python lists are implemented in Python itself. This makes NumPy arrays significantly faster for numerical computations, especially when dealing with large datasets or performing complex mathematical operations.
- **Memory efficiency:** NumPy arrays are more memory efficient compared to Python lists. This is because NumPy arrays store data in contiguous memory blocks, whereas Python lists store references to objects scattered throughout memory. As a result, NumPy arrays consume less memory and offer better performance for memory-intensive operations.
- **Vectorized operations:** NumPy arrays support vectorized operations, where operations are applied element-wise to entire arrays without the need for explicit looping. This enables concise and efficient code, as complex computations can be expressed in a few lines of code using array operations, rather than writing explicit loops.
- **Broad range of functions:** NumPy provides a wide range of mathematical functions and operations specifically designed for numerical computing, including linear algebra, Fourier transforms, random number generation, and more. These functions are highly optimized and provide reliable results for scientific and engineering applications.
- **Compatibility with scientific libraries:** NumPy is the foundation for many other scientific computing libraries in Python, such as SciPy, pandas, Matplotlib, and scikit-learn.

- Using NumPy arrays as the underlying data structure ensures compatibility and seamless integration with these libraries, enabling the development of complex data analysis and machine learning workflows.
- **Broadcasting:** NumPy supports broadcasting, a powerful feature that allows arrays with different shapes to be combined in arithmetic operations. This simplifies the process of performing element-wise operations on arrays with different dimensions, leading to cleaner and more efficient code.

### Here are key aspects of NumPy:

- **Arrays:** The core of NumPy is the ndarray (n-dimensional array), a powerful data structure that allows efficient storage and manipulation of large datasets. Arrays in NumPy can be one-dimensional (1D), two-dimensional (2D), or even higher-dimensional.
- **Operations and functions:** NumPy offers a plethora of functions for mathematical operations on arrays, ranging from basic arithmetic to advanced linear algebra operations.
- **Broadcasting:** NumPy allows operations between arrays of different shapes and dimensions through broadcasting, making code concise and readable.
- **Indexing and slicing:** NumPy provides versatile indexing and slicing capabilities for efficient extraction and manipulation of array elements.
- **Fancy indexing:** It supports advanced indexing techniques, such as boolean indexing and integer array indexing.
- **Vectorized operations:** NumPy promotes vectorized operations, where operations are performed on entire arrays, eliminating the need for explicit looping and enhancing computational efficiency.
- **Integration with other libraries:** NumPy is seamlessly integrated with other scientific computing libraries in Python, such as SciPy, Matplotlib, and scikit-learn, creating a cohesive ecosystem for scientific and data-intensive computing.
- **Performance:** NumPy is implemented in C and Fortran, ensuring high performance for numerical computations. It also provides a convenient C API for interfacing with other low-level languages.

# Overview of pandas



IBM ICE (Innovation Centre for Education)

- Pandas is a robust and versatile data manipulation and analysis library for Python.
- It is built on top of NumPy and provides two primary data structures, Series and DataFrame, that excel in handling labeled and structured data.
- The name "Pandas" is derived from "Panel Data" and "Python Data Analysis".
- It was developed by Wes McKinney in 2008 to address the need for a flexible and efficient data analysis tool in Python.

*Figure 7-18. Overview of pandas*

## Key points about pandas:

- **Data analysis and manipulation:** Pandas provides a wide range of functions and methods for analyzing, cleaning, exploring, and manipulating datasets. It offers tools for data indexing, slicing, merging, reshaping, and more.
- **Usage:** Pandas is widely used in various fields such as data science, finance, research, and more. It is particularly useful for handling large datasets and performing complex data analysis tasks.

## Reasons to use pandas:

- **Analyzing big data:** Pandas enables analysts to analyze big datasets efficiently and make informed decisions based on statistical theories and algorithms. It provides tools for aggregating, summarizing, and visualizing data to extract meaningful insights.
- **Data cleaning:** One of the primary tasks in data analysis is data cleaning, which involves handling missing values, removing duplicates, and transforming data into a readable and relevant format. Pandas offers convenient functions for cleaning messy datasets and preparing them for analysis.
- **Data relevance:** In data science, the quality and relevance of data are crucial for obtaining accurate results and making sound decisions. Pandas facilitates the process of identifying and selecting relevant data, allowing analysts to focus on the most important aspects of their analysis.

Overall, Pandas simplifies the data analysis workflow, making it accessible to users of all levels of expertise. Its intuitive syntax, rich functionality, and broad range of applications make it a preferred choice for data analysis tasks in Python.

**Here's a detailed look at the key features of pandas:**

- **DataFrame:** The DataFrame is the cornerstone of Pandas, offering a two-dimensional, tabular data structure with labeled axes (rows and columns). It is highly efficient for handling heterogeneous data types and managing missing data.
- **Series:** A Series is a one-dimensional array-like object, akin to a column in a DataFrame. It consists of an array of data and an associated array of labels, known as the index.
- **Data alignment and indexing:** Pandas excels in aligning data through its indexes. Operations between different DataFrames automatically align on their index and column labels, simplifying data manipulation.
- **Data cleaning:** Pandas provides a rich set of tools for cleaning and preprocessing data. It facilitates operations like removing duplicates, handling missing values, and reshaping data.
- **Merging and joining:** Pandas supports powerful merging and joining operations, allowing the combination of datasets based on common columns or indices.
- **Time series functionality:** Pandas has robust support for time-series data, offering convenient ways to handle time-based indexing and perform time-related operations.
- **Data input/output:** Pandas supports reading and writing data in various formats, including CSV, Excel, SQL databases, and more, facilitating seamless integration with external data sources.
- **Visualization:** Integration with Matplotlib for data visualization is a notable feature of Pandas. It allows users to create informative plots directly from DataFrame and Series objects.
- **Integration with NumPy:** Pandas is built on top of NumPy, enabling seamless integration with NumPy arrays and functions. This integration enhances performance and provides a unified environment for data manipulation and analysis.



IBM ICE (Innovation Centre for Education)

# Data manipulation with NumPy

- Data manipulation with NumPy encompasses a spectrum of tasks, with a primary focus on array operations and manipulation.
- Leveraging its versatile array objects, NumPy enables efficient reshaping, slicing, and mathematical operations, establishing it as a cornerstone for tasks involving data transformation and manipulation in Python.

*Figure 7-19. Data manipulation with NumPy*

## Creating NumPy arrays

NumPy arrays are instantiated using functions and routines designed to generate arrays of specific shapes and values. For instance, the `np.zeros` function is employed to create arrays filled with zeros, and the `np.linspace` function is utilized for generating arrays with evenly spaced values. Additionally, arrays can be created through operations like stacking and reshaping, allowing for dynamic manipulation of data structures.

A NumPy array can have different numbers of dimensions, commonly referred to as the array's "shape." The term "dimension" is often used interchangeably with "axis." The `shape` attribute of a NumPy array provides information about its dimensions, and various array manipulation functions allow for reshaping and transforming arrays across different dimensions. Understanding dimensions is crucial for performing operations on NumPy arrays, as it determines how the data is organized and accessed.

Let's explore the concept of array creation with examples:

- **0-Dimensional Array (Scalar):** A 0-dimensional array represents a single element.
  - `import numpy as np`
  - `scalar = np.array(42)`
  - `print(scalar) # Output: 42`
- **1-Dimensional Array (Vector):** A 1-dimensional array is a sequence of elements along a single axis.
  - `import numpy as np`
  - `vector = np.array([1, 2, 3])`

- print(vector) # Output: [1, 2, 3]
- **2-Dimensional Array (Matrix):** A 2-dimensional array has rows and columns, forming a matrix.
  - import numpy as np
  - matrix = np.array([[1, 2, 3], [4, 5, 6]])
  - print(matrix)
  - # Output:
- # [[1, 2, 3]]
- # [4, 5, 6]]
- **3-Dimensional Array:** A 3-dimensional array introduces another axis, creating a cuboid or a stack of matrices.

```
import numpy as np
cuboid = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(cuboid)
# Output:
# [[[1, 2]
# [3, 4]]
# [[5, 6]
# [7, 8]]]
```

- **Zeros and Ones:** Creating arrays filled with zeros or ones.

```
import numpy as np
# Creating a 1D array of zeros
zeros_array = np.zeros(5)
print("Zeros Array:", zeros_array)
# Creating a 2D array of ones
ones_array = np.ones((3, 4))
print("Ones Array:", ones_array)
```

- **Arange and Linspace:** Generating arrays with evenly spaced values.

```
import numpy as np
# Creating a 1D array using arange
arange_array = np.arange(0, 10, 2)
print("Arange Array:", arange_array)
# Creating a 1D array using linspace
linspace_array = np.linspace(0, 1, 5)
print("Linspace Array:", linspace_array)
```

### Explanation

- **arange Array:** The np.arange(0, 10, 2) function generates a 1D array starting from 0, incrementing by 2, and stopping before 10. The resulting array is [0, 2, 4, 6, 8]. This array is printed as "Arange Array."
- **Linspace Array:** The np.linspace(0, 1, 5) function creates a 1D array with 5 evenly spaced values between 0 and 1 (inclusive). The resulting array is [0. 0.25, 0.5, 0.75, 1.]. This array is printed as "Linspace Array." Therefore, the expected output when running the code will be:
- Arange Array: [0 2 4 6 8]

- Linspace Array: [0. 0.25 0.5 0.75 1. ]
- **Random:** Generating arrays with random values.

```
import numpy as np

# Creating a 1D array with random values between 0 and 1
random_array = np.random.rand(3)
print("Random Array:", random_array)

# Creating a 2D array with random integers between 1 and 10
random_int_array = np.random.randint(1, 11, size=(2, 3))
print("Random Integer Array:", random_int_array)
```

`numpy.random.choice()` is a function in NumPy that allows you to randomly sample elements from an array or sequence. This function is commonly used for tasks such as random sampling, shuffling, and generating random permutations. It provides a convenient way to randomly select elements from an array or sequence according to specified criteria.

#### Here's an overview:

- **Function:** `numpy.random.choice(a, size=None, replace=True, p=None)`
- **Parameters:** The array or sequence from which to sample elements.
- **size:** The shape of the output. If None, a single value is returned.
- **replace:** Whether sampling is done with replacement. If True, the same element can be sampled multiple times.
- **p:** The probabilities associated with each element in a. If None, all elements are equally likely to be selected.
- **Returns:** An array containing the randomly sampled elements.

```
import numpy as np

# Define an array
arr = np.array([1, 2, 3, 4, 5])

# Sample elements from the array
sampled_elements = np.random.choice(arr, size=3, replace=False)

print(sampled_elements) # Output may vary: e.g., [3 5 1]
```

- **Identity and Eye:** Creating identity and diagonal arrays.

```
import numpy as np

# Creating a 3x3 identity matrix
identity_matrix = np.eye(3)

print("Identity Matrix:\n", identity_matrix)

# Creating a 4x4 diagonal matrix with custom values
diagonal_matrix = np.diag([1, 2, 3, 4])

•print("Diagonal Matrix:\n", diagonal_matrix)
```

- **Reshape and Stack:** Manipulating array shapes and stacking arrays.

```
import numpy as np

# Creating a 1D array
original_array = np.arange(12)

# Reshaping the array into a 3x4 matrix
```

```
reshaped_array = original_array.reshape((3, 4))
print("Reshaped Array:\n", reshaped_array)
# Stacking two arrays vertically
stacked_vertically = np.vstack([reshaped_array, reshaped_array])
print("Stacked Vertically:\n", stacked_vertically)
```

**Output:**

Reshaped Array: The original\_array is initially a 1D array created using np.arange(12). It is then reshaped into a 3x4 matrix using reshape((3, 4)). The resulting matrix is printed as "Reshaped Array":

Reshaped Array:

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

Stacked Vertically: The np.vstack function is used to stack two copies of the reshaped\_array vertically. The resulting array is printed as "Stacked Vertically," and it will be a 6x4 matrix:

Stacked Vertically:

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```



IBM ICE (Innovation Centre for Education)

## Array operations and manipulation

- NumPy simplifies data handling by offering efficient array operations and manipulation, allowing for easy reshaping, slicing, and mathematical operations on arrays in Python.
- It's a versatile toolkit for streamlined data manipulation tasks.
- Understanding axes:
  - In NumPy, the axis parameter is commonly used to specify the axis or axes along which a particular operation should be performed.

*Figure 7-20. Array operations and manipulation*

Let's explore an example with code to understand better how the axis parameter works:

```
import numpy as np

# Creating a 2D array (3x4 matrix)
matrix = np.array([[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12]])

# Example 1: Summing along axis 0 (column-wise)
sum_along_axis_0 = np.sum(matrix, axis=0)
print("Sum along axis 0 (column-wise):", sum_along_axis_0)

# Example 2: Summing along axis 1 (row-wise)
sum_along_axis_1 = np.sum(matrix, axis=1)
print("Sum along axis 1 (row-wise):", sum_along_axis_1)

# Example 3: Finding the maximum value along axis 0
max_along_axis_0 = np.max(matrix, axis=0)
print("Maximum along axis 0 (column-wise):", max_along_axis_0)

# Example 4: Finding the minimum value along axis 1
```

```
min_along_axis_1 = np.min(matrix, axis=1)
print("Minimum along axis 1 (row-wise):", min_along_axis_1)
```

**Output-**

Sum along axis 0 (column-wise): [15 18 21 24]

Sum along axis 1 (row-wise): [10 26 42]

Maximum along axis 0 (column-wise): [ 9 10 11 12]

Minimum along axis 1 (row-wise): [ 1 5 9]



## Broadcasting

- NumPy broadcasting is a powerful mechanism that allows operations on arrays of different shapes and sizes without explicitly reshaping them.
- Broadcasting in NumPy allows arrays with different shapes to be used together in arithmetic operations. Here are the rules for broadcasting in NumPy:
  - Dimensions compatibility.
  - Size compatibility.
  - Broadcasting along multiple dimensions.
  - Final output shape.
  - Broadcasting rule.

*Figure 7-21. Broadcasting*

Broadcasting in NumPy allows arrays with different shapes to be used together in arithmetic operations. Here are the rules for broadcasting in NumPy:

- **Dimensions compatibility:** Two arrays are compatible for broadcasting if their dimensions are either equal or one of them is 1. If the arrays have different numbers of dimensions, the smaller shape is padded with ones on its left side.
- **Size compatibility:** For any dimension where one array has size 1 and the other has a size greater than 1, the array with size 1 is stretched or "broadcast" to match the size of the other array along that dimension.
- **Broadcasting along multiple dimensions:** Broadcasting can occur along multiple dimensions simultaneously. NumPy automatically aligns the shapes of the input arrays, starting from the trailing dimensions and working its way forward.
- **Final output shape:** The final shape of the broadcasted arrays is determined by taking the element-wise maximum along each dimension of the input arrays' shapes.
- **Broadcasting rule:** The broadcasting rule allows the arrays to be broadcast together if, for each dimension, the size of one array is either the same as the size of the corresponding dimension in the other array, or one of them is 1.
- Broadcasting enables NumPy to perform element-wise operations efficiently on arrays with different shapes without explicitly copying data or using loops. It simplifies array arithmetic and makes code more concise and readable.

- Broadcasting automatically aligns dimensions when performing operations, making it more flexible and efficient. Broadcasting follows a set of rules to determine how dimensions should be aligned. It starts with the trailing dimensions and works backward. Two dimensions are compatible for broadcasting if:
  - They are equal.
  - One of them is 1.

This flexibility allows NumPy to handle operations between arrays of different shapes seamlessly. Broadcasting is extensively used in NumPy for performing operations efficiently on arrays with varying dimensions.

Let's explore a simple example to illustrate NumPy broadcasting:

### Example 1

```
import numpy as np

# Creating a 2D array (3x3 matrix)
matrix = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])
```

#### **# Adding a scalar to the entire matrix using broadcasting**

```
scalar_addition_result = matrix + 10
print("Original Matrix:\n", matrix)
print("\nResult after scalar addition:\n", scalar_addition_result)
```

#### **Output:**

Original Matrix:

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Result after scalar addition:

```
[[11 12 13]
 [14 15 16]
 [17 18 19]]
```

**Explanation:** In this example, the scalar value 10 is added to every element of the original matrix using broadcasting. NumPy automatically aligns the dimensions, allowing the operation to be performed without explicitly creating a 3x3 matrix of 10s.

### Example 2

Perform element-wise multiplication between a 2D array and a 1D array using broadcasting:

```
import numpy as np
```

#### **# Creating a 2D array (3x4 matrix)**

```
matrix = np.array([[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12]])
```

#### **# Creating a 1D array (1x4)**

```
row_vector = np.array([0, 1, 2, 3])
```

#### **# Performing element-wise multiplication using broadcasting**

```
result = matrix * row_vector
```

```
print("Original Matrix:\n", matrix)
print("\nRow Vector:\n", row_vector)
print("\nResult after element-wise multiplication:\n", result)
```

### Output

Original Matrix:

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

Row Vector:

```
[0 1 2 3]
```

Result after element-wise multiplication:

```
[[ 0  2  6 12]
 [ 0  6 14 24]
 [ 0 10 22 36]]
```

**Explanation:** In this example, the row vector [0, 1, 2, 3] is multiplied element-wise with each row of the original matrix using broadcasting. NumPy automatically aligns the dimensions, and the multiplication is performed as if the row vector is replicated along the rows. This results in a new matrix where each element is the product of the corresponding elements in the original matrix and the row vector.



## Numpy operations

- Fundamental array creation: NumPy provides functions like `numpy.array()` for creating arrays efficiently.
- Advanced mathematical operations: It offers capabilities for advanced mathematical operations such as dot products and statistical analyses.
- Efficient numerical computations: With seamless integration into Python, NumPy enables efficient handling of large datasets and complex mathematical computations.
- Indispensable tool: NumPy is indispensable for scientific computing, data analysis, and machine learning tasks due to its versatility and efficiency.
- Pivotal role in python: It plays a pivotal role in enhancing Python's capabilities for numerical tasks, serving as a cornerstone of numerical computing in the Python ecosystem.

*Figure 7-22. Numpy operations*

### The data pointer:

- The data pointer in NumPy refers to the memory location where the array's elements are stored.
- It allows efficient access and manipulation of array elements.

### The Data Type or dtype:

- NumPy arrays have a fixed data type (`dtype`) that specifies the type of elements stored in the array.
- The `dtype` can be specified explicitly during array creation or inferred from the input data.
- Common data types include integers, floats, and complex numbers, with different precision levels.

### The shape:

- The shape of a NumPy array defines the size of each dimension of the array.
- It is represented as a tuple of integers, where each integer corresponds to the size of a particular dimension.
- The shape determines the number of elements along each axis of the array.

### The strides:

- Strides in NumPy represent the number of bytes to move from one element to the next along each axis.
- They define the memory layout of the array and are closely related to the shape of the array.
- Strides are important for efficient indexing and slicing of arrays, especially for non-contiguous memory layouts.

### Popular examples of numpy operations:

- **Array creation:** numpy.array(), numpy.zeros(), numpy.ones()
- **Mathematical operations:** numpy.dot(), numpy.sum(), numpy.mean()
- **Statistical functions:** numpy.std(), numpy.median(), numpy.histogram()

**Matrix manipulation:** A new column is added to the matrix using np.hstack(), resulting in the extended\_matrix.

```
import numpy as np
matrix = np.array([[1, 2], [3, 4]])
new_column = np.array([[5], [6]])
extended_matrix = np.hstack([matrix, new_column])
```

Output:

[1 2 5]

[3 4 6]]

- **Flatten a matrix:** The matrix is flattened into a 1D array using np.flatten(), producing the flattened\_array.

```
import numpy as np
matrix = np.array([[1, 2], [3, 4]])
flattened_array = matrix.flatten()
```

Output:

[1 2 3 4]

- **Element-wise matrix multiplication:** Element-wise multiplication is performed by multiplying corresponding elements of matrix1 and matrix2 using the \* operator, yielding the elementwise\_product matrix.

```
import numpy as np
matrix1 = np.array([[1, 2], [3, 4]])
matrix2 = np.array([[5, 6], [7, 8]])
elementwise_product = matrix1 * matrix2
```

Output:

[5 12]

[21 32]]

- **Sum of matrix elements:** The sum of all elements in the matrix is calculated using np.sum(), resulting in the sum\_of\_elements.

```
import numpy as np
matrix = np.array([[1, 2], [3, 4]])
sum_of_elements = np.sum(matrix)
```

10

- **Max and Min values in a matrix:**

The maximum and minimum values in the matrix are obtained using np.max() and np.min(), producing the max\_value and min\_value.

```
import numpy as np
matrix = np.array([[1, 2], [3, 4]])
max_value = np.max(matrix)
```

```
min_value = np.min(matrix)
```

Output:

Max: 4

Min: 1

- **Reshaping a matrix:** The matrix is reshaped into a 1D array using `np.reshape()`, resulting in the `reshaped_array`.

```
import numpy as np
matrix = np.array([[1, 2], [3, 4]])
reshaped_array = np.reshape(matrix, -1)
```

Output:

[1 2 3 4]

- **Identity matrix:** An identity matrix of size 3x3 is generated using `np.eye()`, creating the `identity_matrix`.

```
import numpy as np
```

```
identity_matrix = np.eye(3)
```

Output:

lua

Copy code

`[[1. 0. 0.]`

`[0. 1. 0.]`

`[0. 0. 1.]]`

- **Random matrix:** A random matrix of size 2x3 is generated using `np.random.rand()`, resulting in the `random_matrix`.

```
import numpy as np
```

```
random_matrix = np.random.rand(2, 3)
```

Copy code

`[[0.123 0.456 0.789]`

`[0.321 0.654 0.987]]`



IBM ICE (Innovation Centre for Education)

## Data analysis with pandas

- Data analysis is a crucial aspect of extracting meaningful insights from raw data, and Pandas, a powerful library in Python, provides an extensive toolkit for this purpose.
- Pandas is particularly adept at handling structured data, offering high-performance, easy-to-use data structures such as Series and DataFrame.
- Series represents one-dimensional labeled arrays, while DataFrame is a two-dimensional, tabular structure that resembles a spreadsheet

*Figure 7-23. Data analysis with pandas*

- **Extensive capabilities:** Pandas provides extensive capabilities for data manipulation and analysis, simplifying tasks like indexing, filtering, grouping, and statistical operations.
- **Effortless data manipulation:** Analysts and data scientists can effortlessly manipulate, clean, and analyze data using Pandas, regardless of the data source.
- **Support for various data sources:** Pandas simplifies the process of loading, transforming, and exploring data from diverse sources such as CSV files, Excel spreadsheets, SQL databases, and more.
- **Integration with other libraries:** Its integration with other Python libraries, such as NumPy and Matplotlib, enhances capabilities for statistical analysis and data visualization.
- **Fundamental tool for data analysis:** Pandas serves as a fundamental tool for anyone engaged in data analysis, offering a versatile and efficient platform to unlock valuable insights from datasets. The installation of Pandas is straightforward if Python and PIP are already installed on the system.

Pandas can be installed with the following command:

```
C:\Users\Your Name>pip install pandas
```

In case of command failure, consider using a Python distribution that comes with Pandas pre-installed, such as Anaconda or Spyder.

Upon successful installation, Pandas can be imported into applications using the "import" keyword:]

```
import pandas
```

**Pandas as pd:**

Pandas is usually imported under the pd alias.

alias: In Python alias are an alternate name for referring to the same thing.

Create an alias with the as keyword while importing:

```
import pandas as pd
```

**Checking pandas version:**

The version string is stored under \_\_version\_\_ attribute.

```
import pandas as pd
```

```
print(pd.__version__)
```



IBM ICE (Innovation Centre for Education)

## Working with DataFrames

- The pandas DataFrame is a two-dimensional data structure encompassing data along with its associated labels.
- Employed extensively in data science, machine learning, scientific computing, and various other data-intensive domains, DataFrames share similarities with SQL tables or spreadsheets akin to those in Excel or Calc.
- Frequently, DataFrames exhibit superior performance, ease of use, and enhanced capability compared to traditional tables or spreadsheets, owing to their seamless integration within the Python and NumPy ecosystems.

Figure 7-24. Working with DataFrame

## Components of a DataFrame

### The Columns, Index, and Data

Columns												
	Index	Subscriber	Gender	Address	Identifiers	Timestamp	Subscription	Street_Address	Latitude_North	Longitude_West	Neighborhood	Zipcode
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-18:17:00	985	Lake Shore Dr & Monroe St	41.8811	-87.617	11	Michigan Ave & Oak St	41
1	7524	Subscriber	Male	2013-06-28 20:53:00	2013-06-23:03:00	623	Clinton St & Washington Blvd	41.8834	-87.6412	31	Wells St & Walton St	41
2	10807	Subscriber	Male	2013-06-30 14:43:00	2013-06-15:01:00	1040	Shelford Ave & Kingsbury St	41.8986	-87.6035	15	Dearborn St & Monroe St	41
3	12967	Subscriber	Male	2013-07-01 10:05:00	2013-07-10:16:00	867	Cayenne St & Huron St	41.8946	-87.6034	19	Cook St & Randolph St	41
4	13168	Subscriber	Male	2013-07-01 11:18:00	2013-07-11:18:00	130	Damen Ave & Pierce Ave	41.8984	-87.6777	19	Damen Ave & Pierce Ave	41

Data												
Description	Alternative Names	Axis Number										
<ul style="list-style-type: none"> <li>Columns - label each column</li> <li>Index - label each row</li> <li>Data - actual values in DataFrame</li> </ul>	<ul style="list-style-type: none"> <li>Columns - column names/labels, column index</li> <li>Index - index names/labels, row names/labels</li> <li>Data - values</li> </ul>	<ul style="list-style-type: none"> <li>Columns: 1</li> <li>Index: 0</li> </ul>										

Figure 3: Components of a Dataframe

[<https://medium.com/dunder-data/the-pandas-dataframe-and-series>]

Pandas DataFrames are data structures that contain:

- Data organized in two dimensions, rows and columns
- Labels that correspond to the rows and columns



## Creating pandas dataframe

- A pandas DataFrame can be created through various methods.
- Typically, the DataFrame constructor is utilized, and information such as data, labels, and other relevant details are supplied.
- The data can be presented as a two-dimensional list, tuple, or NumPy array.
- Various ways of creating a pandas dataframe is mentioned below-

Column labels  
↓  
**name**    **city**    **age**    **py-score**

	<b>name</b>	<b>city</b>	<b>age</b>	<b>py-score</b>
<b>101</b>	Xavier	Mexico City	41	88.0
<b>102</b>	Ann	Toronto	28	79.0
<b>103</b>	Jana	Prague	33	81.0
<b>104</b>	Yi	Shanghai	34	80.0
<b>105</b>	Robin	Manchester	38	68.0
<b>106</b>	Amal	Cairo	31	61.0
<b>107</b>	Nori	Osaka	37	84.0

↑ Row labels      ↑ Data

Figure 7-25. Creating pandas dataframe

### Creating a pandas dataframe with dictionaries

In the context of this example, it is assumed that a dictionary is being employed to convey the data.

```
data = {
    'name': ['Xavier', 'Ann', 'Jana', 'Yi', 'Robin', 'Amal', 'Nori'],
    'city': ['Mexico City', 'Toronto', 'Prague', 'Shanghai',
             'Manchester', 'Cairo', 'Osaka'],
    'age': [41, 28, 33, 34, 38, 31, 37],
    'py-score': [88.0, 79.0, 81.0, 80.0, 68.0, 61.0, 84.0]
}
```

row\_labels = [101, 102, 103, 104, 105, 106, 107]

In the above example, data is a python variable that refers to the dictionary that holds the candidate data. It contains the labels of the columns:

- 'name'
- 'city'
- 'age'
- 'py-score'

row\_labels refers to a list that contains the labels of the rows, which are numbers ranging from 101 to 107.

### **Creating a pandas dataframe with lists**

```
I = [{"x": 1, "y": 2, "z": 100},
      {"x": 2, "y": 4, "z": 100},
      {"x": 3, "y": 8, "z": 100}]
pd.DataFrame(I)
```

#### **Output-**

```
x y z
0 1 2 100
1 2 4 100
2 3 8 100
```

Here, the dictionary keys are the column labels, and the dictionary values are the data values in the DataFrame.

### **Creating a pandas dataframe with numpy arrays**

A two-dimensional NumPy array can be passed to the DataFrame constructor the same as a list:

```
arr = np.array([[1, 2, 100],
               [2, 4, 100],
               [3, 8, 100]])
df_ = pd.DataFrame(arr, columns=['x', 'y', 'z'])
```

#### **Output-**

```
x y z
0 1 2 100
1 2 4 100
2 3 8 100
```

### **Creating a pandas DataFrame from files**

Creating a Pandas DataFrame from files is a common operation in data analysis, enabling the efficient loading and manipulation of structured data. Various file formats, such as CSV, Excel, JSON, and more, are supported by Pandas for this purpose. This overview provides insights into the process of creating a Pandas DataFrame from files.

- **Loading Data from CSV:**

```
import pandas as pd
# Load data from a CSV file into a DataFrame
df_csv = pd.read_csv('your_file.csv')
```

- **Loading Data from Excel:**

```
import pandas as pd
# Load data from an Excel file into a DataFrame
df_excel = pd.read_excel('your_file.xlsx', sheet_name='Sheet1')
```

- **Loading Data from JSON:**

```
import pandas as pd
# Load data from a JSON file into a DataFrame
df_json = pd.read_json('your_file.json')
```

- **Loading Data from SQL Database:**

```
import pandas as pd
import sqlite3
# Connect to a SQLite database
conn = sqlite3.connect('your_database.db')
# Query data from a table into a DataFrame
df_sql = pd.read_sql_query('SELECT * FROM your_table', conn)
# Close the database connection
conn.close()
```



IBM ICE (Innovation Centre for Education)

## Retrieving labels and data

- Retrieving labels and data from a Pandas DataFrame involves accessing specific rows, columns, or individual elements.
- Here are several common operations for retrieving labels and data in Pandas.

*Figure 7-26. Retrieving labels and data*

- **Accessing columns by label:**

```
import pandas as pd

# Create a sample DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35],
        'City': ['New York', 'San Francisco', 'Los Angeles']}
df = pd.DataFrame(data)

# Retrieve a specific column by label
name_column = df['Name']
```

- **Accessing rows by label:**

```
import pandas as pd

# Create a sample DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35],
        'City': ['New York', 'San Francisco', 'Los Angeles']}
df = pd.DataFrame(data, index=['a', 'b', 'c'])
```

```
# Retrieve a specific row by label
row_b = df.loc['b']

• Accessing specific data by row and column label:

import pandas as pd
# Create a sample DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35],
        'City': ['New York', 'San Francisco', 'Los Angeles']}
df = pd.DataFrame(data, index=['a', 'b', 'c'])

# Retrieve specific data by row and column label
specific_data = df.at['b', 'Age']

• Accessing data by position:

import pandas as pd
# Create a sample DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35],
        'City': ['New York', 'San Francisco', 'Los Angeles']}
df = pd.DataFrame(data)

# Retrieve data by position (row 1, column 2)
data_position = df.iat[1, 2]
```

### **Popular data manipulation functions:**

Method	Description	Example
head()	Display the initial rows of a DataFrame.	import pandas as pd data = {'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'], 'Age': [25, 30, 35, 28, 22], 'City': ['NY', 'SF', 'LA', 'Chicago', 'Miami']} df = pd.DataFrame(data) print(df.head(2))
tail()	Display the final rows of a DataFrame.	import pandas as pd data = {'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'], 'Age': [25, 30, 35, 28, 22], 'City': ['NY', 'SF', 'LA', 'Chicago', 'Miami']} df = pd.DataFrame(data) print(df.tail(3))
info()	Provide a concise summary of a DataFrame, including data types and missing values.	print(df.info())
describe()	Generate descriptive statistics of a DataFrame, such as mean, standard deviation, and quartiles.	print(df.describe())
value_counts()	Count unique values in a column.	print(df['City'].value_counts())
groupby()	Group the DataFrame by a specified column and calculate the mean age for each group.	print(df.groupby('City')['Age'].mean())
merge()	Combine two DataFrames based on a common key, creating a new DataFrame with merged information.	df1 = pd.DataFrame({'ID': [1, 2, 3], 'Name': ['Alice', 'Bob', 'Charlie']}) df2 = pd.DataFrame({'ID': [2, 3, 4], 'Salary': [50000, 60000, 70000]}) merged_df = pd.merge(df1, df2, on='ID') print(merged_df)
pivot_table()	Create a spreadsheet-style pivot table. Calculate the mean age for each city.	print(pd.pivot_table(df, values='Age', index='City', aggfunc='mean'))
apply()	Apply a function to each element in a column. Subtract 2 from each age using a lambda function, creating a new column.	df['Discounted_Age'] = df['Age'].apply(lambda x: x - 2) print(df)
sort_values()	Sort the DataFrame based on the specified column in descending order. Sort by 'Age' in this example.	print(df.sort_values(by='Age', ascending=False))

Table 4: Popular Data manipulation functions

## Data cleaning and exploration



IBM ICE (Innovation Centre for Education)

- Data cleaning and exploration involve the preparation and analysis of datasets, processes necessary for ensuring data quality and gaining insights.
- It is required to identify and address inconsistencies, errors, and missing values in the data, as well as to understand its characteristics before analysis.

---

*Figure 7-27. Data cleaning and exploration*

In this context, Pandas proves valuable by offering tools for handling missing data, filtering, sorting, and transforming datasets, allowing for seamless data exploration and cleaning without requiring extensive manual intervention. In real-world data science projects, the datasets used for analysis often contain various imperfections and inconsistencies. For example, these imperfections may include missing data, redundant entries, data in incorrect formats, and the presence of outliers.

Data cleaning involves the preprocessing and transformation of raw data to make it suitable for further analysis, whether it's for descriptive analysis through data visualization or for building predictive models. It's crucial to ensure that the data used for analysis is clean, accurate, and reliable because the quality of the data directly impacts the quality of the predictive models and insights derived from the analysis.



## Handling missing values

- Missing values in a dataset refer to the absence of data in specific entries or variables.
- These values are typically denoted by placeholders like "NaN" (Not a Number) in Python or "NULL" in databases.
- Missing values can occur for various reasons, such as data entry errors, equipment malfunctions, or intentional non-responses in surveys.
- Dealing with missing values is a crucial aspect of data cleaning and analysis, as they can impact the accuracy and reliability of statistical conclusions drawn from the data.
- Strategies for handling missing values include imputation, where missing values are estimated or replaced, or exclusion, where rows or columns with missing values are omitted from analysis.

*Figure 7-28. Handling missing values*

In the below example, a DataFrame (`df_missing_values`) with missing values (represented by `np.nan`) is created. The `isnull()` function is then used to identify missing values, generating a DataFrame of boolean values indicating the presence of missing values in each cell.

Subsequently, the `dropna()` function is employed to remove rows containing any missing values, resulting in a DataFrame (`df_no_missing_values`) without any missing values.

```
import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],
        'Age': [25, None, 35, 28, 22],
        'City': ['NY', 'SF', 'LA', 'Chicago', None]}
df = pd.DataFrame(data)

# Check for missing values
print(df.isnull())

Output:
   Name  Age  City
0  False False  False
1  False  True  False
2  False False  False
```

```

3 False False False
4 False False True
# Drop rows with missing values
df_cleaned = df.dropna()
# Fill missing values with mean
df['Age'].fillna(df['Age'].mean(), inplace=True)
print(df_cleaned)

```

Output:

	Name	Age	City
0	Alice	25.0	NY
2	Charlie	35.0	LA
1.	David	28.0	Chicago

### **Handling outliers:**

Outliers are data points that deviate significantly from the rest of the data. They can skew statistical analyses and machine learning models, making it essential to handle them appropriately.

Example Code:

```

import numpy as np
import pandas as pd

# Generate sample data with outliers
data = pd.DataFrame({'Value': np.random.normal(loc=100, scale=20, size=100)})
data.iloc[0:5] = 500 # Introduce outliers
# Detect outliers using z-score
z_scores = (data - data.mean()) / data.std()
outliers = data[(z_scores > 3) | (z_scores < -3)]
# Replace outliers with median
median = data.median()
data_outliers_removed = data.mask((z_scores > 3) | (z_scores < -3), median, axis=1)

print("Original Data with Outliers:")
print(data.head())
print("\nOutliers Detected:")
print(outliers)
print("\nData with Outliers Removed:")
print(data_outliers_removed.head())

```

### **Encoding categorical variables:**

Categorical variables are non-numeric data that represent categories or groups. They need to be encoded into numerical values for machine learning algorithms to process them effectively.

Example Code:

```

import pandas as pd
# Sample dataset with categorical variables

```

```

data = pd.DataFrame({'Category': ['A', 'B', 'C', 'A', 'C', 'B', 'A', 'B']})
# One-hot encoding
one_hot_encoded = pd.get_dummies(data['Category'], prefix='Category')
print("Original Data:")
print(data)
print("\nOne-Hot Encoded Data:")
print(one_hot_encoded)

```

### **Feature engineering:**

Feature engineering involves creating new features or modifying existing ones to improve the performance of machine learning models. In the below example, we're creating new features such as 'Total\_Area' (house size + additional area per bedroom), 'Age' of the house, and 'Bathrooms\_per\_Bedroom' ratio. We then scale these features using StandardScaler to bring them to a similar scale. These new features can provide additional insights and improve the performance of machine learning models.

### **Example Code:**

```

import pandas as pd
from sklearn.preprocessing import StandardScaler
# Sample dataset with features
data = pd.DataFrame({
    'House_Size': [1500, 2000, 1200, 1800, 2200],
    'Num_Bedrooms': [3, 4, 2, 3, 4],
    'Num_Bathrooms': [2, 2.5, 1.5, 2, 3],
    'Year_Built': [1990, 2005, 1985, 1998, 2010],
    'Garage_Size': [2, 2, 1, 2, 3],
    'Price': [250000, 350000, 200000, 300000, 400000]
})
# Creating new features based on existing ones
data['Total_Area'] = data['House_Size'] + (data['Num_Bedrooms'] * 50) # Adding 50 sqft per bedroom
data['Age'] = 2024 - data['Year_Built'] # Calculating age of the house
data['Bathrooms_per_Bedroom'] = data['Num_Bathrooms'] / data['Num_Bedrooms'] # Ratio of bathrooms to bedrooms
# Scaling numerical features using StandardScaler
scaler = StandardScaler()
scaled_features = scaler.fit_transform(data[['Total_Area', 'Age', 'Bathrooms_per_Bedroom', 'Garage_Size']])
# Adding scaled features to the dataframe
data[['Total_Area', 'Age', 'Bathrooms_per_Bedroom', 'Garage_Size']] = scaled_features
print("Original Data:")
print(data)

```

### **Commonly used data cleaning tools:**

- Pandas: Pandas is a powerful library in Python for data manipulation and analysis. It provides functions and methods for handling missing data, removing duplicates, transforming data, and more.
- NumPy: NumPy is another essential library for numerical computing in Python. It offers functions for mathematical operations, array manipulation, and handling missing values.
- scikit-learn: scikit-learn is a machine learning library in Python that provides tools for data preprocessing, including imputation of missing values, feature scaling, and encoding categorical variables.
- OpenRefine: OpenRefine is a free, open-source tool for data cleaning and transformation. It offers a user-friendly interface for exploring and cleaning messy data, removing duplicates, and performing operations like clustering and normalization.
- Trifecta wrangler: Trifecta Wrangler is a commercial data preparation platform that offers visual tools for cleaning and transforming data. It provides features like automatic data profiling, smart suggestions for cleaning operations, and collaboration features for teams.
- Dedupe: Dedupe is a Python library for data deduplication. It helps in identifying and removing duplicate records from datasets, which is a common data cleaning task.
- Excel: Excel is a widely used spreadsheet software that offers built-in tools for data cleaning, such as filtering, sorting, removing duplicates, and performing basic calculations. While not as powerful as specialized data cleaning tools, Excel can still be useful for simple data cleaning tasks.

These tools can be used individually or in combination to effectively clean and prepare datasets for analysis or modeling.

## **Data exploration**

Data cleaning and exploration are crucial steps in the data analysis process. Let's explore various aspects in-depth with code examples using the Pandas library in Python. Data exploration refers to the initial step in data analysis where data analysts utilize data visualization and statistical techniques to describe dataset characteristics such as size, quantity, and accuracy. The primary objective is to gain a deeper understanding of the underlying structure and patterns within the data. By exploring the data, analysts can identify trends, anomalies, and relationships that may inform subsequent steps in the analysis process. This process often involves generating summary statistics, creating visualizations like histograms, scatter plots, and box plots, and conducting exploratory data analysis (EDA) to uncover insights and potential areas of interest. Overall, data exploration plays a crucial role in informing subsequent analysis and decision-making processes by providing valuable insights into the nature and quality of the data.

### **# Summary statistics**

```
print(df.describe())
```

Output:

	Age
count	5.000000
mean	27.500000
std	5.315073
min	22.000000
25%	25.000000
50%	28.000000
75%	30.000000
max	35.000000

### **# Unique values in a column**

```
print(df['City'].unique())
```

Output:

```
[NY' 'SF' 'LA' 'Chicago' None]
```

**# Count occurrences of each value**

```
print(df['City'].value_counts())
```

Output:

```
NY      1
```

```
LA      1
```

```
Chicago  1
```

```
SF      1
```

Name: City, dtype: int64

**# Filtering data**

```
young_people = df[df['Age'] < 30]
```

```
print(young_people)
```

Output:

	Name	Age	City
--	------	-----	------

```
0 Alice 25.0 NY
```

```
3 David 28.0 Chicago
```

```
4 Eva 22.0 None
```

**# Sorting**

```
df_sorted = df.sort_values(by='Age', ascending=False)
```

```
print(df_sorted)
```

Output:

	Name	Age	City
--	------	-----	------

```
2 Charlie 35.0 LA
```

```
3 David 28.0 Chicago
```

```
0 Alice 25.0 NY
```

```
4 Eva 22.0 None
```

```
1 Bob 27.5 SF
```

**# Grouping and aggregation**

```
age_mean_by_city = df.groupby('City')['Age'].mean()
```

```
print(age_mean_by_city)
```

Output:

City

```
Chicago 28.0
```

```
LA      35.0
```

```
NY      25.0
```

```
SF      27.5
```

Name: Age, dtype: float64

**# Creating new columns**

```
df['Is_Young'] = df['Age'] < 30
```

```
print(df)
```

Output:

	Name	Age	City	Is_Young
0	Alice	25.0	NY	True
1	Bob	27.5	SF	True
2	Charlie	35.0	LA	False
3	David	28.0	Chicago	True
2.	Eva	22.0	None	True

Automated data exploration tools, including data visualization software, empower data scientists to efficiently monitor data sources and analyze large datasets. Graphical representations, such as bar charts and scatter plots, are invaluable for visualizing data and gaining insights.

Additionally, Microsoft Excel spreadsheets are a widely used tool for manual data exploration. Excel enables users to create basic charts, visualize raw data, and identify correlations between variables. While Excel may not offer the advanced features of dedicated data visualization software, it provides a user-friendly interface for preliminary data analysis and exploration.



IBM ICE (Innovation Centre for Education)

## Dealing with duplicates

- Data cleaning and exploration involve the preparation and analysis of datasets, processes necessary for ensuring data quality and gaining insights.
- Duplicate values in a DataFrame refer to rows that have identical values across all columns.
- These duplicates can arise from data entry errors, data merging, or other data manipulation processes.
- Pandas provides tools to detect and handle duplicate values efficiently.
- In the below example, a DataFrame with duplicate values is first created.
- The duplicated() function is then utilized to identify duplicate rows, resulting in a boolean Series indicating the duplicity of each row.
- Subsequently, the drop\_duplicates() function is employed to eliminate duplicate rows, yielding a DataFrame containing only unique rows based on the specified columns ('ID' and 'Name' in this case).

*Figure 7-29. Dealing with duplicates*

### # Create a DataFrame with duplicates

```
data = {'ID': [1, 2, 2, 3, 4],
        'Name': ['Alice', 'Bob', 'Bob', 'Charlie', 'David']}
df_duplicates = pd.DataFrame(data)
```

### # Check for duplicates

```
print(df_duplicates.duplicated())
```

Output:

```
0  False
1  False
2  True
3  False
4  False
```

```
dtype: bool
```

### # Drop duplicates

```
df_no_duplicates = df_duplicates.drop_duplicates()
print(df_no_duplicates)
```

Output:

ID Name

0 1 Alice

1 2 Bob

3 3 Charlie\*

4 4 David



IBM ICE (Innovation Centre for Education)

## Database applications

- Databases are utilized, and connections are established, with records being created, read, updated, and deleted to meet the dynamic requirements of applications.
- The execution of these operations involves the formulation and execution of SQL queries, allowing for efficient communication between Python and the chosen database management system.
- Changes are committed, and resources are managed by creating and closing connections and cursors.
- This section provides a comprehensive overview of the database application's functionality.

---

Figure 7-30. Database applications

### Database connectivity in Python

- Python offers database connectivity through various libraries for interaction with relational database systems.
- Supported database management systems include MySQL, PostgreSQL, SQLite, among others.
- Developers can connect, query, and manipulate data programmatically using Python.
- The connection process involves specifying parameters such as the database server address, username, and password.
- SQL queries can be executed through dedicated modules once the connection is established.
- Python facilitates fetching and modifying data efficiently.
- Connection objects and cursor objects are used in these libraries for effective communication with the database.
- Robust database connectivity in Python enables seamless integration of data storage and retrieval into applications.
- This enhances the versatility and functionality of Python in diverse software projects.
- Python offers powerful features for database programming, supporting various databases including SQLite, MySQL, Oracle, Sybase, PostgreSQL, MongoDB, and more.
- It supports Data Definition Language (DDL), Data Manipulation Language (DML), and Data Query Statements for database operations.

The Python standard for database interfaces is the Python Database API (DB-API), providing a consistent interface for interacting with different database systems.

### Database Management Systems (DBMS)

- Database Management Systems (DBMS) are essential components in information technology, providing structured data storage, retrieval, and management.
- DBMS software facilitates the creation, modification, and querying of databases, ensuring data integrity, managing user access, and optimizing data retrieval efficiency.
- The relational database model organizes data into tables with defined relationships, promoting relational integrity.
- Popular examples of DBMS include MySQL, PostgreSQL, Oracle Database, and Microsoft SQL Server, each serving specific requirements and preferences.
- DBMS play a crucial role in supporting applications, websites, and software systems, offering a secure and scalable framework for managing large volumes of structured data.

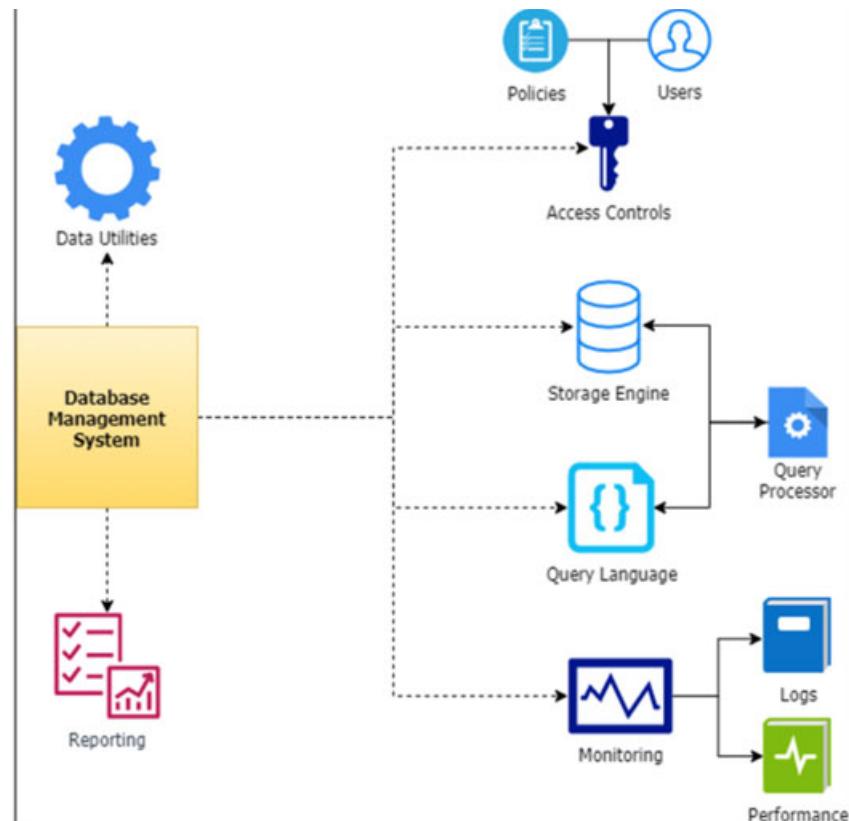


Figure 4: Components of DBMS

[<https://www.bmc.com/blogs/dbms-database-management-systems/>]

## Connecting to databases



IBM ICE (Innovation Centre for Education)

- Connecting to databases in Python is a fundamental process facilitated by various database connectors and libraries.
- Python supports a diverse range of Database Management Systems (DBMS) such as MySQL, PostgreSQL, SQLite, and more.
- To establish a connection, developers typically utilize dedicated libraries like mysql-connector-python, psycopg2 for PostgreSQL, or sqlite3 for SQLite.
- The connection process involves providing essential details such as the database server address, port, username, password, and database name.

---

Figure 7-31. Connecting to databases

Here's a simple example using sqlite3 for connecting to an SQLite database:

```
import sqlite3
# Connect to the SQLite database
connection = sqlite3.connect('example.db')
# Create a cursor object to execute SQL queries
cursor = connection.cursor()
# Execute a sample query
cursor.execute('SELECT * FROM users')
# Fetch and print the results
results = cursor.fetchall()
for row in results:
    print(row)
# Close the cursor and connection
cursor.close()
connection.close()
```

This code snippet illustrates the basic steps of connecting to an SQLite database, executing a query, and fetching results. Similar principles apply when connecting to other databases, with the specific library and connection parameters adjusted accordingly. These database connections empower Python developers to integrate their applications with diverse data storage solutions seamlessly.



IBM ICE (Innovation Centre for Education)

## SQL queries and database operations

- SQL queries and database operations constitute a fundamental aspect of database management, allowing developers to interact with and manipulate data within a database.
- Structured Query Language (SQL) serves as the standard language for performing these operations.
- In the context of Python, developers leverage SQL queries to communicate with relational database systems seamlessly.

---

Figure 7-32. SQL queries and database operations

Various operations are accomplished through SQL queries, including:

**Data retrieval:**

SELECT column1, column2 FROM table WHERE condition;

Retrieve specific columns from a table based on specified conditions.

**Data insertion:**

INSERT INTO table (column1, column2) VALUES (value1, value2);

Insert new records into a table.

**Data update:**

UPDATE table SET column1 = value1 WHERE condition;

Modify existing records in a table based on certain conditions.

**Data deletion:**

DELETE FROM table WHERE condition;

Remove records from a table based on specified conditions.

These SQL queries are executed using Python's database connectors and cursor objects. Python provides a straightforward interface for developers to perform database operations programmatically, making it an integral tool for applications requiring robust data management capabilities. This seamless integration of SQL

queries and database operations in Python contributes to the efficiency and effectiveness of data-driven applications.

In the domain of SQL Queries and Database Operations, proficiency in Structured Query Language (SQL) is paramount. SQL serves as the universal language for interacting with relational databases, and executing SQL Queries from Python seamlessly integrates database operations into Python applications, enabling dynamic and efficient data manipulation and retrieval.



IBM ICE (Innovation Centre for Education)

# Structured Query Language (SQL)

- Structured Query Language (SQL) is a powerful and standardized language designed for managing and manipulating relational databases.
- Here are key steps and concepts in SQL:
  - Database creation.
  - Table creation.
  - Data insertion.
  - Data retrieval.
  - Data update.
  - Data deletion.
  - Aggregation.
  - Joining Tables.
  - Indexing.

*Figure 7-33. Structured Query Language (SQL)*

Here are key steps and concepts in SQL:

- **Database creation:** Use CREATE DATABASE to create a new database.  
CREATE DATABASE dbname;
- **Table creation:** Employ CREATE TABLE to define a new table and its structure.

```
CREATE TABLE tablename (
  column1 datatype,
  column2 datatype,
  ...
);
```

- **Data insertion:** Utilize INSERT INTO to add new records to a table.

```
INSERT INTO tablename (column1, column2, ...) VALUES (value1, value2, ...);
```

- **Data retrieval:** Employ SELECT to retrieve data from one or more tables.

```
SELECT column1, column2 FROM tablename WHERE condition;
```

- **Data update:** Use UPDATE to modify existing records in a table.

```
UPDATE tablename SET column1 = value1 WHERE condition;
```

- **Data deletion:** Utilize DELETE to remove records from a table based on specified conditions.

DELETE FROM tablename WHERE condition;

Filtering and Sorting: Incorporate WHERE for filtering results and ORDER BY for sorting.

SELECT column1, column2 FROM tablename WHERE condition ORDER BY column1 ASC;

- **Aggregation:** Apply aggregate functions like SUM, AVG, COUNT, etc., for data summarization.

SELECT AVG(column1) FROM tablename WHERE condition;

- **Joining Tables:**

Use JOIN to combine data from multiple tables based on related columns.

SELECT \* FROM table1 INNER JOIN table2 ON table1.column = table2.column;

- **Indexing:** Create indexes using CREATE INDEX to enhance data retrieval performance.

CREATE INDEX indexname ON tablename (column);

These fundamental SQL steps form the basis for effective database management, enabling developers to create, manipulate, and retrieve data with precision and efficiency.

# Executing SQL queries from python (1 of 2)



IBM ICE (Innovation Centre for Education)

- Executing SQL queries from Python involves several steps, seamlessly integrating Python code with relational database systems.
- Let's delve into each step of executing SQL queries from Python in detail.
- By following these steps, developers can seamlessly integrate SQL queries into their Python applications, allowing for dynamic and interactive interaction with relational databases.
- This approach is applicable across various database management systems, with adjustments made to the specific connector library and connection parameters.

*Figure 7-34. Executing SQL queries from python (1 of 2)*

- **Importing the database connector:** Begin by importing the appropriate database connector library. For example, if working with SQLite, use the sqlite3 module:

```
import sqlite3
```

- **Establishing a database connection:** Use the connect() function from the imported module to establish a connection to the database. Provide connection parameters such as the database name or file.

```
connection = sqlite3.connect('example.db')
```

- **Creating a cursor object:** A cursor object is essential for executing SQL queries. It acts as a pointer or iterator to navigate through the results of the executed queries.

```
cursor = connection.cursor()
```

- **Executing SQL Queries:** Formulate SQL queries as strings and execute them using the execute() method of the cursor.

```
cursor.execute('SELECT * FROM tablename WHERE condition;')
```

- **Fetching results:** After executing a SELECT query, fetch the results using methods like fetchall(), fetchone(), or fetchmany(size).

```
results = cursor.fetchall()
```

- **Processing results:** Process the fetched results according to the application's requirements. Results are often stored in variables for further analysis or display.

```
for row in results:
```

```
print(row)
```

- **Closing cursor and connection:** Always close the cursor and the connection after executing queries to release resources and ensure proper database interaction.

```
cursor.close()
```

```
connection.close()
```

## Executing SQL queries from python (2 of 2)



IBM ICE (Innovation Centre for Education)

- Below is a simple Python project that takes user registration data through a form, processes the request, and stores the data into a SQLite database.
- This example uses the Flask framework for the web application and SQLite for the database.

Figure 7-35. Executing SQL queries from python (2 of 2)

- **Install flask:** pip install flask
- **Create a file named app.py with the following content:**
- from flask import Flask, render\_template, request, redirect
- import sqlite3

```
app = Flask(__name__)
# SQLite database setup
conn = sqlite3.connect('user_database.db')
cursor = conn.cursor()
cursor.execute('''
    CREATE TABLE IF NOT EXISTS users (
        id INTEGER PRIMARY KEY,
        username TEXT NOT NULL,
        email TEXT NOT NULL
    );
''')
conn.commit()
```

```

conn.close()
@app.route('/')
def index():
    return render_template('index.html')
@app.route('/register', methods=['POST'])
def register():
    if request.method == 'POST':
        username = request.form['username']
        email = request.form['email']
        # Insert user data into the database
        conn = sqlite3.connect('user_database.db')
        cursor = conn.cursor()
        cursor.execute('INSERT INTO users (username, email) VALUES (?, ?)', (username, email))
        conn.commit()
        conn.close()
        return redirect('/success')
@app.route('/success')
def success():
    return render_template('success.html')
if __name__ == '__main__':
    app.run(debug=True)

```

- **Create a folder named templates and inside it, create two HTML files:**

#### **index.html:**

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>User Registration</title>
</head>
<body>
    <h1>User Registration</h1>
    <form method="POST" action="/register">
        <label for="username">Username:</label>
        <input type="text" name="username" required><br>
        <label for="email">Email:</label>
        <input type="email" name="email" required><br>
        <button type="submit">Register</button>
    </form>

```

```
</body>
</html>
success.html:
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Success</title>
</head>
<body>
    <h1>Registration Successful</h1>
    <p>Thank you for registering!</p>
</body>
</html>
```

### **Run the flask application:** Python app.py

Upon visiting <http://127.0.0.1:5000/> in the web browser, the registration form can be completed, and upon submission, redirection to a success page is expected. The user data is then stored in the user\_database.db file. This example is foundational, and in a real-world context, the incorporation of additional features, validations, and security measures is recommended. Furthermore, for production-grade applications, the consideration of a more robust database system, such as PostgreSQL or MySQL, is advisable.



IBM ICE (Innovation Centre for Education)

## Building database-driven applications

- In the development of database-driven applications, a seamless integration of data management functionalities is achieved through the orchestration of database interactions.
- The user's experience is enhanced as data is dynamically stored, retrieved, updated, and deleted within the application.

---

Figure 7-36. Building database-driven applications

Emphasis is placed on the establishment of connections to chosen database management systems, fostering a fluid exchange of information between Python and the designated databases. Furthermore, the creation and utilization of cursors play a pivotal role in executing Structured Query Language (SQL) commands, enabling precise control over data manipulation. Committing changes ensures the persistence of modifications, while the meticulous closure of cursors and connections signifies the responsible management of resources.

### Building data-driven applications

- **Data quality assurance:**
  - Ensure data accuracy, consistency, completeness, and reliability.
  - Implement data validation and cleansing processes.
  - Regularly monitor and audit data quality metrics.
- **User interface design:**
  - Design intuitive and user-friendly interfaces.
  - Incorporate interactive data visualization for better understanding.
  - Prioritize usability and accessibility for diverse user demographics.
- **Testing and debugging:**
  - Conduct thorough testing of all app functionalities.

- Implement unit tests, integration tests, and end-to-end tests.
- Establish effective debugging mechanisms to identify and resolve issues promptly.
- **Deployment and maintenance:**
  - Plan a robust deployment strategy, considering scalability and performance.
  - Regularly update and maintain the app to address evolving requirements and bugs.
  - Implement version control and rollback procedures for seamless maintenance.
- **Security considerations:**
  - Implement encryption techniques to protect sensitive data.
  - Secure authentication and authorization mechanisms.
  - Regularly update security patches and monitor for vulnerabilities.

# Integrating python with databases



IBM ICE (Innovation Centre for Education)

- Building database-driven applications involves integrating Python seamlessly with databases, allowing for dynamic data storage, retrieval, and manipulation.
- The steps involving data integration using Python are explained below.

*Figure 7-37. Integrating python with databases*

- **Choosing a Database Management System (DBMS):** Select a suitable DBMS that aligns with the application's requirements. Common choices include MySQL, PostgreSQL, SQLite, and MongoDB.
- **Importing database connector:** Import the relevant database connector module in Python. For example, use sqlite3 for SQLite, mysql.connector for MySQL, or psycopg2 for PostgreSQL.

```
import sqlite3
```

- **Establishing database connection:** Use the connector's connect() function to establish a connection to the database by providing connection parameters like the database name, username, password, and host.

```
connection = sqlite3.connect('example.db')
```

- **Creating a cursor:** Create a cursor object using the connection. The cursor is essential for executing SQL queries and fetching results.

```
cursor = connection.cursor()
```

- **Executing SQL queries:** Formulate SQL queries as strings and execute them using the cursor's execute() method.

```
cursor.execute('CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY, name TEXT, age INTEGER);')
```

Committing changes: For operations that modify the database (e.g., INSERT, UPDATE, DELETE), commit the changes to persist them.

```
connection.commit()

• Fetching and processing results: For SELECT queries, fetch the results using methods like fetchall(), fetchone(), or fetchmany(size). Process the results as needed.

cursor.execute('SELECT * FROM users;')

results = cursor.fetchall()

for row in results:
    print(row)

• Closing cursor and connection: Always close the cursor and connection to release resources and maintain proper database interaction.

cursor.close()

connection.close()
```

This seamless integration empowers developers to leverage the capabilities of relational databases within their Python applications, enabling robust and efficient data-driven functionality. The specific steps may vary based on the chosen DBMS and connector library, but the core principles remain consistent.

# CRUD operations in database applications



IBM ICE (Innovation Centre for Education)

- CRUD stands for Create, Read, Update, and Delete. It represents the fundamental operations used to interact with and manage data in databases.
- Let's explore CRUD operations in the context of a Python project involving a simple database application.
- For this illustration, SQLite is used as the database system and performs CRUD operations on a "users" table.

*Figure 7-38. CRUD operations in database applications*

### Step 1: Database connection establishment

```
import sqlite3

# Step 1: Establish a connection to the SQLite database (creates a new file if not exists)
connection = sqlite3.connect('user_database.db')

# Create a cursor object to execute SQL queries
cursor = connection.cursor()
```

### Step 2: Table creation (if not exists)

```
# Create the 'users' table if it doesn't exist
cursor.execute('''
    CREATE TABLE IF NOT EXISTS users (
        id INTEGER PRIMARY KEY,
        name TEXT NOT NULL,
        age INTEGER
    );
''')
# Commit the changes to persist the table creation
```

```

connection.commit()

Step 3: Record creation (Create)

# Insert a new user record

cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", ('John Doe', 25))

# Commit the changes to persist the new record

connection.commit()

```

#### **Step 4: Record retrieval (Read)**

```

# Retrieve all users from the 'users' table

cursor.execute("SELECT * FROM users")

users = cursor.fetchall()

# Print the retrieved users

for user in users:

    print(user)

```

#### **Step 5: Record updating (Update)**

```

# Update the age of the user with id=1

cursor.execute("UPDATE users SET age=? WHERE id=?", (30, 1))

# Commit the changes to persist the update

connection.commit()

```

#### **Step 6: Record deletion (Delete)**

```

# Delete the user with id=1

cursor.execute("DELETE FROM users WHERE id=?", (1,))

# Commit the changes to persist the deletion

connection.commit()

Step 7: Closing Cursor and Connection

# Close the cursor and the database connection

cursor.close()

connection.close()

```

This project demonstrates CRUD operations:

- Create: Adding a new user to the 'users' table.
- Read: Retrieving all users from the 'users' table.
- Update: Modifying the age of a specific user.
- Delete: Removing a user from the 'users' table.

These operations can be extended and integrated into larger projects, showcasing the versatility and power of Python in handling database interactions.

#### **Conclusion**

A comprehensive exploration of Networking and Serialization, Data Processing and Analysis, and Database Applications is undertaken. Networking and serialization lay the groundwork, establishing communication foundations, and addressing efficient data handling. This knowledge assumes pivotal importance in the transition to Data Processing and Analysis, where NumPy and Pandas leverage networking principles for effective data manipulation.

The insights gained in data analysis, in turn, prove crucial for mastering Database Applications, influencing decisions on database connectivity, SQL queries, and the development of robust database-driven applications. This interplay ensures a cohesive learning experience, where each topic contributes to a comprehensive skill set in Python programming.



IBM ICE (Innovation Centre for Education)

## Self evaluation: Exercise 30

- **Exercise 30:** Client-Server Communication
- **Estimated time:** 00:15 minutes
- **Aim:** Create a simple client-server program using socket programming in Python. Implement basic communication between the client and server.
- **Learning objective:**
  - This exercise aims to introduce participants to client-server communication using socket programming in Python. Learners will understand the fundamental concepts of establishing connections, sending data, and receiving responses between a client and a server
- **Learning outcome:**
  - Upon completing this exercise, participants will be able to implement basic client-server communication in Python. They will gain insights into socket programming and be equipped to build more advanced networked applications

---

Figure 7-39. Self evaluation: Exercise 30

Self evaluation exercise 30 is as stated above:



IBM ICE (Innovation Centre for Education)

## Self evaluation: Exercise 31

- **Exercise 31:** Enhanced Client-Server Communication
- **Estimated time:** 00:15 minutes
- **Aim:** Extend the client-server program to support data transfer between the client and server. Implement sending and receiving data between the two
- **Learning objective:**
  - This exercise aims to build upon the basic client-server communication model by enhancing it to support bidirectional data transfer. Participants will learn how to send and receive data between the client and server, enabling more interactive communication
- **Learning outcome:**
  - Upon completing this exercise, participants will be able to enhance a simple client-server program to facilitate the exchange of data between the two entities. Learners will gain practical insights into building more interactive networked applications

---

Figure 7-40. Self evaluation: Exercise 31

Self evaluation exercise 31 is as stated above:

## Self evaluation: Exercise 32



IBM ICE (Innovation Centre for Education)

- **Exercise 32:** JSON Serialization and Deserialization
- **Estimated time:** 00:15 minutes
- **Aim:** Serialize Python objects into JSON format and then deserialize them back into Python objects. Demonstrate data interchange between JSON and Python.
- **Learning objective:**
  - This exercise aims to familiarize participants with the process of serializing and deserializing Python objects using JSON (JavaScript Object Notation). Participants will learn how to convert Python data structures into a JSON format and vice versa for effective data interchange
- **Learning outcome:**
  - Upon completing this exercise, participants will be able to apply JSON serialization and deserialization techniques to facilitate communication between Python programs and other systems that support JSON data interchange

---

Figure 7-41. Self evaluation: Exercise 32

Self evaluation exercise 32 is as stated above:



IBM ICE (Innovation Centre for Education)

## Self evaluation: Exercise 33

- **Exercise 33:** Pickle Serialization
- **Estimated time:** 00:15 minutes
- **Aim:** Use Pickle to serialize and deserialize Python objects. Save Python objects to a file using Pickle and then load them back
- **Learning objective:**
  - This exercise aims to introduce participants to Pickle, a module in Python that provides a convenient way to serialize and deserialize Python objects. Participants will learn how to use Pickle to store and retrieve Python objects, enabling data persistence
- **Learning outcome:**
  - Upon completing this exercise, participants will be able to apply Pickle for serializing complex Python objects, storing them in a file, and later retrieving them. This skill is particularly useful for saving and loading data structures between different program runs

---

Figure 7-42. Self evaluation: Exercise 33

Self evaluation exercise 33 is as stated above:



IBM ICE (Innovation Centre for Education)

## Self evaluation: Exercise 34

- **Exercise 34:** Networked Data Exchange
- **Estimated time:** 00:15 minutes
- **Aim:** Develop a Python script that simulates data exchange over a network using sockets and serialization. Send and receive data between client and server
- **Learning objective:**
  - This exercise aims to introduce participants to networked data exchange using sockets and serialization in Python. Participants will learn how to send and receive structured data between a client and server, facilitating more complex communication
- **Learning outcome:**
  - Upon completing this exercise, participants will be able to implement a basic client-server system that exchanges serialized data over a network. Learners will gain practical knowledge of sending and receiving structured information between different entities in a networked environment

---

Figure 7-43. Self evaluation: Exercise 34

Self evaluation exercise 34 is as stated above:



IBM ICE (Innovation Centre for Education)

## Self evaluation: Exercise 35

- **Exercise 35:** Basic NumPy Operations
- **Estimated time:** 00:15 minutes
- **Aim:** Create a NumPy array and perform basic operations like addition, subtraction, multiplication, and division.
- **Learning objective:**
  - This exercise aims to introduce participants to basic operations using NumPy arrays in Python. Participants will learn how to create arrays and perform fundamental mathematical operations efficiently using NumPy
- **Learning outcome:**
  - Create NumPy arrays
  - Perform addition, subtraction, multiplication, and division operations on NumPy arrays

---

Figure 7-44. Self evaluation: Exercise 35

Self evaluation exercise 35 is as stated above:

## Self evaluation: Exercise 36



IBM ICE (Innovation Centre for Education)

- Exercise 36: Data Filtering and Selection
- Estimated time: 00:15 minutes
- Aim: Explore data filtering and selection techniques using Pandas in Python
- **Learning objective:**
  - This exercise aims to familiarize participants with data filtering and selection methods in Pandas, allowing them to extract specific subsets of data from a DataFrame.
- **Learning outcome:**
  - Filter and select data based on specific conditions
  - Utilize Pandas functions for data filtering and selection

---

Figure 7-45. Self evaluation: Exercise 36

Self evaluation exercise 36 is as stated above:

## Self evaluation: Exercise 37



IBM ICE (Innovation Centre for Education)

- **Exercise 37:** Data Analysis with Pandas
- **Estimated time:** 00:15 minutes
- **Aim:** Use Pandas to read data from a CSV file and perform data analysis. Calculate statistics like mean, median, and standard deviation on the dataset.
- **Learning objective:**
  - This exercise aims to familiarize participants with the data analysis capabilities of Pandas. Participants will learn how to read data from a CSV file and perform basic statistical analysis on the dataset
- **Learning outcome:**
  - Load data from a CSV file into a Pandas DataFrame
  - Understand and apply basic statistical analysis functions in Pandas
  - Calculate measures like mean, median, and standard deviation

---

Figure 7-46. Self evaluation: Exercise 37

Self evaluation exercise 37 is as stated above:

## Self evaluation: Exercise 38



IBM ICE (Innovation Centre for Education)

- **Exercise 38:** DataFrame Manipulation
- **Estimated time:** 00:15 minutes
- **Aim:** Explore DataFrame manipulation in Pandas. Sort, filter, and perform various operations on a dataset loaded into a DataFrame
- **Learning objective:**
  - This exercise aims to introduce participants to DataFrame manipulation using the Pandas library. Participants will learn how to sort, filter, and perform various operations on data stored in a Pandas DataFrame
- **Learning outcome:**
  - Load data into a Pandas DataFrame
  - Understand and apply sorting techniques on DataFrame columns
  - Filter and subset data based on specific conditions
  - Perform basic operations such as adding new columns and aggregating data

---

Figure 7-47. Self evaluation: Exercise 38

Self evaluation exercise 38 is as stated above:

## Self evaluation: Exercise 39



IBM ICE (Innovation Centre for Education)

- **Exercise 39:** Database CRUD Operations
- **Estimated time:** 00:20 minutes
- **Aim:** Connect to a SQLite database in Python and perform CRUD (Create, Read, Update, Delete) operations using SQL queries. Create a Python script that interacts with a database.
- **Learning objective:**
  - This exercise aims to familiarize participants with the basics of interacting with a SQLite database in Python. Participants will learn how to establish a connection, execute SQL queries, and perform CRUD operations (Create, Read, Update, Delete) using Python's SQLite module
- **Learning outcome:**
  - Establish a connection to an SQLite database in Python
  - Execute SQL queries for creating tables, inserting data, fetching data, updating records, and deleting records
  - Understand the principles of CRUD operations in a database context

---

Figure 7-48. Self evaluation: Exercise 39

Self evaluation exercise 39 is as stated above:

## Checkpoint (1 of 2)



IBM ICE (Innovation Centre for Education)

### Multiple choice questions:

1. Which Pandas method is used to display the first few rows of a DataFrame?
  - a) `head()`
  - b) `tail()`
  - c) `sample()`
  - d) `describe()`
2. What is the correct SQL syntax to insert a new record into a table named "customers"?
  - a) `INSERT INTO customers (name, age) VALUES ('John', 35);`
  - b) `ADD RECORD TO customers (name, age) VALUES ('John', 35);`
  - c) `CREATE NEW ENTRY IN customers (name, age) VALUES ('John', 35);`
  - d) `UPDATE customers SET name = 'John', age = 35;`
3. You are performing data cleaning and exploration using Pandas in Python. Which Pandas method is commonly used to check for missing values in a DataFrame?
  - a) `detect_missing()`
  - b) `check_null()`
  - c) `isnull()`
  - d) `missing_data()`

Figure 7-49. Checkpoint (1 of 2)

Write your answers here:

- 1.
- 2.
- 3.



IBM ICE (Innovation Centre for Education)

## Checkpoint (2 of 2)

**Fill in the blanks:**

1. The two primary data structures in Pandas are \_\_\_\_\_ and \_\_\_\_\_.
2. To handle missing values in a DataFrame, you can use methods like \_\_\_\_\_ or \_\_\_\_\_.
3. To connect to a database in Python, you typically use a \_\_\_\_\_ library.
4. The SQL keyword used to retrieve data from a database table is \_\_\_\_\_.

**True/False:**

1. NumPy arrays are multi-dimensional matrices that offer efficient data storage and manipulation for numerical calculations. **True/False**
2. In socket programming, a server socket is responsible for initiating communication with the client. **True/False**
3. JSON and Pickle are both commonly used serialization formats in Python, but Pickle is recommended for transmitting data over a network due to its human-readable structure. **True/False**

---

Figure 7-50. Checkpoint (2 of 2)

Write your answers here:

Fill in the blanks:

- 1.
- 2.
- 3.
- 4.

True or false:

- 1.
- 2.
- 3.

# Question bank (1 of 2)



IBM ICE (Innovation Centre for Education)

## Two mark questions:

1. Compare and contrast the JSON and Pickle modules in Python for data serialization. Discuss their strengths, weaknesses, and appropriate use cases, considering factors like data format, security, and compatibility.
2. Discuss the methods used for data transfer and communication over sockets, including sending and receiving data, handling potential errors, and ensuring efficient data transfer.
3. Describe the steps involved in creating a client socket and a server socket in Python using the socket module.
4. What is the primary purpose of SQL (Structured Query Language)?

## Four mark questions:

1. You are given a dataset containing information about sales transactions. Using Pandas, load the dataset into a DataFrame and perform the following tasks:  
Display the first five rows of the DataFrame.  
Provide summary statistics (mean, median, standard deviation) for a numerical column in the dataset.  
Filter the dataset to include only rows where sales are above a specified threshold.  
Save the filtered dataset to a new CSV file.
2. Define the four basic CRUD operations in database applications. Provide Python code examples to illustrate how each operation can be performed using database connectivity libraries.

Figure 7-51. Question bank (1 of 2)



IBM ICE (Innovation Centre for Education)

## Question bank (2 of 2)

3. Describe the process of connecting to a database using Python. What are the essential steps involved, and what libraries are commonly used for this task?
4. Explain the concept of indexing and selection in Pandas DataFrames. Provide code examples to demonstrate how to:

Select a specific column by its name.

Access a single row by its index.

Select a subset of rows and columns based on conditions.

Filter data based on multiple criteria.

### Eight mark questions:

1. You have a dataset containing information about student grades. Use Pandas to:  
Load the dataset into a DataFrame.  
Add a new column to calculate the average grade for each student.  
Sort the DataFrame based on the average grade in descending order.  
Select the top 10 students with the highest average grades.
2. Explain the Pandas functions and methods and discuss the importance of these operations in data analysis.

---

Figure 7-52. Question bank (2 of 2)

## Unit summary



IBM ICE (Innovation Centre for Education)

- Having completed this unit, you should be able to:

- Able to Explore Python applications, including networking, serialization, and data processing with NumPy and Pandas, along with database applications

---

Figure 7-53. Unit summary

The unit summary are as stated above

---

# Unit 8. Advanced Cloud Topics and Case Studies

## Overview

This Unit will provide an overview of This unit will provide a comprehensive overview of solutions for all the lab exercises.

## How you will check your progress

- Checkpoint

## References

IBM Knowledge Center

## Unit objectives



IBM ICE (Innovation Centre for Education)

- After completing this unit, you should be able to:

---

*Figure 8-1. Unit objectives*

Unit objectives are as stated above.



IBM ICE (Innovation Centre for Education)

## Solution: Exercise 1

- Exercise 1: Hello, World! Program
- Estimated time: 00:10 minutes.
- Aim: Write a Python program that prints "Hello, World!" to the console.
- Learning objective:
  - The learner will understand the basic structure of a Python program and the usage of the print statement.
- Learning outcome:
  - The learner will be able to write and execute a simple Python program to display a message.

---

Figure 8-2. Solution: Exercise 1

### Methodology:

- Open a text editor or integrated development environment (IDE).
- Write the following code.
- Save the file with an appropriate name and the ".py" extension.
- Open a terminal or command prompt.
- Navigate to the directory where the file is saved.
- Run the script using the command python filename.py, replacing "filename" with the actual name of your file.

**Analysis:** This exercise introduces learners to the fundamental process of writing, saving, and executing a Python script. It emphasizes the basic syntax and structure of a Python program.

### Code Implementation:

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

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

Usage: Experiment\_1.1.py

Online

Colab

Notebook

Link

[https://colab.research.google.com/drive/1jTy\\_o5G-SOKB-V83Rz0W\\_ACx1o5xRua\\_?usp=sharing](https://colab.research.google.com/drive/1jTy_o5G-SOKB-V83Rz0W_ACx1o5xRua_?usp=sharing)

:



IBM ICE (Innovation Centre for Education)

## Solution: Exercise 2

- Exercise 2: Interactive Mode Basics
- Estimated time: 00:10 minutes
- Aim: Use Python's interactive mode to perform basic arithmetic operations like addition, subtraction, multiplication, and division.
- Learning objective:
  - The learner will understand how to use Python's interactive mode for basic arithmetic operations.
- Learning outcome:
  - The learner will be able to perform addition, subtraction, multiplication, and division using Python's interactive mode.

*Figure 8-3. Solution: Exercise 2*

### Methodology:

- Open a terminal or command prompt.
- Type python to enter Python's interactive mode.
- Perform basic arithmetic operations using the interactive mode prompt.

**Analysis:** This exercise aims to familiarize learners with Python's interactive mode, allowing them to perform quick calculations and understand the immediate feedback provided by the interpreter.

### Code Implementation:

```
# Code Implementation for Exercise 1.2: Interactive Mode Basics
# Perform basic arithmetic operations in Python's interactive mode prompt
# Addition
2 + 3
# Subtraction
5 - 1
# Multiplication
4 * 6
```

# Division

8 / 2

```
[root@learnvm lab_exercises]# python3
Python 3.9.18 (main, Sep 7 2023, 00:00:00)
[GCC 11.4.1 20230605 (Red Hat 11.4.1-2)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 2 + 3
5
>>> 5 - 1
4
>>> 4 * 6
24
>>> 8 / 2
4.0
>>> █
```

Usage: Experiment\_1.2.py



IBM ICE (Innovation Centre for Education)

## Solution: Exercise 3

- Exercise 3: User Input and Display
- Estimated time: 00:15 minutes
- Aim: Create a Python script that takes user input for their name and displays a personalized greeting.
- Learning objective:
  - The learner will understand how to use Python to take user input and display output.
- Learning outcome:
  - The learner will be able to write a Python script that interacts with the user, takes input, and provides personalized output.

*Figure 8-4. Solution: Exercise 3*

### **Methodology:**

- Open a text editor or integrated development environment (IDE).
- Write a Python script that prompts the user to enter their name using the `input()` function.
- Store the entered name in a variable.
- Use the `print()` function to display a personalized greeting that includes the user's name.

### **Analysis:**

This exercise introduces learners to user interaction in Python, allowing them to create scripts that dynamically respond to user input.

### **Code Implementation:**

```
# Prompt user for their name
user_name = input("Enter your name: ")

# Display personalized greeting
print(f"Hello, {user_name}! Welcome to the Python world.")

Usage: Experiment_1.3.py
```

```
[root@learnvm lab_exercises]# python3 exercise_1_3.py
Enter your name: abc
Hello, abc! Welcome to the Python world.
```



## Solution: Exercise 4

- Exercise 4: Calculate Rectangle And Circle Area
- Estimated time: 00:10 minutes
- Aim: Write a Python program that calculates and prints the area of a rectangle And Circle. Prompt the user for the length and width And Radius Respectivley.
- Learning objective:
  - The learner will understand how to write a Python program to calculate the area of a rectangle And Circle.
- Learning outcome:
  - The learner will be able to write a Python program that takes user input for the length and width of a rectangle, Radius Of The Circle And Calculates the area, and prints the result.
  - Here we can add circle area also. As circle area has power in the formula ( $3.14 * r^2$ ) we can add learning of `**` operator.

Figure 8-5. Solution: Exercise 4

### Methodology:

- Open a text editor or integrated development environment (IDE).
- Write a Python program that uses the `input()` function to prompt the user for the length and width of a rectangle.
- Convert the user input to numerical values (float or int).
- Calculate the area of the rectangle using the formula: `area = length * width`.
- Print the calculated area

### Analysis:

This exercise focuses on applying basic mathematical operations in Python to solve a real-world problem, encouraging learners to interact with the program by providing input.

### Code Implementation:

```
# Prompt user for the length and width
length = float(input("Enter the length of the rectangle: "))
width = float(input("Enter the width of the rectangle: "))
# Calculate the area
area = length * width
```

```
# Display the calculated area
print(f"The area of the rectangle is: {area}")
Code Implementation With Circle
# Prompt user for the dimensions of rectangle
length = float(input("Enter the length of the rectangle: "))
width = float(input("Enter the width of the rectangle: "))
# Prompt user for the radius of the circle
radius = float(input("Enter the radius of the circle: "))
# Calculate the area of rectangle
rectangle_area = length * width
# Calculate the area of circle
circle_area = 3.14 * radius * radius
# Display the calculated areas
print(f"The area of the rectangle is: {rectangle_area}")
print(f"The area of the circle is: {circle_area}")
```

```
[root@learnvm lab_exercises]# python3 exercise_1_4.py
Enter the length of the rectangle: 2
Enter the width of the rectangle: 4
The area of the rectangle is: 8.0
[root@learnvm lab_exercises]#
[root@learnvm lab_exercises]# python3 exercise_1_4.py
Enter the length of the rectangle: 1.1
Enter the width of the rectangle: 2.3
The area of the rectangle is: 2.53
```

```
Enter the length of the rectangle: 5
Enter the width of the rectangle: 4
Enter the radius of the circle: 3
The area of the rectangle is: 20.0
The area of the circle is: 28.26
```



## Solution: Exercise 5

- Exercise 5: Temperature conversion
- Estimated time: 00:10 minutes
- Aim: Write a Python program that converts a temperature from Fahrenheit to Celsius. Prompt the user for the temperature in Fahrenheit and display the result in Celsius upto 2 Decimals.
- Learning objective:
  - The learner will understand how to write a Python program for temperature conversion.
- Learning outcome:
  - The learner will be able to write a Python program that takes user input for temperature in Fahrenheit, converts it to Celsius, and prints the result.
  - Here, we should mention that print result up to 2 decimals.

*Figure 8-6. Solution: Exercise 5*

### Methodology:

- Open a text editor or integrated development environment (IDE).
- Write a Python program that uses the `input()` function to prompt the user for the temperature in Fahrenheit.
- Convert the user input to a numerical value (`float`).
- Use the conversion formula:  $\text{Celsius} = (\text{Fahrenheit} - 32) * 5/9$  to calculate the temperature in Celsius.
- Print the converted temperature in Celsius.

### Analysis:

This exercise introduces learners to basic temperature conversion using a simple Python program, emphasizing the application of mathematical formulas.

### Code Implementation:

```
# Prompt user for temperature in Fahrenheit
fahrenheit = float(input("Enter the temperature in Fahrenheit: "))

# Convert Fahrenheit to Celsius
celsius = (fahrenheit - 32) * 5/9

# Display the converted temperature in Celsius
print(f"The temperature in Celsius is: {celsius:.2f}")
```

Make sure we have learning of how f-strings works.

```
[root@learnvm lab_exercises]# python3 exercise_1_5.py
Enter the temperature in Fahrenheit: 100
The temperature in Celsius is: 37.78
[root@learnvm lab_exercises]# python3 exercise_1_5.py
Enter the temperature in Fahrenheit: 90
The temperature in Celsius is: 32.22
```



IBM ICE (Innovation Centre for Education)

## Solution: Exercise 6

- Exercise 6: Even or Odd Checker
- Estimated time: 00:10 minutes
- Aim: Implement a Python program that checks if a given number is even or odd and prints the result.
- Learning objective:
  - The learner will understand how to implement a Python program to determine if a number is even or odd.
- Learning outcome:
  - The learner will be able to write a Python program that checks the parity of a given number and prints whether it's even or odd.

*Figure 8-7. Solution: Exercise 6*

### Methodology:

- Open a text editor or integrated development environment (IDE).
- Write a Python program that uses the `input()` function to prompt the user for a number.
- Convert the user input to a numerical value (`int`).
- Check if the number is even or odd using the modulo operator (`%`).
- Print the result indicating whether the number is even or odd.

### Analysis:

This exercise aims to reinforce the concept of conditional statements and the use of the modulo operator to determine the parity of a number.

### Code Implementation:

```
# Prompt user for a positive integer
number = input("Enter a positive integer: ")

# Check if the input is a positive integer
if number.isdigit():

    number = int(number)

    # Check if the number is even or odd
```

```
if number > 0:  
    if number % 2 == 0:  
        print(f"{number} is an even number.")  
    else:  
        print(f"{number} is an odd number.")  
else:  
    print("Please enter a positive integer.")  
else:  
    print("Invalid input. Please enter a positive integer.")
```

**Output:**

```
[root@learnvm lab_exercises]# python3 exercise_2_1.py  
Enter a number: 2  
2 is an even number.  
[root@learnvm lab_exercises]# python3 exercise_2_1.py  
Enter a number: 21  
21 is an odd number.
```

```
Enter a positive integer: -10  
Please enter a positive integer.
```



IBM ICE (Innovation Centre for Education)

## Solution: Exercise 7

- Exercise 7: Largest among Three Numbers
- Estimated time: 00:10 minutes
- Aim: Implement a Python program that finds and prints the largest among three given numbers.
- Learning objective:
  - The learner will understand how to implement a Python program to determine the largest among three numbers.
- Learning outcome:
  - The learner will be able to write a Python program that compares three numbers and prints the largest one.

*Figure 8-8. Solution: Exercise 7*

### Methodology:

- Open a text editor or integrated development environment (IDE).
- Write a Python program that uses the `input()` function to prompt the user for three numbers.
- Convert the user inputs to numerical values (`int`).
- Use conditional statements to compare the three numbers and find the largest one.
- Print the result indicating the largest number.

### Analysis:

This exercise focuses on enhancing the learner's skills in implementing conditional statements and comparisons in Python.

### Code Implementation:

```
# Prompt user for three numbers
num1 = int(input("Enter the first number: "))
num2 = int(input("Enter the second number: "))
num3 = int(input("Enter the third number: "))

# Find the largest number using conditional statements
if num1 >= num2 and num1 >= num3:
```

```
largest = num1
elif num2 >= num1 and num2 >= num3:
    largest = num2
else:
    largest = num3
# Print the result
print(f"The largest number among {num1}, {num2}, and {num3} is: {largest}")
```

```
Enter the first number: 10
Enter the second number: -23
Enter the third number: 0
The largest number among 10, -23, and 0 is: 10
> |
```



IBM ICE (Innovation Centre for Education)

## Solution: Exercise 8

- Exercise 8: Factorial Calculation with a While Loop
- Estimated time: 00:15 minutes
- Aim: Create a Python program that calculates and prints the factorial of a number entered by the user using a while loop.
- Learning objective:
  - The learner will understand how to implement a Python program to calculate the factorial of a number using a while loop.
- Learning outcome:
  - The learner will be able to write a Python program that utilizes a while loop to calculate the factorial of a user-entered number.
  - We can add sum of n numbers exercise before factorial calculation. It will be easy to built logic of factorial calculation after solving sum of n numbers problem.

Figure 8-9. Solution: Exercise 8

### Methodology:

- Open a text editor or integrated development environment (IDE).
- Write a Python program that uses the `input()` function to prompt the user for a number.
- Convert the user input to a numerical value (`int`).
- Initialize variables for the factorial (`fact`) and a counter (`counter`).
- Use a while loop to iteratively calculate the factorial.
- Print the result.

### Analysis:

This program has a time complexity of  $O(n)$  and a space complexity of  $O(1)$ . The time complexity is linear because the while loop iterates  $n$  times, where  $n$  is the user-entered number. The space complexity is constant, as the program uses a fixed amount of memory regardless of the input.

### Code Implementation:

```
# Prompt user for a number
num = int(input("Enter a number: "))

# Initialize variables
fact = 1
counter = 1
```

```
# Calculate factorial using a while loop
while counter <= num:
    fact *= counter
    counter += 1
# Print the result
print(f"The factorial of {num} is: {fact}")
```

```
[root@learnvm lab_exercises]# python3 exercise_2_3.py
Enter a number: 5
The factorial of 5 is: 120
```



## Solution: Exercise 9

- Exercise 9: Fibonacci Sequence with For Loop
- Estimated time: 00:15 minutes
- Aim: Use a for loop to generate and print the Fibonacci sequence up to a specified number of terms. Prompt the user for the number of terms.
- Learning objective:
  - The learner will understand how to implement a Python program to generate the Fibonacci sequence using a for loop.
- Learning outcome:
  - The learner will be able to write a Python program that utilizes a for loop to generate and print the Fibonacci sequence based on the user-specified number of terms.
  - We can add example of Fibonacci sequence.(addition of First 2 element as third element)

Figure 8-10. Solution: Exercise 9

### Methodology:

- Open a text editor or integrated development environment (IDE).
- Write a Python program that uses the `input()` function to prompt the user for the number of terms in the Fibonacci sequence.
- Convert the user input to a numerical value (`int`).
- Initialize variables for the first two terms of the sequence (first and second).
- Use a for loop to iteratively generate and print the Fibonacci sequence.
- Print the result.

### Analysis:

This program has a time complexity of  $O(n)$  and a space complexity of  $O(1)$ . The time complexity is linear because the for loop iterates  $n$  times, where  $n$  is the user-specified number of terms. The space complexity is constant, as the program uses a fixed amount of memory regardless of the input.

### Code Implementation:

```
# Prompt user for the number of terms
num_terms = int(input("Enter the number of terms for the Fibonacci sequence: "))

# Initialize variables
first, second = 0, 1
```

```
# Generate and print Fibonacci sequence using a for loop
for _ in range(num_terms):
    print(first, end=" ")
    first, second = second, first + second
# Print a newline for better formatting
print()

Example 2: Fibonacci sequence.(addition of First 2 element as third element)
# Prompt user for the length of Fibonacci sequence
length = int(input("Enter the length of the Fibonacci sequence: "))
# Check if the input length is valid
if length <= 0:
    print("Please enter a positive integer for the length of the Fibonacci
sequence.")
else:
    # Initialize Fibonacci sequence with first two elements
    fibonacci_sequence = [0, 1]
    # Generate Fibonacci sequence
    for i in range(2, length):
        next_element = fibonacci_sequence[-1] + fibonacci_sequence[-2]
        fibonacci_sequence.append(next_element)
    # Display Fibonacci sequence
    print("The Fibonacci sequence is:")
    print(fibonacci_sequence)
```

```
[root@learnvm lab_exercises]# python3 exercise_2_4.py
Enter the number of terms for the Fibonacci sequence: 5
0 1 1 2 3
```



IBM ICE (Innovation Centre for Education)

## Solution: Exercise 10

- Exercise 10: Sum of Prime Numbers in a Given Range
- Estimated time: 00:15 minutes
- Aim: Write a Python program that calculates and prints the sum of all prime numbers in a given range. Prompt the user for the range.
- Learning objective:
  - The learner will understand how to implement a Python program to calculate the sum of prime numbers in a specified range.
- Learning outcome:
  - The learner will be able to write a Python program that utilizes loops, conditionals, and user input to find and sum prime numbers within a given range.

*Figure 8-11. Solution: Exercise 10*

### Methodology:

- Open a text editor or integrated development environment (IDE).
- Write a Python program that uses the `input()` function to prompt the user for the range (start and end values).
- Convert the user input to numerical values (`int`).
- Implement a function to check whether a given number is prime.
- Use a loop to iterate through the numbers in the specified range.
- For each number, check if it is prime using the function from step 4.
- If the number is prime, add it to the sum.
- Print the sum of prime numbers.

### Analysis:

The program has a time complexity of  $O(n * \sqrt{m})$ , where  $n$  is the size of the range and  $m$  is the current number being checked for primality. The space complexity is  $O(1)$  as the program uses a fixed amount of memory.

### Code Implementation:

```
# Code Implementation for Exercise XXX: Sum of Prime Numbers in a Given Range
# Calculate and print the sum of all prime numbers in a given range. Prompt the user for the range.
```

```
def is_prime(num):
    if num < 2:
        return False
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True

# Prompt user for the range
start = int(input("Enter the start of the range: "))
end = int(input("Enter the end of the range: "))
# Initialize sum of prime numbers
sum_primes = 0
# Find and sum prime numbers in the given range
for num in range(start, end + 1):
    if is_prime(num):
        sum_primes += num
# Print the sum of prime numbers
print(f"The sum of prime numbers in the range ({start}, {end}) is: {sum_primes}")
```

```
[root@learnvm lab_exercises]# python3 exercise_2_5.py
Enter the start of the range: 100
Enter the end of the range: 200
The sum of prime numbers in the range (100, 200) is: 3167
```



## Solution: Exercise 11

- Exercise 11: Simple Calculator Class
- Estimated time: 00:15 minutes
- Aim: Define a Python class for a simple calculator that has methods for addition and subtraction. Allow the user to perform calculations using objects of this class.
- Learning objective:
  - The learner will understand how to create and use a simple class in Python for basic arithmetic operations.
- Learning outcome:
  - The learner will be able to define a class, create objects, and perform addition and subtraction operations using methods.

*Figure 8-12. Solution: Exercise 11*

### Methodology:

- Open a text editor or integrated development environment (IDE).
- Define a class named `SimpleCalculator` with methods for addition and subtraction , Multiplication And Division.
- Create an instance of the `SimpleCalculator` class.
- Use the class methods to perform addition and subtraction operations.
- Print the results.

### Analysis:

This exercise introduces learners to creating a simple class in Python. The class `SimpleCalculator` encapsulates basic arithmetic operations. The time complexity of the class methods is constant, O(1), as they involve straightforward mathematical operations. The space complexity is also constant, O(1), as no additional data structures are used. Learners will understand the concept of encapsulation and how to use class methods for calculations.

### Code Implementation:

```
# Simple Calculator Class
class SimpleCalculator:
    def add(self, num1, num2):
```

```
        return num1 + num2
def subtract(self, num1, num2):
    return num1 - num2
def multiply(self, num1, num2):
    return num1 * num2
def divide(self, num1, num2):
    if num2 == 0:
        return "Error! Division by zero is not allowed."
    else:
        return num1 / num2
# Create an instance of SimpleCalculator
calculator = SimpleCalculator()
# Perform arithmetic operations
result_addition = calculator.add(10, 5)
result_subtraction = calculator.subtract(10, 5)
result_multiplication = calculator.multiply(10, 5)
result_division = calculator.divide(10, 5)
# Print the results
print("Result of Addition:", result_addition)
print("Result of Subtraction:", result_subtraction)
print("Result of Multiplication:", result_multiplication)
print("Result of Division:", result_division)
Here we can add some more operation like Multiplication , Division , Percentage.
```

```
Result of Addition: 15
Result of Subtraction: 5
Result of Multiplication: 50
Result of Division: 2.0

== Code Execution Successful ==
```



## Solution: Exercise 12

- Exercise 12: Class Inheritance Hierarchy
- Estimated time: 00:15 minutes
- Aim: Create a class hierarchy with a base class and two derived classes. Demonstrate inheritance by accessing attributes and methods of each class.
- Learning objective:
  - The learner will understand the concept of class inheritance in Python and how to create a hierarchy of classes.
- Learning outcome:
  - The learner will be able to define a base class, create derived classes, and demonstrate inheritance by accessing attributes and methods.

*Figure 8-13. Solution: Exercise 12*

### Methodology:

- Open a text editor or integrated development environment (IDE).
- Define a base class named `Shape` with attributes and methods related to shapes.
- Create two derived classes, e.g., `Circle` and `Rectangle`, inheriting from the base class `Shape`.
- Instantiate objects of the derived classes and demonstrate access to attributes and methods of both base and derived classes.

### Analysis:

This exercise introduces learners to class inheritance, a fundamental concept in object-oriented programming. The base class `Shape` represents common attributes and methods, while the derived classes specialize in specific shapes. Learners will understand how to create and use an inheritance hierarchy in Python. The time and space complexity depend on the specific methods and attributes implemented.

### Code Implementation:

```
# Class Inheritance Hierarchy
class Shape:
    def __init__(self, color):
        self.color = color
    def display_color(self):
```

```
    print(f"Color: {self.color}")  
class Circle(Shape):  
    def __init__(self, color, radius):  
        super().__init__(color)  
        self.radius = radius  
    def display_radius(self):  
        print(f"Radius: {self.radius}")  
class Rectangle(Shape):  
    def __init__(self, color, width, height):  
        super().__init__(color)  
        self.width = width  
        self.height = height  
    def display_dimensions(self):  
        print(f"Width: {self.width}, Height: {self.height}")  
# Instantiate objects  
circle = Circle("Red", 5)  
rectangle = Rectangle("Blue", 8, 10)  
# Display information  
circle.display_color()  
circle.display_radius()  
rectangle.display_color()  
rectangle.display_dimensions()
```

```
[root@learnvm lab_exercises]# python3 exercise_3_2.py  
Color: Red  
Radius: 5  
Color: Blue  
Width: 8, Height: 10
```



## Solution: Exercise 13

- Exercise 13: Method Overriding
- Estimated time: 00:15 minutes
- Aim: Implement method overriding in a Python class. Create a base class with a method and override it in a derived class.
- Learning objective:
  - The learner will understand the concept of method overriding in Python and how to implement it in a class hierarchy.
- Learning outcome:
  - The learner will be able to define a base class with a method, create a derived class, and override the method to provide a specialized implementation.

Figure 8-14. Solution: Exercise 13

### Methodology:

- Open a text editor or integrated development environment (IDE).
- Define a base class named `Vehicle` with a method `start\_engine`.
- Create a derived class `Car` that inherits from the base class and overrides the `start\_engine` method to provide a car-specific implementation.
- Instantiate objects of both the base and derived classes and demonstrate method overriding.

### Analysis:

This exercise introduces learners to method overriding, a crucial concept in object-oriented programming. The base class `Vehicle` represents a generic method, while the derived class `Car` overrides the method to provide a specialized behavior for cars. Learners will understand how to apply method overriding to extend and customize class behavior. The time and space complexity depend on the specific methods and attributes implemented.

### Code Implementation:

```
# Method Overriding Example
class Vehicle:
    def start_engine(self):
        print("Generic engine start")
```

```
class Car(Vehicle):
    def start_engine(self):
        print("Car engine start")
# Instantiate objects
vehicle = Vehicle()
car = Car()
# Demonstrate method overriding
print("Vehicle:")
vehicle.start_engine()
print("\nCar:")
car.start_engine()
```

```
[root@learnvm lab_exercises]# python3 exercise_3_3.py
Vehicle:
Generic engine start

Car:
Car engine start
```



## Solution: Exercise 14

- Exercise 14: Encapsulation Demonstration
- Estimated time: 00:15 minutes
- Aim: Use encapsulation to restrict access to class attributes. Create a class with private attributes and demonstrate encapsulation principles.
- Learning objective:
  - The learner will understand the concept of encapsulation in Python and how to use it to control access to class attributes.
- Learning outcome:
  - The learner will be able to define a class with private attributes, implement getter and setter methods, and demonstrate the benefits of encapsulation.

Figure 8-15. Solution: Exercise 14

### **Methodology:**

Open a text editor or integrated development environment (IDE).

Define a class named `Person` with private attributes like `\_\_name` and `\_\_age`.

Implement getter and setter methods to access and modify the private attributes.

Instantiate objects of the class and demonstrate encapsulation by accessing attributes through methods.

### **Analysis:**

Encapsulation is a fundamental concept in object-oriented programming that involves restricting access to certain components of an object. In this exercise, learners will explore encapsulation by creating a class with private attributes and using getter and setter methods. The analysis will focus on the advantages of encapsulation, such as improved data integrity and security.

### **Code Implementation:**

```
# Encapsulation Demonstration
```

```
class Person:
    def __init__(self, name, age):
        self.__name = name # Private attribute
        self.__age = age   # Private attribute
    # Getter method for name
```

```
def get_name(self):
    return self._name
# Getter method for age
def get_age(self):
    return self._age
# Setter method for age
def set_age(self, new_age):
    if 0 < new_age < 150:
        self._age = new_age
    else:
        print("Invalid age value")
# Instantiate object
person = Person("John Doe", 25)
# Demonstrate encapsulation
print("Name:", person.get_name())
print("Age:", person.get_age())
# Try to set an invalid age
person.set_age(200)
```

```
[root@learnvm lab_exercises]# python3 exercise_3_4.py
Name: John Doe
Age: 25
Invalid age value
```



## Solution: Exercise 15

- Exercise 15: Abstract Geometric Shape Class
- Estimated time: 00:15 minutes
- Aim: Create an abstract class representing a geometric shape with abstract methods like area and perimeter. Define derived classes (e.g., Circle, Rectangle) to implement these methods.
- Learning objective:
  - The learner will understand the concept of abstract classes in Python and how to use them to define a common interface for related classes.
- Learning outcome:
  - The learner will be able to create an abstract class for geometric shapes, define abstract methods for common operations, and implement those methods in derived classes.

*Figure 8-16. Solution: Exercise 15*

### Methodology:

- Open a text editor or integrated development environment (IDE).
- Define an abstract class named `GeometricShape` with abstract methods like `area` and `perimeter`.
- Create derived classes (e.g., `Circle`, `Rectangle`) that inherit from the abstract class and implement the abstract methods.
- Instantiate objects of the derived classes and demonstrate the use of common methods.

### Analysis:

Abstract classes provide a way to define a common interface for a group of related classes, ensuring that each derived class implements certain methods. In this exercise, learners will explore the concept of abstract classes in the context of geometric shapes, understanding how to create a base class with abstract methods and implement those methods in derived classes.

### Code Implementation:

```
from abc import ABC, abstractmethod
import math

# Abstract Geometric Shape Class
class GeometricShape(ABC):
    @abstractmethod
```

```
def area(self):  
    pass  
  
@abstractmethod  
def perimeter(self):  
    pass  
  
# Derived Class: Circle  
  
class Circle(GeometricShape):  
    def __init__(self, radius):  
        self.radius = radius  
  
    def area(self):  
        return math.pi * self.radius**2
```

```
[root@learnvm lab_exercises]# python3 exercise_3_5.py  
Circle Area: 78.53981633974483  
Circle Perimeter: 31.41592653589793  
Rectangle Area: 24  
Rectangle Perimeter: 20
```



## Solution: Exercise 16

- Exercise 16: Custom Exception Handling.
- Estimated time: 00:15 minutes.
- Aim: Write a Python program that raises and handles a custom exception. Define a custom exception class and demonstrate its usage.
- Learning objective:
  - The learner will understand how to create and handle custom exceptions in Python, enhancing the ability to manage errors in code.
- Learning outcome:
  - The learner will be able to define a custom exception class, raise the exception in specific situations, and handle it gracefully within the program.

Figure 8-17. Solution: Exercise 16

### Methodology:

- Open a text editor or integrated development environment (IDE).
- Define a custom exception class (e.g., `CustomException`) by inheriting from the base `Exception` class.
- Write a Python program that raises the custom exception in response to a certain condition.
- Implement exception handling using `try` and `except` blocks to gracefully manage the custom exception.

### Analysis:

Custom exceptions provide a way to handle specific errors in a more structured manner, making code more robust. In this exercise, learners will explore the creation and usage of custom exceptions, gaining insights into effective error handling strategies.

### Code Implementation:

```
# Custom Exception Class
class CustomException(Exception):
    def __init__(self, message="Custom Exception"):
        self.message = message
        super().__init__(self.message)
# Python Program with Custom Exception Handling
```

```
def example_function(value):
    if value < 0:
        raise CustomException("Negative value is not allowed")
# Example of using custom exception
try:
    user_input = int(input("Enter a non-negative number: "))
    example_function(user_input)
    print("No custom exception raised.")
except CustomException as ce:
    print(f"Custom Exception Caught: {ce}")
except ValueError:
    print("Invalid input. Please enter a valid integer.")
```

```
[root@learnvm lab_exercises]# python3 exercise_4_1.py
Enter a non-negative number: 100
No custom exception raised.
```

```
Enter a non-negative number: -10
Custom Exception Caught: Negative value is not allowed
> |
```



## Solution: Exercise 17

- Exercise 17: File Exception Handling
- Estimated time: 00:15 minutes
- Aim: Implement a function that reads data from a file and handles file-related exceptions such as FileNotFoundError and PermissionError.
- Learning objective:
  - The learner will understand how to handle file-related exceptions in Python, ensuring robust file I/O operations in programs.
- Learning outcome:
  - The learner will be able to implement file exception handling mechanisms, making their programs more resilient to errors during file operations

Figure 8-18. Solution: Exercise 17

### Methodology:

- Open a text editor or integrated development environment (IDE).
- Define a function (e.g., `read\_file\_data`) that reads data from a file.
- Implement exception handling using `try`, `except`, and `finally` blocks to gracefully manage file-related exceptions such as FileNotFoundError and PermissionError.
- Within the function, attempt to open and read data from a file, handle exceptions, and finally close the file.

### Analysis:

File exception handling is crucial for ensuring that a program can gracefully recover from issues related to file access. In this exercise, learners will explore the implementation of file exception handling, improving the reliability of file operations in Python programs.

### Code Implementation:

```
# Function to read data from a file with exception handling
def read_file_data(file_path):
    try:
        with open(file_path, 'r') as file:
            data = file.read()
            print("File Data:", data)
    except FileNotFoundError:
        print("File not found")
    except PermissionError:
        print("Permission denied")
    except Exception as e:
        print(f"An error occurred: {e}")
```

```
except FileNotFoundError:  
    print(f"File not found: {file_path}")  
except PermissionError:  
    print(f"Permission error: Unable to read file {file_path}")  
finally:  
    print("File reading process completed.")  
  
# Example of using file exception handling function  
file_path = "sample.txt" # Update with your file path  
read_file_data(file_path)
```

```
[root@learnvm lab_exercises]# python3 exercise_4_2.py  
File not found: sample.txt  
File reading process completed.
```



## Solution: Exercise 18

- Exercise 18: Division by Zero Handling
- Estimated time: 00:10 minutes
- Aim: Implement a Python program that performs division and handles the division by zero exception.
- Learning objective:
  - The learner will understand how to handle division by zero exceptions in Python programs, ensuring robust arithmetic operations.
- Learning outcome:
  - The learner will be able to implement exception handling to gracefully manage division by zero scenarios in their Python programs.

Figure 8-19. Solution: Exercise 18

### Methodology:

- Open a text editor or integrated development environment (IDE).
- Write a Python program that includes a division operation.
- Implement exception handling using `try`, `except`, and `finally` blocks to manage the division by zero exception.
- Test the 1. program with different inputs, including scenarios where division by zero might occur.

### Analysis:

Handling division by zero is essential to prevent runtime errors and unexpected crashes in Python programs. This exercise focuses on implementing exception handling to gracefully manage division by zero scenarios, improving the overall reliability of arithmetic operations.

### Code Implementation:

```
# Python program to demonstrate division by zero handling
def perform_division(dividend, divisor):
    try:
        result = dividend / divisor
        print(f"Result of division: {result}")
    except ZeroDivisionError:
```

```
    print("Error: Division by zero is not allowed.")
finally:
    print("Division operation completed.")

# Example of using the division by zero handling function
perform_division(10, 2)  # Valid division
perform_division(5, 0)   # Division by zero scenario
```

```
[root@learnvm lab_exercises]# python3 exercise_4_3.py
Result of division: 5.0
Division operation completed.
Error: Division by zero is not allowed.
Division operation completed.
```



IBM ICE (Innovation Centre for Education)

## Solution: Exercise 19

- Exercise 19: File Handling with Multiple Exceptions
- Estimated time: 00:15 minutes
- Aim: Modify a file reading program to handle both FileNotFoundError and PermissionError exceptions.
- Learning objective:
  - The learner will understand how to handle multiple file-related exceptions in Python programs, enhancing the robustness of file operations.
- Learning outcome:
  - The learner will be able to implement exception handling for scenarios involving file reading, handling both FileNotFoundError and PermissionError exceptions.

*Figure 8-20. Solution: Exercise 19*

### Methodology:

- Open a text editor or integrated development environment (IDE).
- Modify an existing Python program that reads data from a file.
- Implement exception handling using `try`, `except`, and `finally` blocks to manage both FileNotFoundError and PermissionError exceptions.
- Test the program with different file scenarios, including missing files and permission issues.

### Analysis:

Handling multiple file-related exceptions is crucial for creating resilient file operations in Python programs. This exercise focuses on enhancing a file reading program by incorporating exception handling for both FileNotFoundError and PermissionError, ensuring a more robust file-handling mechanism.

### Code Implementation:

```
# Python program to demonstrate file handling with multiple exceptions
def read_file(file_path):
    try:
        with open(file_path, 'r') as file:
            content = file.read()
            print(f"File content:\n{content}")
    except FileNotFoundError:
        print("File not found")
    except PermissionError:
        print("Permission denied")
```

```

except FileNotFoundError:
    print("Error: File not found.")
except PermissionError:
    print("Error: Permission denied. Unable to read the file.")
finally:
    print("File handling completed.")

# Call the function and read a file
file_path = "example.txt" # Replace "example.txt" with the path to your file
read_file(file_path)

```

```

# Exercise 4.4: File Handling with Multiple Exceptions

def read_file(file_path):
    try:
        with open(file_path, 'r') as file:
            content = file.read()
            print(f"File content:\n{content}")
    except FileNotFoundError:
        print("Error: File not found.")
    except PermissionError:
        print("Error: Permission denied. Unable to read the file.")
    finally:
        print("File handling completed.")

# Call the function and read a file
file_path = "example.txt" # Replace "example.txt" with the path to your file
read_file(file_path)

```

Error: File not found.  
File handling completed.

- The `read_file` function takes a `file_path` as an argument.
- Inside the function, it attempts to open the file specified by `file_path` in read mode ('r') using a `with` statement. The `with` statement ensures that the file is properly closed after reading, even if an exception occurs.
- The `try` block contains the code that might raise exceptions. In this case, it tries to read the content of the file using `file.read()`.
- If the file is successfully opened and read, it prints the content of the file.
- If a `FileNotFoundException` occurs, meaning the specified file is not found, it prints an error message.
- If a `PermissionError` occurs, meaning there is a permission issue (e.g., the file is read-protected), it prints a different error message.
- The `finally` block contains code that will be executed no matter what, whether an exception is raised or not. In this case, it prints "File handling completed."

When you call the `read_file` function with a valid file path, it will print the file content if the file exists and is readable. If the file is not found or if there is a permission issue, it will catch the respective exceptions and print the corresponding error messages. In either case, the "File handling completed" message in the `finally` block will be printed.



IBM ICE (Innovation Centre for Education)

## Solution: Exercise 20

- Exercise 20: Finally Block Usage
- Estimated time: 00:15 minutes
- Aim: Develop a program that demonstrates the use of the finally block in exception handling. Open a file and ensure it is properly closed even if exceptions occur.
- Learning objective:
  - The learner will understand the importance of the finally block in exception handling and its role in ensuring resource cleanup, particularly file closure.
- Learning outcome:
  - The learner will be able to implement exception handling with the finally block to guarantee proper resource cleanup, such as closing files, even in the presence of exceptions.

*Figure 8-21. Solution: Exercise 20*

### Methodology:

- Open a text editor or integrated development environment (IDE).
- Write a Python program that attempts to open a file, read its content, and perform some operations.
- Implement exception handling using `try`, `except`, and `finally` blocks.
- Within the finally block, include code to ensure the file is closed, regardless of whether exceptions occurred or not.

### Analysis:

The finally block in exception handling is essential for ensuring cleanup operations, such as closing files or releasing resources. This exercise focuses on demonstrating the use of the finally block to guarantee proper file closure, promoting more robust and error-tolerant Python programs.

```
Usage: Experiment_4.5.py
def get_value_from_dict(dictionary, key):
    try:
        value = dictionary[key]
        print(f"The value corresponding to key '{key}' is:", value)
    except KeyError:
        print(f"Error: Key '{key}' not found in the dictionary.")
```

```
value = None
finally:
    print("Dictionary access attempt completed.")
    # Additional action within the finally block
    print("Performing cleanup or logging...")
return value

# Example usage
my_dict = {'a': 1, 'b': 2, 'c': 3}
key_to_find = 'd'
found_value = get_value_from_dict(my_dict, key_to_find)
if found_value is None:
    print(f"Key '{key_to_find}' not found in the dictionary.")
```

Output

```
ERROR!
Error: Key 'd' not found in the dictionary.
Dictionary access attempt completed.
Performing cleanup or logging...
Key 'd' not found in the dictionary.
```



IBM ICE (Innovation Centre for Education)

## Solution: Exercise 21

- Exercise 21: Email Validation with Regular Expressions
- Estimated time: 00:15 minutes
- Aim: Write a Python program that validates email addresses using regular expressions. Prompt the user for an email address and validate it.
- Learning objective:
  - The learner will understand the use of regular expressions for validating complex patterns, specifically focusing on email address validation.
- Learning outcome:
  - The learner will be able to implement email validation in Python using regular expressions, enhancing their understanding of pattern matching.

Figure 8-22. Solution: Exercise 21

### Methodology:

- Open a text editor or integrated development environment (IDE).
- Write a Python program that takes user input for an email address.
- Implement regular expressions to validate the email address pattern.
- Use the 're' module for regular expression operations in Python.
- Provide feedback to the user indicating whether the entered email address is valid or not.

### Analysis:

Regular expressions offer a powerful tool for pattern matching and validation. This exercise introduces learners to the application of regular expressions in validating email addresses. Understanding these patterns is essential for building robust data validation mechanisms.

### Code Implementation:

```
import re

def validate_email(email):
    # Regular expression for a basic email validation
    email_pattern = r'^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$'
    # Use re.match() to validate the email address
    if re.match(email_pattern, email):
```

```
    print(f"The email address '{email}' is valid.")  
else:  
    print(f"The email address '{email}' is invalid. Please enter a valid  
email.")  
  
# Prompt the user for input  
user_email = input("Enter an email address: ")  
  
# Validate the user-provided email  
validate_email(user_email)
```

```
[root@learnvm lab_exercises]# python3 exercise_5_1.py  
Enter an email address: abc@abc.com  
The email address 'abc@abc.com' is valid.  
[root@learnvm lab_exercises]# python3 exercise_5_1.py  
Enter an email address: abc.abc.com  
The email address 'abc.abc.com' is invalid. Please enter a valid email.
```



IBM ICE (Innovation Centre for Education)

## Solution: Exercise 22

- Exercise 22: Phone Number Extraction
- Estimated time: 00:15 minutes
- Aim: Create a Python script that extracts phone numbers from a given text using regular expressions.
- Learning objective:
  - The learner will gain proficiency in using regular expressions to extract specific patterns, focusing on phone number extraction in this exercise.
- Learning outcome:
  - The learner will be able to implement a Python script that utilizes regular expressions to extract phone numbers from a given text, enhancing their skills in pattern matching.

*Figure 8-23. Solution: Exercise 22*

### Methodology:

- Open a text editor or integrated development environment (IDE).
- Write a Python program that takes a text input containing phone numbers.
- Implement regular expressions to extract phone numbers from the provided text.
- Utilize the `re` module in Python for regular expression operations.
- Display the extracted phone numbers.

### Analysis:

Regular expressions are valuable tools for searching and extracting patterns from text data. This exercise introduces learners to the practical application of regular expressions in extracting phone numbers, a common task in data processing and analysis.

### Code Implementation:

```
import re

def extract_phone_numbers(text):
    # Regular expression pattern for extracting phone numbers
    phone_pattern = r'\b\d{3}[-.\s]?\d{3}[-.\s]?\d{4}\b'
    # Use re.findall() to extract phone numbers from the text
    extracted_numbers = re.findall(phone_pattern, text)
```

```
if extracted_numbers:  
    print("Extracted Phone Numbers:")  
    for number in extracted_numbers:  
        print(number)  
else:  
    print("No phone numbers found in the provided text.")  
  
# Example text containing phone numbers  
example_text = """  
Here are some contact numbers:  
John: 123-456-7890  
Alice: 987.654.3210  
Emergency: 911  
"""  
  
# Extract phone numbers from the example text  
extract_phone_numbers(example_text)
```

```
[root@learnvm lab_exercises]# python3 exercise_5_2.py  
Extracted Phone Numbers:  
123-456-7890  
987.654.3210
```



## Solution: Exercise 23

- Exercise 23: Custom Python Module
- Estimated time: 00:15 minutes.
- Aim: Develop a Python module with functions to perform basic arithmetic operations. Import this module into another script and use its functions.
- Learning objective:
  - The learner will acquire the skills to create and use custom Python modules, promoting code modularity and reusability.
- Learning outcome:
  - The learner will be able to design a Python module containing custom functions and successfully import and utilize these functions in another script.

Figure 8-24. Solution: Exercise 23

### Methodology:

- Open a text editor or integrated development environment (IDE).
- Create a Python module file (e.g., `math\_operations.py`) containing functions for basic arithmetic operations like addition, subtraction, multiplication, and division.
- Implement the module functions to perform the specified arithmetic operations.
- Save the module file.
- Create another Python script (e.g., `main\_script.py`) in the same directory.
- Import the custom module using `import math\_operations`.
- Utilize the functions from the imported module to perform arithmetic operations in the main script.

### Analysis:

This exercise emphasizes the importance of code organization and modularization by creating a custom Python module. Learners will gain insights into structuring code for reusability and ease of maintenance.

### Code Implementation:

```
math_operations.py
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):
    if y != 0:
        return x / y
    else:
        return "Cannot divide by zero"
main_script.py
# Import the custom module
import math_operations
# Perform arithmetic operations using functions from the imported module
result_add = math_operations.add(10, 5)
result_subtract = math_operations.subtract(10, 5)
result_multiply = math_operations.multiply(10, 5)
result_divide = math_operations.divide(10, 5)
# Display the results
print("Addition:", result_add)
print("Subtraction:", result_subtract)
print("Multiplication:", result_multiply)
print("Division:", result_divide)
```

Addition: 15  
Subtraction: 5  
Multiplication: 50  
Division: 2.0



IBM ICE (Innovation Centre for Education)

## Solution: Exercise 24

- Exercise 24: Exploring Built-in Modules
- Estimated time: 00:15 minutes
- Aim: Explore Python's built-in modules like math and datetime. Use them to perform mathematical operations and work with date and time.
- Learning Objective:
  - The learner will gain hands-on experience in utilizing Python's built-in modules, specifically math and datetime, to perform mathematical operations and work with date and time functionalities.
- Learning Outcome:
  - The learner will be proficient in leveraging Python's math module for mathematical computations and datetime module for managing date and time in their scripts.

*Figure 8-25. Solution: Exercise 24*

### Methodology:

- Open a text editor or integrated development environment (IDE).
- Create a Python script (e.g., `built\_in\_modules.py`) to explore the functionalities of the math and datetime modules.
- Import the math module using `import math`.
- Perform mathematical operations using the functions provided by the math module, such as square root, power, and trigonometric functions.
- Import the datetime module using `import datetime`.
- Utilize the datetime module to work with date and time objects, accessing attributes like year, month, day, hour, minute, and second.
- Display the results of mathematical operations and date/time manipulations in the script.

### Analysis:

This exercise aims to familiarize learners with Python's built-in modules, showcasing the versatility and utility of modules like math and datetime. By exploring these modules, learners will enhance their programming skills and understand the significance of leveraging existing tools.

### Code Implementation:

```
# Python script exploring built-in modules
```

```
import math
import datetime
# Math module exploration
sqrt_result = math.sqrt(25)
power_result = math.pow(2, 3)
sin_result = math.sin(math.radians(90))
# Datetime module exploration
current_datetime = datetime.datetime.now()
current_date = current_datetime.date()
current_time = current_datetime.time()
# Displaying results
print(f"Square root of 25: {sqrt_result}")
print(f"2 raised to the power of 3: {power_result}")
print(f"Sine of 90 degrees: {sin_result}")
print(f"Current Date: {current_date}")
print(f"Current Time: {current_time}")
```

```
[root@learnvm lab_exercises]# python3 exercise_5_4.py
Square root of 25: 5.0
2 raised to the power of 3: 8.0
Sine of 90 degrees: 1.0
Current Date: 2024-01-06
Current Time: 16:55:39.633274
```



IBM ICE (Innovation Centre for Education)

## Solution: Exercise 25

- Exercise 25: Third-Party Library Usage
- Estimated time: 00:15 minutes
- Aim: Install and use a third-party library (e.g., requests) to fetch data from a web API. Retrieve data and display it in your Python script.
- Learning objective:
  - The primary objective of this exercise is to familiarize learners with the process of installing and utilizing third-party libraries in Python. Specifically, the focus is on incorporating the requests library to interact with web APIs and retrieve data.
- Learning outcome:
  - Upon completing this exercise, learners will acquire practical skills in integrating external libraries into Python scripts. This experience enhances their ability to retrieve and manipulate data from web APIs, a valuable skill in various programming scenarios.

*Figure 8-26. Solution: Exercise 25*

### Methodology:

- **Environment setup:**
  - Open a text editor or an integrated development environment (IDE) for Python.
  - Ensure that Python is installed on your system.
- **Library installation:**
  - Use the package manager pip to install the requests library:

```
```bash
pip install requests
```

```
- **Script development:**
  - Create a Python script (e.g., `third\_party\_usage.py`) to implement the exercise.
- **API interaction**
  - Identify a public web API (e.g., JSONPlaceholder) and obtain the API endpoint URL.
  - Import the requests library into your script.
- **Data retrieval:**
  - Use the requests library to make an HTTP GET request to the API endpoint and retrieve data.

- **Data processing:**

- Process the retrieved data as needed for your specific use case.

- **Display output:**

- Display relevant information from the retrieved data within your Python script.

**Analysis:**

This exercise provides hands-on experience in the practical application of third-party libraries. By working with a real-world API, learners gain insights into the seamless integration of external tools, expanding the capabilities of Python scripts.

**Code implementation:**

```
import requests
# API endpoint URL (Example: JSONPlaceholder)
import requests
# API endpoint URL (Example: JSONPlaceholder)
api_url = "https://jsonplaceholder.typicode.com/todos/1"
# Make an HTTP GET request to the API
response = requests.get(api_url)
# Check if the request was successful (status code 200)
if response.status_code == 200:
    # Parse and display the retrieved data
    data = response.json()
    print("Data retrieved from the API:")
    print(f"User ID: {data['userId']}")
    print(f"Title: {data['title']}")
    print(f"Completed: {data['completed']}")
else:
    print(f"Failed to retrieve data. Status code: {response.status_code}")
```

**Output:**

The provided code makes an HTTP GET request to the JSONPlaceholder API endpoint and retrieves information about a todo item with ID 1. Here's the expected output:

Expected Output (Assuming the request is successful - status code 200):

**Methodology:**

- **Environment setup:**

- Open a text editor or an integrated development environment (IDE) for Python.
- Ensure that Python is installed on your system.

- **Library installation:**

- Use the package manager pip to install the requests library:

```
```bash
pip install requests
```

```

- **Script development:**

- Create a Python script (e.g., `third\_party\_usage.py`) to implement the exercise.

- **API interaction:**

- Identify a public web API (e.g., JSONPlaceholder) and obtain the API endpoint URL.
- Import the requests library into your script.

- **Data retrieval:**

- Use the requests library to make an HTTP GET request to the API endpoint and retrieve data.

- **Data processing:**

- Process the retrieved data as needed for your specific use case.

- **Display output:**

- Display relevant information from the retrieved data within your Python script.

**Analysis:**

This exercise provides hands-on experience in the practical application of third-party libraries. By working with a real-world API, learners gain insights into the seamless integration of external tools, expanding the capabilities of Python scripts.

**Code Implementation:**

```
import requests
# API endpoint URL (Example: JSONPlaceholder)
import requests
# API endpoint URL (Example: JSONPlaceholder)
api_url = "https://jsonplaceholder.typicode.com/todos/1"
# Make an HTTP GET request to the API
response = requests.get(api_url)
# Check if the request was successful (status code 200)
if response.status_code == 200:
    # Parse and display the retrieved data
    data = response.json()
    print("Data retrieved from the API:")
    print(f"User ID: {data['userId']}")
    print(f"Title: {data['title']}")
    print(f"Completed: {data['completed']}")
else:
    print(f"Failed to retrieve data. Status code: {response.status_code}")
```

**Output:**

The provided code makes an HTTP GET request to the JSONPlaceholder API endpoint and retrieves information about a todo item with ID 1. Here's the expected output:

Expected Output (Assuming the request is successful - status code 200):

**Methodology:**

- Environment Setup:
  - Open a text editor or an integrated development environment (IDE) for Python.
  - Ensure that Python is installed on your system.
- Library Installation:
  - Use the package manager pip to install the requests library:

```
```bash
```

```
pip install requests
```

```
...
```

- Script Development:
  - Create a Python script (e.g., `third\_party\_usage.py`) to implement the exercise.
- API Interaction
  - Identify a public web API (e.g., JSONPlaceholder) and obtain the API endpoint URL.
  - Import the requests library into your script.
- Data Retrieval:
  - Use the requests library to make an HTTP GET request to the API endpoint and retrieve data.
- Data Processing:
  - Process the retrieved data as needed for your specific use case.
- Display Output:
  - Display relevant information from the retrieved data within your Python script.

### **Analysis:**

This exercise provides hands-on experience in the practical application of third-party libraries. By working with a real-world API, learners gain insights into the seamless integration of external tools, expanding the capabilities of Python scripts.

### **Code Implementation:**

```
import requests

# API endpoint URL (Example: JSONPlaceholder)
import requests

# API endpoint URL (Example: JSONPlaceholder)
api_url = "https://jsonplaceholder.typicode.com/todos/1"

# Make an HTTP GET request to the API
response = requests.get(api_url)

# Check if the request was successful (status code 200)
if response.status_code == 200:
    # Parse and display the retrieved data
    data = response.json()
    print("Data retrieved from the API:")
    print(f"User ID: {data['userId']}")
    print(f"Title: {data['title']}")
    print(f"Completed: {data['completed']}")

else:
    print(f"Failed to retrieve data. Status code: {response.status_code}")
```

### **Output:**

The provided code makes an HTTP GET request to the JSONPlaceholder API endpoint and retrieves information about a todo item with ID 1. Here's the expected output:

Expected Output (Assuming the request is successful - status code 200):

```
[root@learnvm lab_exercises]# python3 exercise_5_5.py
Data retrieved from the API:
User ID: 1
Title: delectus aut autem
Completed: False
```



IBM ICE (Innovation Centre for Education)

## Solution: Exercise 26

- Exercise 26: Simple tkinter GUI
- Estimated time: 00:15 minutes
- Aim: Design a simple tkinter GUI application with buttons and labels. Implement functionality to update labels when buttons are clicked.
- Learning objective:
  - The primary objective of this exercise is to introduce learners to building graphical user interfaces (GUIs) using the tkinter library in Python. Participants will gain hands-on experience in creating basic GUI components and implementing event-driven functionality.
- Learning outcome:
  - Upon completing this exercise, learners will have acquired foundational skills in GUI development with Python's tkinter library. They will be able to design simple interactive applications, laying the groundwork for more complex GUI projects.

*Figure 8-27. Solution: Exercise 26*

### **Methodology:**

#### Environment Setup:

- Ensure that Python is installed on your system.
- tkinter is a standard Python library, so no additional installation is required.

#### Script development:

Open a text editor or an integrated development environment (IDE) for Python.

Create a Python script (e.g., `simple\_gui.py`) to implement the exercise.

#### GUI design:

Utilize the tkinter library to design a simple GUI with buttons and labels.

Place at least one label and two buttons on the GUI.

#### Event handling:

Implement functionality to update the label text when buttons are clicked.

For example, clicking Button 1 could change the label text to "Button 1 clicked."

#### GUI layout:

Experiment with different geometry managers (e.g., pack, grid, or place) to arrange the buttons and labels.

**Analysis:**

This exercise provides a practical introduction to GUI development in Python. Participants will gain insights into creating interactive applications, understanding the principles of event-driven programming and basic GUI layout.

**Code implementation:**

```
import tkinter as tk

def update_label_text(button_text):
    label.config(text=f"{button_text} clicked.")

# Create the main application window
root = tk.Tk()
root.title("Simple GUI")

# Create a label
label = tk.Label(root, text="Welcome to Simple GUI")
label.pack(pady=10)

# Create buttons
button1      = tk.Button(root,      text="Button      1",      command=lambda:
update_label_text("Button 1"))
button1.pack(side="left", padx=10)
button2      = tk.Button(root,      text="Button      2",      command=lambda:
update_label_text("Button 2"))
button2.pack(side="right", padx=10)

# Start the GUI application
root.mainloop()
```





## Solution: Exercise 27

- Exercise 27: GUI Button Actions
- Estimated time: 00:15 minutes
- Aim: Create a Python program that responds to user button clicks in a tkinter GUI. Perform actions like displaying messages when buttons are clicked.
- Learning objective:
  - This exercise aims to familiarize learners with handling button actions in a tkinter graphical user interface (GUI). Participants will gain practical experience in associating actions with button clicks and enhancing interactivity in GUI applications.
- Learning outcome:
  - By completing this exercise, learners will acquire the skills to design tkinter GUIs with responsive buttons. They will understand how to define actions for buttons and improve the user experience in their Python GUI applications.

*Figure 8-28. Solution: Exercise 27*

### **Methodology:**

- **Environment setup:**

Ensure Python is installed on your system.

Use the built-in tkinter library for GUI development.

- **Script development:**

Open a text editor or an integrated development environment (IDE) for Python.

Create a Python script (e.g., `button\_actions.py`) to implement the exercise.

- **GUI design:**

Design a tkinter GUI with at least two buttons.

Place a label on the GUI to display messages.

- **Button actions:**

Define actions for each button. For example, clicking Button 1 might display a "Button 1 clicked" message.

### **User interaction:**

Ensure that the GUI responds to user interactions by displaying appropriate messages.

### **Analysis:**

This exercise focuses on enhancing user interaction in GUI applications. Participants will gain insights into associating actions with buttons, a fundamental skill for creating responsive and dynamic GUIs.

### Code implementation:

```
import tkinter as tk

def display_message(message):
    label.config(text=message)

# Create the main application window
root = tk.Tk()
root.title("Button Actions")

# Create a label
label = tk.Label(root, text="Click a button to display a message.")
label.pack(pady=10)

# Create buttons with associated actions
button1      = tk.Button(root,      text="Button      1",      command=lambda:
display_message("Button 1 clicked"))
button1.pack(side="left", padx=10)

button2      = tk.Button(root,      text="Button      2",      command=lambda:
display_message("Button 2 clicked"))
button2.pack(side="right", padx=10)

# Start the GUI application
root.mainloop()
```





## Solution: Exercise 28

- Exercise 28: HTML Form Handling
- Estimated time: 00:45
- Aim: Building an HTML Form for Data Input and Implementing Form Handling in Python
- Learning objective
  - Create an HTML form with input fields and a submit button for user data entry.
  - Develop a Python CGI script to handle form submissions and display the entered data.
- Learning outcome
  - Upon implementing and executing this exercise, the learner will be able to:
  - Construct an HTML form for user input with various input fields.
  - Implement server-side handling of form submissions using Python CGI scripting.
  - Understand the basics of HTML form structure and Python CGI programming.

---

Figure 8-29. Solution: Exercise 28

### Methodology

- **Build the HTML form:**
  - Create an HTML document containing a form with input fields such as text, textarea, radio buttons, checkboxes, etc.
  - Add a submit button to allow users to submit the form.
- **Implement form handling in python:**
  - Write a Python CGI script to handle form submissions.
  - Retrieve data submitted through the form using CGI parameters.
  - Format and display the entered data on the server side.
- **Test the implementation:**
  - Access the HTML form through a web browser.
  - Enter data into the form and submit it.
  - Observe the server response displaying the entered data.
- **Evaluate and debug:**
  - Check for any issues in the form handling process.
  - Debug and optimize the Python CGI script for seamless data retrieval and display.

## Analysis

- Examine the functionality of the HTML form and Python CGI script.
- Analyze the server's ability to handle form submissions and display entered data.
- Understand the role of CGI scripting in processing user inputs from web forms.

## Code implementation

Usage: sample.html

Place the above smaple.html file inside Apache Web Server – “C:/Apache24/htdocs”

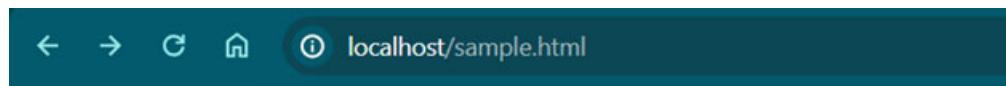
```
<!DOCTYPE html>
<html>
<head>
    <title>Form Handling Exercise</title>
</head>
<body>
    <h2>Data Entry Form</h2>
    <form action="/cgi-bin/process_form.py" method="post">
        <label for="name">Name:</label>
        <input type="text" id="name" name="name" required><br><br>
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required><br><br>
        <label for="age">Age:</label>
        <input type="number" id="age" name="age" required><br><br>
        <label for="comments">Comments:</label><br>
                    <textarea id="comments" name="comments" rows="4" cols="50"></textarea><br><br>
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

Usage: process\_form.py\_

- Please Change the below Python path as per system Python Env Path
- Place the above smaple.html file inside Apache Web Server – “C:/Apache24/cgi-bin/”

```
#!/usr/bin/python3
# C:/Users/Ahsan Alimaad/AppData/Local/Programs/Python/Python39/python.exe"
import cgi
import cgitb
cgitb.enable()
print("Content-type: text/html\n")
# Retrieve form data
form = cgi.FieldStorage()
name = form.getvalue("name")
email = form.getvalue("email")
```

```
age = form.getvalue("age")
comments = form.getvalue("comments")
# Display entered data
print("<h2>Submitted Data:</h2>")
print(f"<p>Name: {name}</p>")
print(f"<p>Email: {email}</p>")
print(f"<p>Age: {age}</p>")
print(f"<p>Comments: {comments}</p>")
Pre-Requirement
Install/Setup Apache Web Server
```



## Data Entry Form

Name:

Email:

Age:

Comments:



IBM ICE (Innovation Centre for Education)

## Solution: Exercise 29

- Exercise 29: Dynamic Web Content with CGI
- Estimated time: 00:45
- Aim: Implementing Dynamic Web Content Generation with Python CGI
- Learning objective
  - Create a Python CGI script to generate dynamic web content based on user requests.
  - Build a simple web application that responds to different URL parameters. Learning Outcome
- Learning outcome
- Upon implementing and executing this exercise, the learner will be able to:
  - Develop a Python CGI script for generating dynamic web content.
  - Understand the concept of handling URL parameters in a web application.
  - Enhance skills in creating dynamic and interactive web pages.

Figure 8-30. Solution: Exercise 29

### Methodology

- **Create python CGI script:**
  - Implement a Python CGI script that extracts and processes URL parameters.
  - Generate dynamic content based on the provided parameters.
  - Consider using the cgi module for handling CGI operations.
- **Build dynamic web application:**
  - Create a simple HTML structure to display the dynamic content.
  - Use JavaScript or other client-side technologies to enhance interactivity (optional).
  - Include placeholders in the HTML to be filled by the Python CGI script.
- **Test the implementation:**
  - Access the web application with different URL parameters.
  - Observe the changes in the displayed content based on the provided parameters.
- **Evaluate and debug:**
  - Check for any issues in dynamic content generation.
  - Optimize the Python CGI script for efficient handling of URL parameters.

### Analysis

- Observe the functionality of the dynamic web application.
- Analyze the effectiveness of the Python CGI script in responding to user requests.
- Understand the significance of dynamic content in web applications.

### **Code implementation**

Usage: process\_form.py\_

- Please Change the below Python path as per system Python Env Path
- Place the above smaple.html file inside Apache Web Server – “C:/Apache24/cgi-bin/”

```
#!"C:/Users/Ahsan Alimaad/AppData/Local/Programs/Python/Python39/python.exe"
import cgi
# Get URL parameters
params = cgi.FieldStorage()
content_type = params.getvalue('content_type', 'default')
# Define dynamic content based on parameters
content_map = {
    'default': 'Welcome to the default page!',
    'about': 'This is the about page. Learn more about our web application.',
    'contact': 'Contact us for any inquiries or support.',
}
# Print HTML content
print("Content-type: text/html\n")
print("<html>")
print("<head><title>Dynamic Web Content</title></head>")
print("<body>")
print("<h2>Dynamic Web Content</h2>")
print("<p>{}</p>".format(content_map.get(content_type, 'Invalid content type')))
print("</body>")
print("</html>")

Pre-Requirement
Install/Setup Apache Web Server
```



## Data Entry Form

Name:

Email:

Age:

Comments:



## Solution: Exercise 30

- Exercise 30: Form Data Validation
- Estimated time: 00:45
- Aim: Implementing Form Data Validation in Python CGI Script
- Learning objective
  - Extend a client-server program with Python CGI scripting to validate form data.
  - Implement form data validation for required fields and display validation messages.
- Learning outcome
- Upon implementing and executing this exercise, the learner will be able to:
  - Enhance a client-server architecture with Python CGI scripting for form data validation.
  - Implement validation checks for required fields in a form
  - Display appropriate validation messages to the user.

*Figure 8-31. Solution: Exercise 30*

### Methodology

- **Modify the client-server program:**
  - Extend the existing client-server program with Python CGI scripting to handle form data.
  - Update the server-side code to process form data received from the client.
- **Implement form data validation:**
  - Identify and check for required fields in the form data.
  - Design validation checks to ensure the correctness of the submitted data.
- **Test the implementation:**
  - Create a sample HTML form on the client side with required and optional fields.
  - Submit the form data to the server and observe the validation messages.
- **Evaluate and debug:**
  - Check for any issues in the form data validation process.
  - Debug and optimize the code for a seamless validation experience.

### Analysis

- Observe the integration of form data validation in the client-server architecture.
- Analyze the effectiveness of the implemented validation checks.

- Understand the importance of validating user inputs for robust server-side processing.

### **Code implementation**

Usage: sample.html

- Place the above smaple.html file inside Apache Web Server – “C:/Apache24/htdocs”

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Form Data Validation Exercise</title>
</head>
<body>
    <h2>Form Data Validation Exercise</h2>
    <form action="/cgi-bin/process_form.py" method="post">
        <label for="name">Name:</label>
        <input type="text" id="name" name="name" required><br>
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required><br>
        <label for="message">Message:</label><br>
        <textarea id="message" name="message" rows="4" cols="50" required></textarea><br>
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

Usage: process\_form.py\_

- Please Change the below Python path as per system Python Env Path
- Place the above smaple.html file inside Apache Web Server – “C:/Apache24/cgi-bin/”

```
#!/usr/bin/python3
# C:/Users/Ahsan Alimaad/AppData/Local/Programs/Python/Python39/python.exe
import cgi

def validate_form_data(name, email, message):
    errors = []
    if not name:
        errors.append("Name is required.")
    if not email:
        errors.append("Email is required.")
    if not message:
        errors.append("Message is required.")
    return errors
```

```
# Get form data
form = cgi.FieldStorage()
name = form.getvalue('name')
email = form.getvalue('email')
message = form.getvalue('message')

# Validate form data
validation_errors = validate_form_data(name, email, message)

# Print HTML content
print("Content-type: text/html\n")
print("<html>")
print("<head><title>Form Submission Result</title></head>")
print("<body>")
print("<h2>Form Submission Result</h2>")

# Display validation errors or submitted data
if validation_errors:
    print("<p><strong>Validation Errors:</strong></p>")
    print("<ul>")
    for error in validation_errors:
        print("<li>{}</li>".format(error))
    print("</ul>")
else:
    print("<p><strong>Name:</strong> {}</p>".format(name))
    print("<p><strong>Email:</strong> {}</p>".format(email))
    print("<p><strong>Message:</strong> {}</p>".format(message))

print("</body>")
print("</html>")
```

Pre-Requirement

Install/Setup Apache Web Server



## Data Entry Form

Name:

Email:

Age:

Comments:



## Solution: Exercise 31

- Exercise 31: Client-Server Communication
- Estimated time: 00:15 minutes
- Aim: Create a simple client-server program using socket programming in Python. Implement basic communication between the client and server.
- Learning objective:
- This exercise aims to introduce participants to client-server communication using socket programming in Python. Learners will understand the fundamental concepts of establishing connections, sending data, and receiving responses between a client and a server.
- Learning outcome:
- Upon completing this exercise, participants will be able to implement basic client-server communication in Python. They will gain insights into socket programming and be equipped to build more advanced networked applications.

*Figure 8-32. Solution: Exercise 31*

### Methodology:

- **Socket initialization:**
  - In the server-side code, initialize a socket using the `socket` module.
  - Bind the socket to a specific address and port using `bind()`.
  - Listen for incoming connections using `listen()`.
- **Accepting connections:**
  - Accept a connection from a client using `accept()`.
  - On the client side, create a socket and connect to the server using `connect()`.

### Data transmission:

- Implement code to send data from the client to the server and vice versa using `send()` and `recv()`.
- **Error handling:**
  - Incorporate error handling mechanisms to manage potential issues during socket communication.

### Analysis:

This exercise provides participants with hands-on experience in establishing a basic client-server communication model. Learners will gain a practical understanding of socket programming, enabling them to build networked applications for various purposes.

### Code implementation:

**Server-Side Code:**

```

import socket
# Create a TCP/IP socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Bind the socket to a specific address and port
server_address = ('localhost', 8080) # Server listens on port 8080
server_socket.bind(server_address)
# Listen for incoming connections
server_socket.listen(1)
# Accept a connection
client_socket, client_address = server_socket.accept()
# Receive data from the client
data = client_socket.recv(1024).decode()
print("Received message from client:", data)
# Send a response back to the client
response = "Hello from server!".encode()
client_socket.sendall(response)
# Close the connections
client_socket.close()
server_socket.close()

```

**Client-Side Code:**

```

import socket
# Create a TCP/IP socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Connect to the server
server_address = ('localhost', 8080)
client_socket.connect(server_address)
# Send a message to the server
message = "Hello from client!".encode()
client_socket.sendall(message)
# Receive the server's response
response = client_socket.recv(1024).decode()
print("Received message from server:", response)
# Close the connection
client_socket.close()

```

```

[root@learnvm lab_exercises]# python3 exercise_7_1_server.py
Received message from client: Hello from client!
[root@learnvm lab_exercises]#
[root@learnvm lab_exercises]# python3 exercise_7_1_client.py
Received message from server: Hello from server!

```



## Solution: Exercise 32

- Exercise 32: Enhanced Client-Server Communication
- Estimated time: 00:45
- Aim: Implementing Data Transfer between Client and Server
- Learning objective
  - Extend a basic client-server program to support bidirectional data transfer.
  - Implement sending and receiving data between the client and server.
- Learning outcome
  - Upon implementing and executing this exercise, the learner will be able to:
  - Enhance a client-server architecture for efficient data exchange.
  - Implement data transmission protocols for improved communication.
  - Develop a deeper understanding of socket programming in Python.

*Figure 8-33. Solution: Exercise 32*

### Methodology

- **Modify the client-server program:**
  - Extend the existing client-server program to support Sending message from client to server.
  - Add functions for sending and receiving data.
- **Implement data transfer:**
  - Use Python's socket library to establish a connection.
  - Design a simple protocol for data exchange.
- **Test the implementation:**
  - Create sample data on the client side.
  - Send the data to the server and receive a response.
- **Evaluate and debug:**
  - Check for any issues in data transmission.
  - Debug and optimize the code for smooth communication.

### Analysis

- Observe the improved communication between the client and server.
- Analyze the efficiency and reliability of the implemented data transfer mechanism.

- Understand the importance of a well-defined protocol in client-server applications.

### Code implementation

Usage: Server\_7.2.py

```
import socket
import threading

def handle_client(client_socket):
    while True:
        # Receive data from the client
        data = client_socket.recv(1024)
        if not data:
            break
        # Print the received data on the server side
        print(f"Received from client: {data.decode('utf-8')}")
        # Send a response back to the client (optional)
        response = "Server received your message"
        client_socket.send(response.encode('utf-8'))
    # Close the client socket when the loop breaks
    client_socket.close()

def start_server():
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    # Bind the server to a specific address and port
    server_socket.bind(('127.0.0.1', 5555))
    # Listen for incoming connections (max 5 clients in the queue)
    server_socket.listen(5)
    print("Server listening on port 5555")
    while True:
        # Accept a connection from a client
        client_socket, addr = server_socket.accept()
        print(f"Accepted connection from {addr}")
        # Create a new thread to handle the client
        client_handler = threading.Thread(target=handle_client,
  args=(client_socket,))
        client_handler.start()

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

Usage: Client\_7.2.py

```
import socket

def start_client():
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    # Connect to the server
```

```
client_socket.connect(('127.0.0.1', 5555))
print("Connected to server")
while True:
    # Get user input to send to the server
    message = input("Enter message to send to server (type 'exit' to quit):
")
    # Send the message to the server
    client_socket.send(message.encode('utf-8'))
    # Receive and print the response from the server
    response = client_socket.recv(1024)
    print(f"Server response: {response.decode('utf-8')}")
    # Check for exit condition
    if message.lower() == 'exit':
        break
    # Close the client socket when done
    client_socket.close()
if __name__ == "__main__":
    start_client()
```

```
Server.py
client.py
```

```
Server.py > start_server
client.py > start_client

33     client_socket, addr = server_socket.accept()
34     print(f"Accepted connection from {addr}")
35
36     # Create a new thread to handle the client
37     client_handler = threading.Thread(
38         target=handle_client, args=(client_socket,))
39     client_handler.start()
40
41 if __name__ == "__main__":
42     start_server()

18     response = client_socket.recv(1024)
19     print(f"Server response: {response.decode('utf-8')}")
20
21     # Check for exit condition
22     if message.lower() == 'exit':
23         break
24
25     # Close the client socket when done
26     client_socket.close()
27
28 if __name__ == "__main__":
29     start_client()

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS Python < > < > < > < >

PS C:\Users\mahia\Downloads\PythonExcercise\ClientServer> & C:/Users/mahia/AppData/Local/Programs/Python/Python310/python.exe c:/Users/mahia/Downloads/PythonExcercise/ClientServer/Server.py
Server listening on port 5555
```



## Solution: Exercise 33

- Exercise 33: JSON Serialization and Deserialization
- Estimated time: 00:15 minutes
- Aim: Serialize Python objects into JSON format and then deserialize them back into Python objects. Demonstrate data interchange between JSON and Python.
- Learning objective:
  - This exercise aims to familiarize participants with the process of serializing and deserializing Python objects using JSON (JavaScript Object Notation). Participants will learn how to convert Python data structures into a JSON format and vice versa for effective data interchange.
- Learning outcome:
  - Upon completing this exercise, participants will be able to apply JSON serialization and deserialization techniques to facilitate communication between Python programs and other systems that support JSON data interchange.

*Figure 8-34. Solution: Exercise 33*

### Methodology:

- **Serialization (Python to JSON):**

Use the `json` module to serialize a Python dictionary or list into JSON format.

- **Deserialization (JSON to Python):**

Demonstrate the reverse process by deserializing a JSON-formatted string into a Python data structure.

- **Data interchange:**

Show an example of how JSON can be used for data interchange between Python programs.

- **Error handling:**

Implement error handling mechanisms for potential issues during serialization and deserialization.

### Analysis:

This exercise emphasizes the importance of JSON as a lightweight data interchange format and its seamless integration with Python. Participants will gain practical insights into using JSON to exchange data between different systems or components.

### Code implementation:

```
import json

# Serialization (Python to JSON)
python_data = { 'name': 'John', 'age': 30, 'city': 'New York'}
```

```
json_data = json.dumps(python_data)    # Serialize Python dictionary to JSON
print("Serialized JSON:", json_data)
# Deserialization (JSON to Python)
decoded_data = json.loads(json_data)   # Deserialize JSON to Python dictionary
print("Deserialized Python data:", decoded_data)
# Error Handling (Optional)
try:
    invalid_json = '{"name": "Alice", "age": 25}'  # Corrected: Use double quotes
for property names
    invalid_data = json.loads(invalid_json)    # This will raise a ValueError
except json.JSONDecodeError as e:
    print("Error during JSON decoding:", str(e))
```

---

```
Serialized JSON: {"name": "John", "age": 30, "city": "New York"}
Deserialized Python data: {'name': 'John', 'age': 30, 'city': 'New York'}
```



## Solution: Exercise 34

- Exercise 34: Pickle Serialization
- Estimated time: 00:15 minutes
- Aim: Use Pickle to serialize and deserialize Python objects. Save Python objects to a file using Pickle and then load them back.
- Learning objective:
  - This exercise aims to introduce participants to Pickle, a module in Python that provides a convenient way to serialize and deserialize Python objects. Participants will learn how to use Pickle to store and retrieve Python objects, enabling data persistence.
- Learning outcome:
  - Upon completing this exercise, participants will be able to apply Pickle for serializing complex Python objects, storing them in a file, and later retrieving them. This skill is particularly useful for saving and loading data structures between different program runs.

Figure 8-35. Solution: Exercise 34

### **Methodology:**

- **Serialization (Python to Pickle):**

Use the `pickle` module to serialize a Python object (e.g., a list or dictionary) and save it to a file.

- **Deserialization (Pickle to Python):**

Demonstrate how to read the Pickle file and deserialize the data back into a Python object.

- **Data persistence:**

Discuss the advantages of using Pickle for data persistence in Python applications.

- **Error handling (Optional):**

Implement error handling mechanisms for potential issues during Pickle serialization and deserialization.

### **Analysis:**

This exercise emphasizes the role of Pickle in preserving complex Python objects across different program sessions. Participants will gain insights into the benefits and considerations when using Pickle for data serialization.

### **Code implementation:**

```
import pickle

# Serialization (Python to Pickle)
python_object = {'name': 'Jane', 'age': 28, 'city': 'San Francisco'}
```

```
with open('serialized_data.pkl', 'wb') as pickle_file:  
    pickle.dump(python_object, pickle_file)  
# Deserialization (Pickle to Python)  
with open('serialized_data.pkl', 'rb') as pickle_file:  
    loaded_object = pickle.load(pickle_file)  
    print("Deserialized Python object:", loaded_object)
```

---

```
Deserialized Python object: {'name': 'Jane', 'age': 28, 'city': 'San Francisco'}
```



IBM ICE (Innovation Centre for Education)

## Solution: Exercise 35

- Exercise 35: Networked Data Exchange
- Estimated time: 00:15 minutes
- Aim: Develop a Python script that simulates data exchange over a network using sockets and serialization. Send and receive data between client and server.
- Learning objective:
  - This exercise aims to introduce participants to networked data exchange using sockets and serialization in Python. Participants will learn how to send and receive structured data between a client and server, facilitating more complex communication.
- Learning outcome:
  - Upon completing this exercise, participants will be able to implement a basic client-server system that exchanges serialized data over a network. Learners will gain practical knowledge of sending and receiving structured information between different entities in a networked environment.

*Figure 8-36. Solution: Exercise 35*

### **Methodology:**

- Serialization and deserialization:

Introduce the concept of serialization using modules like `pickle` or `json`. Discuss how data can be converted into a format that can be easily transmitted over the network.

- **Server-side implementation:**

Develop the server-side code to receive data from the client, deserialize it, perform any necessary processing, and send a response back.

- **Client-side implementation:**

Create the client-side code to serialize data, send it to the server, receive the server's response, and deserialize it for further use.

- **Data exchange simulation:**

Simulate a scenario where the client sends structured data (e.g., a dictionary or object) to the server, and the server processes and responds accordingly.

### **Analysis:**

This exercise provides participants with hands-on experience in implementing a basic networked data exchange system. Understanding how to serialize and deserialize data for communication lays the foundation for more complex networked applications.

### **Code implementation:**

```

import socket
import pickle
# Create a TCP/IP socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Bind the socket to a specific address and port
server_address = ('localhost', 8080) # Server listens on port 8080
server_socket.bind(server_address)
# Listen for incoming connections
server_socket.listen(1)
# Accept a connection
client_socket, client_address = server_socket.accept()
# Receive serialized data from the client
data = client_socket.recv(1024)
received_data = pickle.loads(data)
# Process the received data (example: print it)
print("Received data from client:", received_data)
# Send a response back to the client
response_data = {'status': 'success', 'message': 'Data received and processed'}
response_pickle = pickle.dumps(response_data)
client_socket.sendall(response_pickle)
# Close the connections
client_socket.close()
server_socket.close()

Client-Side Code:
import socket
import pickle
# Create a TCP/IP socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Connect to the server
server_address = ('localhost', 8080)
client_socket.connect(server_address)
# Prepare data to be sent (example: dictionary)
data_to_send = {'name': 'John Doe', 'age': 25, 'city': 'Example City'}
serialized_data = pickle.dumps(data_to_send)
# Send the serialized data to the server
client_socket.sendall(serialized_data)
# Receive the server's response
response_data = client_socket.recv(1024)
received_response = pickle.loads(response_data)
# Process the response (example: print it)

```

```
print("Server response:", received_response)
# Close the connection
client_socket.close()
```

```
[root@learnvm lab_exercises]# python3 exercise_7_5_server.py
Received data from client: {'name': 'John Doe', 'age': 25, 'city': 'Example City'
[root@learnvm lab_exercises]# █
[root@learnvm lab_exercises]# python3 exercise_7_5_client.py
Server response: {'status': 'success', 'message': 'Data received and processed'}
```



## Solution: Exercise 36

- Exercise 36: Basic NumPy Operations
- Estimated time: 00:15 minutes
- Aim: Create a NumPy array and perform basic operations like addition, subtraction, multiplication, and division.
- Learning objective:
  - This exercise aims to introduce participants to basic operations using NumPy arrays in Python. Participants will learn how to create arrays and perform fundamental mathematical operations efficiently using NumPy.
- Learning outcome:
- Upon completing this exercise, participants will be able to:
  - Create NumPy arrays.
  - Perform addition, subtraction, multiplication, and division operations on NumPy arrays.

*Figure 8-37. Solution: Exercise 36*

### **Methodology:**

- **Introduction to numpy:**

Provide a brief overview of NumPy and its advantages for numerical operations in Python.

- **Creating NumPy Arrays:**

Demonstrate how to create NumPy arrays using `numpy.array()` and `numpy.arange()`.

- **Basic operations:**

Explain the basic mathematical operations that can be performed on NumPy arrays.

Addition ('+'), Subtraction ('-'), Multiplication ('\*'), Division ('/'), etc.

- **Hands-on exercise:**

- Allow participants to practice creating NumPy arrays and performing basic operations on them.

### **Analysis:**

Understanding basic NumPy operations is essential for efficient numerical computation in Python. NumPy provides a convenient and optimized way to perform array-based mathematical operations.

### **Code implementation:**

```
import numpy as np
# Create NumPy arrays
```

```
array_a = np.array([1, 2, 3])
array_b = np.array([4, 5, 6])
# Perform basic operations
addition_result = array_a + array_b
subtraction_result = array_a - array_b
multiplication_result = array_a * array_b
division_result = array_a / array_b
# Display the results
print("Array A:", array_a)
print("Array B:", array_b)
print("Addition Result:", addition_result)
print("Subtraction Result:", subtraction_result)
print("Multiplication Result:", multiplication_result)
print("Division Result:", division_result)
```

```
[root@learnvm lab_exercises]# python3 exercise_7_6.py
Array A: [1 2 3]
Array B: [4 5 6]
Addition Result: [5 7 9]
Subtraction Result: [-3 -3 -3]
Multiplication Result: [ 4 10 18]
Division Result: [0.25 0.4 0.5 ]
```



## Solution: Exercise 37

- Exercise 37: Data Filtering and Selection
- Estimated time: 00:15 minutes
- Aim: Explore data filtering and selection techniques using Pandas in Python.
- Learning objective:
  - This exercise aims to familiarize participants with data filtering and selection methods in Pandas, allowing them to extract specific subsets of data from a DataFrame.
- Learning outcome:
- Upon completing this exercise, participants will be able to:
  - Filter and select data based on specific conditions.
  - Utilize Pandas functions for data filtering and selection.

Figure 8-38. Solution: Exercise 37

### Methodology:

- **Introduction to data filtering:**

Provide an overview of the importance of data filtering in data analysis.

Discuss common scenarios where data filtering is necessary.

- **Pandas dataframes:**

Briefly introduce Pandas DataFrames and their role in handling tabular data.

- **Data filtering techniques:**

Demonstrate various techniques for filtering data in Pandas, including boolean indexing, conditional statements, and query methods.

- **Hands-on exercise:**

Provide participants with a dataset and guide them through filtering and selecting specific portions of the data.

### Analysis:

Data filtering and selection are fundamental skills in data analysis, allowing practitioners to focus on relevant subsets of data for further exploration and insights. Participants should understand how to apply these techniques effectively to solve real-world data-related challenges.

### Code Implementation:

```
import pandas as pd
```

```
# Create a sample DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],
        'Age': [25, 30, 22, 35],
        'Salary': [50000, 60000, 45000, 70000]}
df = pd.DataFrame(data)
# Data filtering and selection
young_employees = df[df['Age'] < 30]
high_salary_employees = df[df['Salary'] > 55000]
# Display the results
print("Original DataFrame:")
print(df)
print("\nYoung Employees (Age < 30):")
print(young_employees)
print("\nHigh Salary Employees (Salary > 55000):")
print(high_salary_employees)
```

```
[root@learnvm lab_exercises]# python3 exercise_7_7.py
Original DataFrame:
      Name  Age  Salary
0   Alice   25    50000
1     Bob   30    60000
2  Charlie   22    45000
3   David   35    70000

Young Employees (Age < 30):
      Name  Age  Salary
0   Alice   25    50000
2  Charlie   22    45000

High Salary Employees (Salary > 55000):
      Name  Age  Salary
1     Bob   30    60000
3   David   35    70000
```



## Solution: Exercise 38

- Exercise 38: Data Analysis with Pandas
- Estimated time: 00:15 minutes
- Aim: Use Pandas to read data from a CSV file and perform data analysis. Calculate statistics like mean, median, and standard deviation on the dataset.
- Learning objective:
  - This exercise aims to familiarize participants with the data analysis capabilities of Pandas. Participants will learn how to read data from a CSV file and perform basic statistical analysis on the dataset.
- Learning outcome:
- Upon completing this exercise, participants will be able to:
  - Load data from a CSV file into a Pandas DataFrame.
  - Understand and apply basic statistical analysis functions in Pandas.
  - Calculate measures like mean, median, and standard deviation.

Figure 8-39. Solution: Exercise 38

### Methodology:

- **Data loading:**
  - Load a dataset from a CSV file into a Pandas DataFrame.
- **Statistical analysis:**
  - Use Pandas functions to calculate mean, median, and standard deviation.
- **Data presentation:**
  - Display key statistics and insights from the dataset.

### Analysis:

Participants will gain hands-on experience in using Pandas for data analysis. The analysis phase will involve interpreting the calculated statistics to derive meaningful insights about the dataset.

### Code Implementation:

```
import pandas as pd

# Creating a DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Emma'],
    'Age': [25, 30, 22, 35, 28],
```

```

'Salary': [50000, 60000, 45000, 70000, 55000],
'Hours_Worked': [40, 45, 35, 50, 42]
}

df = pd.DataFrame(data)
# Statistical Analysis
mean_age = df['Age'].mean()
median_salary = df['Salary'].median()
std_dev_hours_worked = df['Hours_Worked'].std()

# Data Presentation
print("Dataset Statistics:")
print(f"Mean Age: {mean_age:.2f} years")
print(f"Median Salary: ${median_salary:.2f}")
print(f"Standard Deviation of Hours Worked: {std_dev_hours_worked:.2f} hours")

Example 2

import pandas as pd
# Create a sample DataFrame
data = {'A': [1, 2, 3, 4, 5],
        'B': [10, 20, 30, 40, 50]}
df = pd.DataFrame(data)
# Add a new column 'C' which is the sum of columns 'A' and 'B'
df['C'] = df['A'] + df['B']
# Find the maximum value from column 'C'
max_value_C = df['C'].max()
# Find the minimum value from column 'C'
min_value_C = df['C'].min()
print("DataFrame with added column 'C':")
print(df)
print("\nMaximum value from column 'C':", max_value_C)
print("Minimum value from column 'C':", min_value_C)

```

**Dataset Statistics:**  
**Mean Age: 28.00 years**  
**Median Salary: \$55000.00**  
**Standard Deviation of Hours Worked: 5.59 hours**



## Solution: Exercise 39

- Exercise 39: DataFrame Manipulation
- Estimated time: 00:15 minutes
- Aim: Explore DataFrame manipulation in Pandas. Sort, filter, and perform various operations on a dataset loaded into a DataFrame.
- Learning objective:
  - This exercise aims to introduce participants to DataFrame manipulation using the Pandas library. Participants will learn how to sort, filter, and perform various operations on data stored in a Pandas DataFrame.
- Learning Outcome:
- Upon completing this exercise, participants will be able to:
  - Load data into a Pandas DataFrame.
  - Understand and apply sorting techniques on DataFrame columns.
  - Filter and subset data based on specific conditions.
  - Perform basic operations such as adding new columns and aggregating data.

*Figure 8-40. Solution: Exercise 39*

### Methodology:

- **Data loading:**

Load a sample dataset into a Pandas DataFrame.

- **Sorting:**

Demonstrate how to sort the DataFrame based on one or more columns.

- **Filtering:**

Show how to filter data based on specific conditions.

- **Column operations:**

Introduce operations like adding a new column and performing calculations.

- **Aggregation:**

Illustrate basic aggregation functions on DataFrame columns.

### Analysis:

Participants will gain practical experience in manipulating data using Pandas, understanding how to organize, sort, and filter information within a DataFrame. The analysis will focus on the effectiveness and efficiency of different DataFrame manipulation techniques.

### Code implementation:

```
import pandas as pd
```

```

# Data Loading
data = {'Name': ['John', 'Alice', 'Bob', 'Charlie', 'David'],
        'Age': [28, 24, 22, 30, 25],
        'Salary': [50000, 60000, 55000, 70000, 48000]}
df = pd.DataFrame(data)

# Sorting
df_sorted = df.sort_values(by='Age')

# Filtering
young_employees = df[df['Age'] < 25]

# Column Operations
df['Salary_Increase'] = df['Salary'] * 1.1 # Increase salary by 10%

# Aggregation
average_salary = df['Salary'].mean()

# Handling NULL or Empty Values
# Checking for NULL values
print("\nChecking for NULL or Empty Values:")
print(df.isnull())

# Filling NULL values with the mean of the column
df.fillna(value=df.mean(), inplace=True)

# Printing DataFrame after filling NULL values
print("\nDataFrame after Filling NULL or Empty Values:")
print(df)

print("\nOriginal DataFrame:")
print(df)

print("\nSorted DataFrame:")
print(df_sorted)

print("\nYoung Employees:")
print(young_employees)

print("\nDataFrame with Salary Increase:")
print(df)

print("\nAverage Salary:", average_salary)

```

```
ert code cell below
```

Checking for NULL or Empty Values:

	Name	Age	Salary	Salary_Increase
0	False	False	False	False
1	False	False	False	False
2	False	False	False	False
3	False	False	False	False
4	False	False	False	False

DataFrame after Filling NULL or Empty Values:

	Name	Age	Salary	Salary_Increase
0	John	28	50000	55000.0
1	Alice	24	60000	66000.0
2	Bob	22	55000	60500.0
3	Charlie	30	70000	77000.0
4	David	25	48000	52800.0

Original DataFrame:

	Name	Age	Salary	Salary_Increase
0	John	28	50000	55000.0
1	Alice	24	60000	66000.0
2	Bob	22	55000	60500.0
3	Charlie	30	70000	77000.0
4	David	25	48000	52800.0

Sorted DataFrame:

	Name	Age	Salary
2	Bob	22	55000
1	Alice	24	60000
4	David	25	48000
0	John	28	50000
3	Charlie	30	70000

Young Employees:

	Name	Age	Salary
1	Alice	24	60000
2	Bob	22	55000

DataFrame with Salary Increase:

	Name	Age	Salary	Salary_Increase
1	Alice	24	60000	66000.0
2	Bob	22	55000	60500.0



## Solution: Exercise 40

- Exercise 40: Database CRUD Operations
- Estimated time: 00:20 minutes
- Aim: Connect to a SQLite database in Python and perform CRUD (Create, Read, Update, Delete) operations using SQL queries. Create a Python script that interacts with a database.
- Learning objective:
  - This exercise aims to familiarize participants with the basics of interacting with a SQLite database in Python. Participants will learn how to establish a connection, execute SQL queries, and perform CRUD operations (Create, Read, Update, Delete) using Python's SQLite module.
- Learning outcome:
- Upon completing this exercise, participants will be able to:
  - Establish a connection to an SQLite database in Python.
  - Execute SQL queries for creating tables, inserting data, fetching data, updating records, and deleting records.
  - Understand the principles of CRUD operations in a database context.

Figure 8-41. Solution: Exercise 40

**Methodology:**

- **Database connection:**

Demonstrate how to establish a connection to an SQLite database using the `sqlite3` module.

- **Create table:**

Execute SQL queries to create a table with relevant fields.

- **Insert data:**

Insert sample data into the created table.

- **Read (select) data:**

Fetch and display data from the database.

- **Update records:**

Modify existing records in the database.

- **Delete records:**

Delete specific records from the database.

### **Analysis:**

Participants will gain hands-on experience in working with databases, understanding the importance of CRUD operations in managing data. The analysis will focus on the practical application of SQL queries within a Python script.

**Code implementation:**

```

import sqlite3
# Database Connection
conn = sqlite3.connect('example.db')
cursor = conn.cursor()
# Create Table
cursor.execute('''
    CREATE TABLE IF NOT EXISTS users (
        id INTEGER PRIMARY KEY,
        name TEXT NOT NULL,
        age INTEGER
    )
''')
# Insert Data
cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", ('John Doe', 30))
cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", ('Alice Smith', 25))
# Read (Select) Data
cursor.execute("SELECT * FROM users")
result = cursor.fetchall()
print("Data from the 'users' table:", result)
# Update Records
cursor.execute("UPDATE users SET age = ? WHERE name = ?", (31, 'John Doe'))
# Delete Records
cursor.execute("DELETE FROM users WHERE name = ?", ('Alice Smith',))
# Commit changes and close connection
conn.commit()
conn.close()

```

```

[root@learnvm lab_exercises]# python3 exercise_7_10.py
Data from the 'users' table: [(1, 'John Doe', 30), (2, 'Alice Smith', 25)]

```

## Unit summary



IBM ICE (Innovation Centre for Education)

**Having completed this unit, you should be able to:**

- Present a case study on the integration of virtualization and cloud technologies
- Explore workload characterization and its suitability for various cloud deployment models
- Discuss industry-specific cloud workloads and scenarios.
- Explore auto-scaling and load balancing strategies.
- Plan and execute cloud migration projects
- Understand the challenges and considerations for migrating to the cloud
- Learn to design scalable and high-performance cloud applications

---

*Figure 8-42. Unit summary*

Unit summary is as stated above.

# Appendix A. Review answers

## Unit 1, "Introduction to Python Programming"

Solutions for [Figure 1-16, "Checkpoint \(1 of 2\),"](#) on page 1-49

### Checkpoint solutions (1 of 2)



IBM ICE (Innovation Centre for Education)

#### Multiple choice questions:

1. Which of the following is/are true regarding Python IDLE?
  - a) It is an integrated development environment for Python.
  - b) It is a Python Shell.
  - c) Both a and b
  - d) None of the above
2. Which of the following is/are not valid variable names in python?
  - a) length (white spaces appended before the word length)
  - b) 100\_emp
  - c) abc\$
  - d) All of the above
3. Which of the following gives an output as 8?
  - a) >>> 2 \* 3
  - b) >>> 2 \*\* 3
  - c) >>> 2 ^ 3
  - d) None of the above

Solutions for [Figure 1-17, "Checkpoint \(2 of 2\),"](#) on page 1-50



## Checkpoint solutions (2 of 2)

### Fill in the blanks:

- Refer the code given below and fill in the blanks:

```
num = 16
num1 = num / 6
num2 = num // 6
num3 = num // 6.0
print("num1 =", num1)
print("num2 =", num2)
print("num3 =", num3)
```

Ans: num1=2.66666, num2=2, num3=2.0

- Consider the following expression. Select the correct operators in place of a, b, c, d from the following given options so that the output is 55.

7  a  6  b  2%4\*(9  c  5)  d  2

Ans: a=+, b= //, c= -, d= \*\*

- Python supports various data types for variables, such as integers, floats, strings, lists.
- The input() function in python is used to take user input from the console.

### True or false:

- global = '30' creates a string variable in Python. **False**
- Assert = True creates a Boolean variable in Python. **True**
- Variable names are case-sensitive. **True**

## Unit 2, "Control Flow and Loops"

Solutions for [Figure 2-14, "Checkpoint \(1 of 2\),"](#) on page 2-37



## Checkpoint solutions (1 of 2)

### Multiple choice questions:

1. Predict the output of the code snippet given below:

```
> Sample_string = "Python is easy"  
> For l in sample_string:  
>     print(l, end=" , ")
```

- a) P,y,t,h,o,n, .i,s, ,e,a,s,y
- b) P,y,t,h,o,n, ,i,s, , e,a,s,y,
- c) Syntax error
- d) Python, is, easy,

2. Conditions in Python are expressions that evaluate to either \_\_\_\_\_ or \_\_\_\_\_.

- a) True or False
- b) NOT or OR
- c) None of the above
- d) All of the above

3. The equivalent functionality can be achieved using\_\_\_\_\_ statements.

- a) if-elif-else
- b) If else
- c) While if
- d) None of the above

---

Solutions for [Figure 2-15, "Checkpoint \(2 of 2\),"](#) on page 2-38



IBM ICE (Innovation Centre for Education)

## Checkpoint solutions (2 of 2)

### Fill in the blanks:

1. The range() function in Python is commonly used in **for** loops for controlled iteration.
2. The **switch-case** statement in Python provides an alternative to using multiple if-elif-else statements for handling different cases.
3. A **set** is an unordered, immutable collection of unique elements in Python.
4. A **Tuple** is an ordered, immutable collection in Python.

### True/False:

1. Once a tuple is created, elements cannot be added or removed from it in Python. **True**
2. The sort() method modifies the original list and returns a new sorted list in Python. **False**
3. The sorted() function can be used with sets and tuples to return a sorted list. **True**

---

## Unit 3, "Object-Oriented Programming (OOP) in Python"

Solutions for [Figure 3-22, "Checkpoint \(1 of 2\),"](#) on page 3-53



IBM ICE (Innovation Centre for Education)

## Checkpoint solutions (1 of 2)

### Multiple choice questions:

1. Inheritance in Python allows a class to:
  - a) Be instantiated multiple times
  - b) Inherit attributes and methods from another class**
  - c) Be defined with private methods only
  - d) Be used as a base class and a derived class simultaneously
  
2. What is polymorphism in Python?
  - a) The ability of a class to inherit from multiple classes
  - b) The ability of a class to have multiple methods with the same name**
  - c) The ability of a class to hide its implementation details
  - d) The ability of a class to access variables of another class
  
3. What is the purpose of the `__init__` method in a Python class?
  - a) It is used to initialize class attributes.**
  - b) It defines the inheritance hierarchy.
  - c) It is required for encapsulation.
  - d) It handles exceptions in the class.

---

Solutions for [Figure 3-23, "Checkpoint \(2 of 2\),"](#) on page 3-54



## Checkpoint solutions (2 of 2)

### Fill in the blanks:

1. \_\_init\_\_ is a special method in Python classes that is automatically called when an object is created.
2. A object is an instance of a class.
3. In Python, a class is a blueprint for creating objects.
4. Encapsulation in Python helps to hide the internal state of an object and restrict access to certain attributes by using access modifiers or private variables.

### True/False:

1. Inheritance allows a class to acquire properties and behaviors from multiple parent classes in Python. **False**
2. Encapsulation helps in achieving data hiding by restricting access to certain attributes and methods. **True**
3. Polymorphism allows a single function or method name to be used for different types of objects in Python. **True**

---

## Unit 4, "Error Handling and Exception Handling"

Solutions for [Figure 4-22, "Checkpoint \(1 of 2\),"](#) on page 4-47



IBM ICE (Innovation Centre for Education)

## Checkpoint solutions (1 of 2)

### Multiple choice questions:

1. The \_\_\_\_\_ mode in file handling is used to read the contents of a file.
  - a) Read
  - b) Write
  - c) Open
  - d) All of the above
  
2. In Python, the with statement is used to ensure that a file is properly \_\_\_\_\_ after operations.
  - a) Closed
  - b) Opened
  - c) Executed
  - d) None of the above
  
3. The \_\_\_\_\_ method in Python overwrites the entire file content if the file already exists.
  - a) Write()
  - b) Read()
  - c) Both a and b
  - d) None of the above

---

Solutions for [Figure 4-23, "Checkpoint \(2 of 2\),"](#) on page 4-48



IBM ICE (Innovation Centre for Education)

## Checkpoint solutions (2 of 2)

### Fill in the blanks:

1. To read the contents of a file line by line, the method readline is used.
2. The raise keyword in Python is used to define a new exception
3. An exception is an unusual event or runtime error that occurs during the execution of a program.
4. The try block in Python allows handling exceptions and executing code even if an exception occurs.

### True/False:

1. The seek() method in Python is used to move the file cursor to a specified position. **True**
2. The else block in a try-except statement is executed only if an exception occurs. **False**
3. A single except block in Python can handle multiple different types of exceptions. **True**

---

## Unit 5, "Introduction to Data & Business Analytics"

Solutions for [Figure 5-13, "Checkpoint \(1 of 2\),"](#) on page 5-41



## Checkpoint solutions (1 of 2)

### Multiple choice questions:

1. Which of the following regular expressions matches a phone number in the format (123) 456-7890?
  - a) \d{3}-\d{3}-\d{4}
  - b) (\d{3}) \d{3}-\d{4}
  - c) \d{3} \d{3}-\d{4}
  - d) [0-9]{3}-[0-9]{3}-[0-9]{4}
2. What does the '^' symbol represent in regular expressions?
  - a) Matches any single character
  - b) **Matches the beginning of a string**
  - c) Matches the end of a string
  - d) Matches any word character
3. Consider two modules "module1.py" and "module2.py" with the following contents:  
When module2 is executed, choose the correct output from the options given below:
  - a) 0
  - b) No output
  - c) Error
  - d) 120

---

Solutions for [Figure 5-14, "Checkpoint \(2 of 2\),"](#) on page 5-42



IBM ICE (Innovation Centre for Education)

## Checkpoint solutions (2 of 2)

### Fill in the blanks:

1. The question mark (?) in a regular expression denotes optional matching for the preceding character or group.
2. The asterisk (\*) in a regular expression indicates zero or more matching for the preceding character or group.
3. The plus sign (+) in a regular expression signifies one or more matching for the preceding character or group.
4. To check if a pattern matches the beginning of a string, the re.match() function is commonly used.

### True/False:

1. Quantifiers in regular expressions, such as \* and +, specify the number of occurrences of a pattern. **True**
2. The sys module provides access to some variables used or maintained by the Python interpreter. **True**
3. Third-party libraries cannot be installed using tools like pip in Python. **False**

---

## Unit 6, "Graphical User Interfaces (GUI) and Web programming"

Solutions for [Figure 6-24, "Checkpoint \(1 of 2\),"](#) on page 6-63



IBM ICE (Innovation Centre for Education)

## Checkpoint solutions (1 of 2)

### Multiple choice questions:

1. Which Python GUI framework is known for its simplicity and is included in the standard library?
  - a) PyQt
  - b) **Tkinter**
  - c) Kivy
  - d) wxPython
2. In GUI programming, what does the term "event handling" refer to?
  - a) Designing graphical elements
  - b) **Handling user interactions**
  - c) Defining colors and fonts
  - d) Structuring the application logic
3. Which of the following cannot be processed by a CGI script?
  - a) User form data
  - b) **Server configuration files**
  - c) Environmental variables
  - d) Database queries

---

Solutions for [Figure 6-25, "Checkpoint \(2 of 2\),"](#) on page 6-64



IBM ICE (Innovation Centre for Education)

## Checkpoint solutions (2 of 2)

### Fill in the blanks:

1. In GUI development, the acronym GUI stands for Graphical User Interface.
2. Buttons and Labels are examples of widgets commonly used in GUI programming for creating buttons and displaying text, respectively.
3. Two types of CGI interfaces are CGI programs and server-side API calls
4. The header line "Content-type: text/html" within an HTTP response indicates the type of content being sent.

### True/False:

1. In GUI programming, widgets are graphical elements like buttons and labels that users interact with. **True**
2. Event handling in GUI programming refers to the process of designing visual elements but does not involve user interactions. **False**
3. CGI stands for Common Gateway Interface. **True**

---

## Unit 7, "Python Applications"

Solutions for [Figure 7-49, "Checkpoint \(1 of 2\),"](#) on page 7-99



## Checkpoint solutions(1 of 2)

### Multiple choice questions:

1. Which Pandas method is used to display the first few rows of a DataFrame?
  - a) `head()`
  - b) `tail()`
  - c) `sample()`
  - d) `describe()`
2. What is the correct SQL syntax to insert a new record into a table named "customers"?
  - a) `INSERT INTO customers (name, age) VALUES ('John', 35);`
  - b) `ADD RECORD TO customers (name, age) VALUES ('John', 35);`
  - c) `CREATE NEW ENTRY IN customers (name, age) VALUES ('John', 35);`
  - d) `UPDATE customers SET name = 'John', age = 35;`
3. You are performing data cleaning and exploration using Pandas in Python. Which Pandas method is commonly used to check for missing values in a DataFrame?
  - a) `detect_missing()`
  - b) `check_null()`
  - c) `isnull()`
  - d) `missing_data()`

---

Solutions for [Figure 7-50, "Checkpoint \(2 of 2\),"](#) on page 7-100



IBM ICE (Innovation Centre for Education)

## Checkpoint solutions (2 of 2)

### Fill in the blanks:

1. The two primary data structures in Pandas are [Series](#) and [DataFrames](#).
2. To handle missing values in a DataFrame, you can use methods like [fillna](#) or [dropna](#).
3. To connect to a database in Python, you typically use a [database connector/adapter](#) library.
4. The SQL keyword used to retrieve data from a database table is [SELECT](#).

### True/False:

1. NumPy arrays are multi-dimensional matrices that offer efficient data storage and manipulation for numerical calculations. [True](#)
2. In socket programming, a server socket is responsible for initiating communication with the client. [False](#)
3. JSON and Pickle are both commonly used serialization formats in Python, but Pickle is recommended for transmitting data over a network due to its human-readable structure. [False](#)

---

## Unit 8, "Advanced Cloud Topics and Case Studies"

No checkpoint solutions in this unit.





© Copyright IBM Corp. 2024